Object-oriented Modelling for Scientific Computing

Euan Russano and Elaine Ferreira Avelino

ARCLER PRESS

OBJECT-ORIENTED MODELLING FOR SCIENTIFIC COMPUTING

Object-oriented Modelling for Scientific Computing

Euan Russano and Elaine Ferreira Avelino

BIBLIOTEX
Digital Library
www.bibliotex.com

Object-oriented Modelling for Scientific Computing

Euan Russano and Elaine Ferreira Avelino

www.bibliotex.com
email: info@bibliotex.com

e-book Edition 2018

ISBN: 978-1-98460-180-3 (e-book)

In Collaboration with Arcler Press. Originally Published in printed book format by Arcler Press with ISBN 978-1-77361-281-2

About the Authors

Euan Russano

Euan Russano was born in Minas Gerais, Brazil. He is a Chemical Engineer since 2012 by the Rural University of Rio de Janeiro (UFRRJ). He obtained his Msc. in 2014 at UFRRJ, in the area of Chemical Engineering, with specializazion in Process Control. In 2014, Russano began to develop his PhD at the University Duisburg-Essen (UDE) in the field of Water Science. His carrer was initiated in a Petrobras project in the Polymers Laboratory, at UFRRJ. From 2012 to 2014 he worked in the Fluids Flow Laboratory (Petrobras/ UFRRJ) with oil well pressure control. Since 2014 he works as an research assistant at the University of Duisburg-Essen, with water systems identification and control.

Elaine Ferreira Avelino

Elaine Ferreira Avelino was born in Rio de Janeiro, Brazil. She obtained her Bsc in Forestry Engineering at the Rural University of Rio de Janeiro (UFRRJ) in 2007 and Msc. in 2012 at UFRRJ, in the area of Forestry and Environmental Sciences, with specializazion in Wood Technology. She started her career in the Secretary for Environment of Rio de Janeiro, with Urban and Environmental Planning. Elaine was a professor of Zoology, Enthomology, Forestry Parasithology and Introduction to Research at the Pitagoras University. Since 2013 she works as an international consultant for forest management and environmental licensing.

Contents

List of Abbreviations

GUI _ Graphical User Interface

IDE – Interactive Development Environment

IDLE – Integrated Development Environment

MATLAB – Matrix Laboratory

ODE – Ordinary Differential Equation

OOP – Object-Oriented Programming

PDE – Partial Differential Equation

UML – Unified Modelling Language

List of Figures

List of Tables

Preface

The main audience of this book are mathematics, biology, physics and engineering students interested in acquiring more knowledge in scientific computing by using object-oriented programming (OOP).

OOP is present in many different programming languages. However, not all of them can be easily used in scientific computing. Therefore, in this book we show the application of OOP technique using C++, Java, Python and Matlab. These languages stand among the most popular ones to solve scientific problems, especially the numerical ones.

The whole book is divided into 4 main sections.

Section 1 gives a brief description of the basic concepts of OOP, terminology and the history of its development. The second section introduces scientific computing and some simple algorithms in numerical methods are presented, which lies among the most common types of problems that the scientist may face.

Section 3 encompasses the major part of the book and contains the practical techniques for the development of object-oriented software solutions. The first tool presented is the Unified Modelling Language (UML), which is not a programming language, but a tool to develop the concepts in software, documenting it and making easier the implementation of necessary algorithms and methods, to satisfy the main objectives of the software.

The second tool presented is the C++ programming language. In the chapter a introduction to the basic features of the language is given, and an introduction to the object-oriented features of it. The same guideline is followed for Matlab, Java, Python and Modelica.

The Section 4 shows practical applications of scientific computing using object-oriented approach to solve problems. Specifically, it is presented the application of this technique to model the famous Predator-Prey model, or Lotka-Volterra model, which represents the relationship between a prey and its predator in a system under certain constraints. The second application is the description of the OpenMDAO tool for modelling and analysis of mathematical problems. The last application consists into modelling a system of tanks, starting from a functional programming and upgrading it until it reaches an object-oriented approach.

Finally, it provides some suggestions of further reading. After, the references used along the book are presented.

Author

SECTION 1

CONCEPTS ON OBJECT-ORIENTED PROGRAMMING

INTRODUCTION

The history behind the object-oriented programming

The evolution of programming languages can be backtracked to the middle 50's, when it started to be developed the so called first-generation languages. Some of which can be mentioned:

- FORTRAN I
- ALGOL 58
- Flowmatic
- IPV V

These first programming languages were used primarily for scientific and engineering computing. Because of that, the syntax and vocabulary of the languages are almost only mathematical expressions, thus providing already an advantage for programmers from these epoch, freeing them from the intricacies of assembly or machine language.

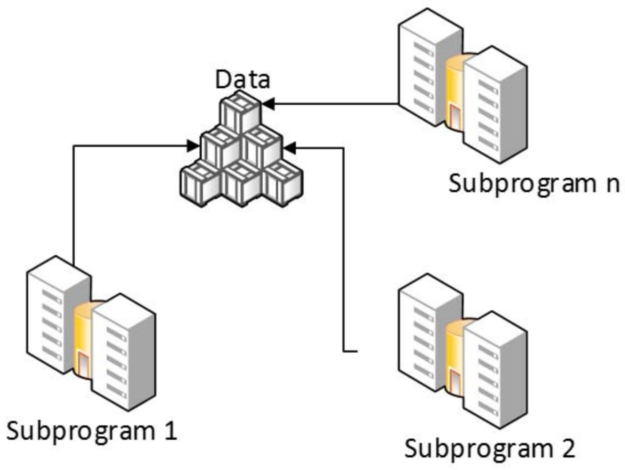


Figure 1: Topology of first-generation programming languages.

The second-generation languages dates from the years 1959-1961, and their focus were on algorithm abstraction. Some that may be mentioned are:

- FORTRAN II
- ALGOL 60
- COBOL
- Lisp

FORTRAN II shift the focus from mathematical expressions to incorporating subroutines, and to have a separate compilation. ALGOL 60 introduced programming with block structures and different data types. COBOL brought the feature of data description and file handling. Lisp had as special features the capabilities of list processing, pointers and garbage collection.

As an example, to calculate the area of a triangle using FORTRAN II, the following program can be used:

```
C AREA OF A TRIANGLE - HERON’S FORMULA
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
C INPUT ERROR DISPLAY ERROR OUTPUT CODE 1 IN JOB CONTROL LISTING
```

```
    INTEGER Q,W,E
    READ(5,501) Q,W,E
 501 FORMAT(3I5)
    IF(Q.EQ.0 .OR. W.EQ.0 .OR. E.EQ.0) STOP 1
    S = (Q + W + E) / 2.0
    AREA = SQRT( S * (S - Q) * (S - W) * (S - E) )
    WRITE(6,601) Q,W,E,AREA
 601 FORMAT(4H Q= ,I5,5H  W= ,I5,5H  E= ,I5,8H  AREA= ,F10.2,
   $13H SQUARE UNITS)
    STOP
    END
```

Which is, although simple, not very easy to understand. The important point is that, the computer at that time had a special boost in performance, and the economics of computer industry meant that the solutions of more problems could be addressed using these available resources. That was especially true for the business applications.

The focus shifted from the use of programs to solve mathematical problems, to telling the computer what to do. As instance: read a file, write one line, close the file, show a report.

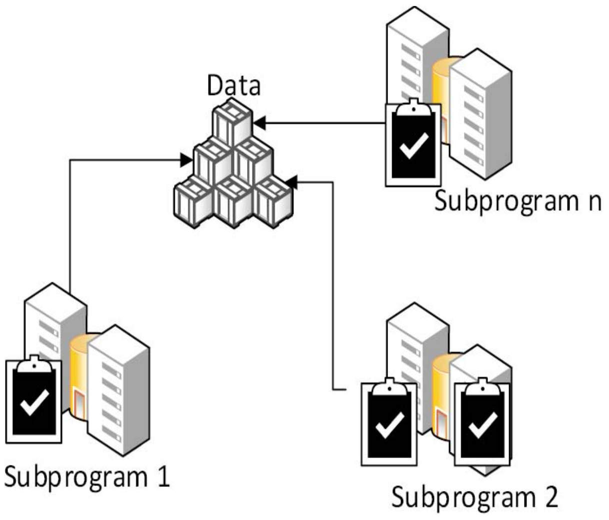


Figure 2: Topology of second-generation programming languages.

The introduction of transistors as a brand new technology, as well as the integrated circuit made the prices of computers to drop significantly, while the processing capacity grown almost exponentially. With these factor, by the middle to the late 60's, a third generation of programming languages rise, allowing data abstraction. Some examples of these new languages are:

- PL/I
- ALGOL 68
- Pascal
- Simula

PL/I was a mixture of FORTRAN, ALGOL and COBOL, incorporating the features of each of them in a single language. Simula was the first one to bring the concept of classes and data abstraction.

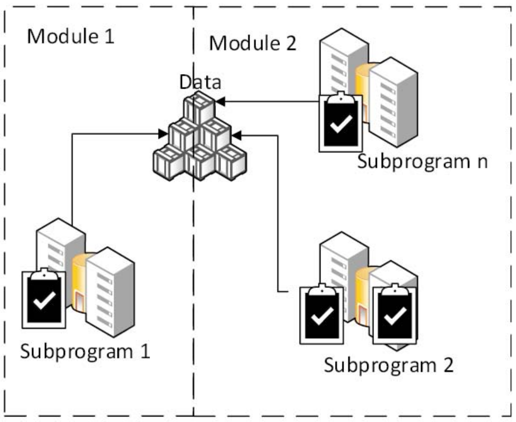


Figure 3: Topology of third-generation programming languages.

The size and complexity of programs started to become bigger and bigger, revealing the inadequacies of earlier languages. At this time programmers started to see that an object-oriented approach of programming could deal much easier with highly complex systems, by partitioning them in smaller parts. Some languages that worth mentioning are:

- Smalltalk 80
- C++
- Ada83

Smalltalk 80 was one of the first pure object-oriented programming languages. Nevertheless, others also incorporated that feature, such as

C++ and Ada83. The boom on developments at this direction dates from 1980-1990, and it drastically increased the productivity and the ability of component reuse.

From this time on, a diversity of languages were developed with focus on object-oriented programming, and that`s our focus on this book.

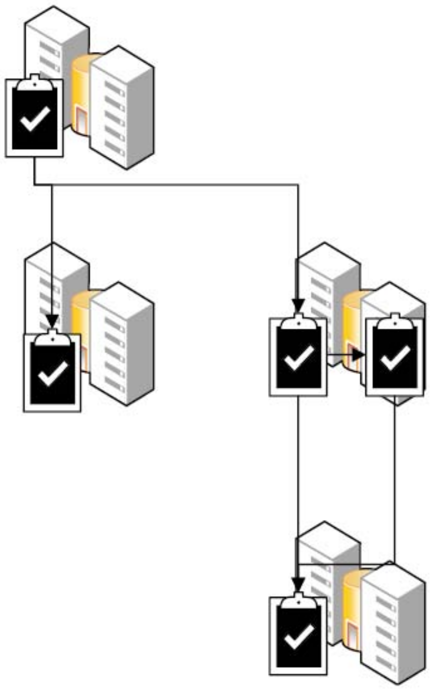


Figure 4: Topology of object-oriented and object-based programming languag-es

Outlook on Complexity of Systems

Is it possible to say that a system is, in its essence, simple? That is a very strict affirmative, for even a single cell, or the smallest amount of sand possess an enormous quantity of atoms, protons, neutrons, electrons, not to mention elements of even smaller sizes and consequently with greater quantity.

Nonetheless, the way that a system is seen by different users, or observers, and the features of that system which are important to them, may make it possible to see a system in a very simple way. For example, a cat in the eyes of a child may be simply a pet, which one can input some feed, and one can play with it. On the other hand, the same cat in the eyes of a veterinary surgeon is seen in a very different manner. The surgeon has noted that the cat suffers from kidney stones, and a surgery may be necessary, which involves knowing how the cat's heart will support the surgery, as well as if some other organs may need to be taken special care during the surgery.

Or just imagine a "simple" water tank. In the eyes of an amateur swimmer, it is important to have enough width or length so he can practice the sport, and also enough height so one can also dive. A strict value for those are not well defined, but depends also on the size and the abilities of the swimmer. On the other hand, in the eyes of a process engineer, he may have planned to use the same tank in a polymerization process. It is important that the water tank can be closed on the top, so it can admit some pressure. It is also important to rightly locate the input and output pipes, so after the polymerization process, the product can be safely removed to the full and new material can be admitted. Besides that, it is necessary to choose the right mixing paddle to the dimensions of the tank, which have to be well known, not only because of that but also to know the residence time of the material. And that is just a brief summary…

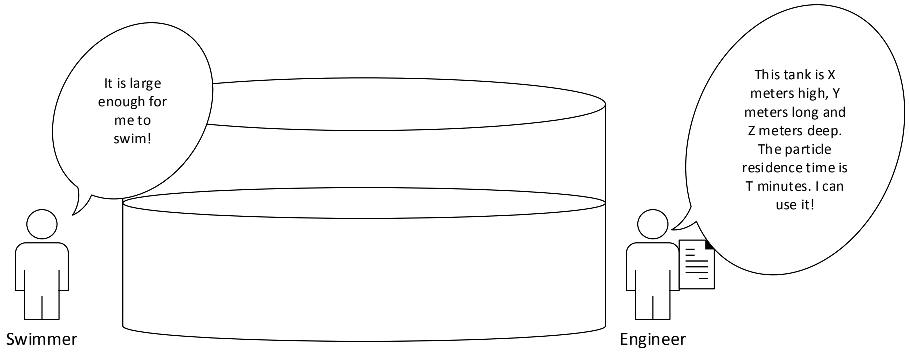

It is large enough for me to swim!
This tank is X meters high, Y meters long and Z meters deep. The particle residence time is T minutes. I can use it!
Swimmer
Engineer

Figure 5: The complexity of a system depends on the observer intention.

This illustrates that, for different users with different intentions, a system can be interpreted in a simple, or in a complex way.

Characteristics of a Complex System

Systems have different levels of complexity. Nevertheless, Booch *et al.* (2007) mention five attributes that are common to all complex systems:

Hierarchic Structure

It is logical to think that a big, complex system can be decomposed in smaller, simpler parts. Interpreting each of these parts as a component itself helps one to understand the big picture.

For instance, the water tank mentioned above can be divided in simpler, smaller components as shown below:

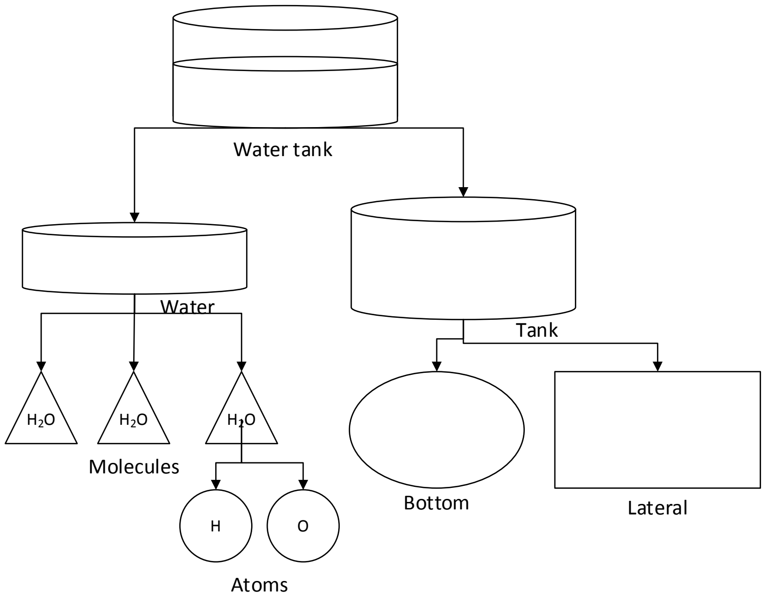


Figure 6: Hierarchic Relationship of a complex system.

One can realize that every system can be decomposed in simpler subsystems, and every system is part of a larger system, whereas the behavior of systems depends on it parts and the relationship among them.

Relative Primitives

Down to what component a complex system can be seen? The choice of the primitive components of a system depends on the observer or user and it is highly arbitrary.

For the water tank above, a swimmer may only be interested in knowing the size of the tank and the water temperature, so he can practice some sport. On the other side, a chemist who wants to use the water tank to do some experiments may want to know what components are inside the water, as well as if the material that forms the tank will react with the product he wants to test. This can go to the level of molecules and ions, their concentration and interaction. Another person may just want to use the water tank to have drinkable water, so the material or dimensions of the water tank are useless and just the volume of water and the substances mixed with it are important for him.

Separation of Concerns

A system's dynamics is defined by its inter- and intracomponent linkage. The latter is assumed to be much more stronger than the first, what makes it possible for an observer to analyze a system, with a clear definition of how to separate its parts. The interaction inside the components are called high frequency dynamics, and they involve the structure of the component, while the interaction between components is referred to as low-frequency dynamics.

Common Patterns

A complex system is not formed only by a variety of different subsystems. Rather, it is composed of patterns, or subsystems which are of the same kind, but they are arranged in a variety of combinations and arrangements. For instance, the water in the tank above is composed of molecules that have 2 atoms of hydrogen for 1 atom of oxygen. These patterns are repeated billions of billions of billions of times until the whole volume of water is formed. Of course, the water may also be contaminated by other substances, but even those are formed by patterns of the same atoms, which are all formed by protons, neutrons, electrons, etc.

Stable Intermediate Forms

Complex systems change over time. Nonetheless, these changes can be much easier understand if the system can be seen in stable intermediate forms. According Gall. (1986), working complex systems must have evolved from a simpler system that also works. A complex system designed from scratch is bound to fail.

As such systems evolve, complex components become primitive of even more complex systems, as in biological systems, which are formed by cells, highly complex systems.

PRINCIPLES AND TERMINOLOGY

Definition of Class and Object

The words object and class are broadly used in programming area, and vendors of database, CASE tools and programming languages tend to use it as a term to attract clients. Many of these does not really knows what the word OO (object-oriented) means.

According Yourdon (1994), a system built with object-oriented methods is one whose components are encapsulated chunks of data and function, which can inherit attributes and behavior from other such components, and whose components communicate via messages with one another.

Regarding Classes, they can be defined as a template, some code which is used as a basis to create objects, providing initial values of states, default components common to the derived objects, and defining the behavior of the derived objects.

The objects are basically things, which behaves according the instructions of the class that it belongs. For instance, a dog is an "object", or instance of a class Mammals. The same apply to cats, horses, whales and so on. By this example, one can see that there are levels of abstraction regarding classes and objects. The objects pertaining to a certain class may have different behavior because they actually are directly derived from subclasses of the main class.

Firesmith (1993) defines object as a software abstraction that models all relevant aspects of a single tangible or conceptual entity or think from the application domain or solution space. An object is one, or the primary software entities in an object oriented application, typically corresponds to a software module, and consists of a set of related attribute types, messages, exceptions, operations and optional component objects.

The object-oriented programming emanates from the programming language Simula, which was a tool developed to simulate processes of the real word. In this sense, the objects in Simula really were representations of real things in the world.

Nevertheless, it was only when the programming language Smalltalk was introduced, that the term object-oriented programming came forth. The whole language was building around the concept of classes and objects, as the authors were fascinated by this technique. The Smalltalk language considers everything an object, from one number to a very complex system. In essence, it sees an object as something which can have some states, and which can perform some actions. In summary:

Object = state + behavior

Definition of Abstraction

The abstraction concept derives from the necessity of interpreting a complex system in a simpler way, by neglecting dynamics or characteristics which are not predominant in the system. Virtually, any piece of software incorporates some level of abstraction, because the programmer hides all but the relevant data to define the object, reducing thus the complexity of the system and increasing efficiency.

Abstraction can be applied with two different focuses:

- Control abstraction
- Data abstraction

The first (Control abstraction) refers to the interpretation of the actions in a meaningful and simple manner, incorporating important actions and neglecting those who has little or no effect in the system being modelled.

The second (Data abstraction) refers to way data is structured, so each data has its own type according the programmer necessity. As an example, a list can be seen as an abstraction of a sequence of items, indexed by their position.

As a general example, consider the following model of a car:

```
#----------------------------------------------------------------------
Model Car
        Properties
                Integer wheels = 4;
                Integer seats = 2;
```

```
End Car;
#---------------------------------------------------------------------
```

What the above model describes about the car? Even someone without knowledge about programming can see that the car **model** has two specifications: wheels (an integer, i.e 1,2,3,…) specified as four and seats, also an integer, specified as one. But what is the color of the car? What is the horse-power? The model does not say, because the programmer has decided that, for the system under analysis, it only important to know the number of wheels of the car and the number of seats on it. This shows how abstraction concept can be used to simplify a system up to the level which is desired according the problem under analysis.

We can also provide one example of an abstraction of a dynamic model. Consider also a model of a car, but of the following format:

```
#---------------------------------------------------------------------
Model Car
        Properties
                Float position
        Methods
                Function drive_car (time = 10.0, velocity = 1.0 )
                        position = position + velocity * time;
                End drive_car
End Car;
#---------------------------------------------------------------------
```

In this case, the **class** car has one single attribute: the position of it. But a function, or action was incorporated, the drive_car function, which is used to change the position of the car, by inputting the time of driving, as well as the average velocity of the car. The new position is calculated with the simple equation for uniform movement:

$s = s_0 + v_0 * t$

Where:

s – final position

s_0 – initial position

v_0 – average velocity

t – time

As in the previous case, one does know what is the model of the car, the color, how big it is. That is because these characteristics are not relevant, according the programmer criteria, to represent the system under interest.

Definition of Encapsulation

An encapsulated information is something that is not clearly seen in the implementation of the object. It is somewhere hidden , so the other components of the system are not aware of it, or can not use it unless it is specified so. In the practical sense, it means that the properties, methods and any other characteristic of the object are packaged together.

This is a huge advantage in object-oriented modelling, for the programmer can control which piece of information in an object will communicate with which piece of other object. This is usually done in the form of sent messages that are sent to specified methods or function of the receiving object.

As an example, imagine a car model. As driver gives as input if the car is on or off, the Steering wheel angle, the accelerator, the brake and the car gear. So a simple representation of this model would be:

```
#-------------------------------------------------------------------
Model Car
        Properties
                Boolean engine
                Float Steer_wheel_angle
                Float accelerator
                Float brake
                Float gear
        Methods
                Function turn_on
                Function turn_off
                Function car_drive
End Car;
#-------------------------------------------------------------------
```

When the driver turns the car engine on (by setting the Boolean engine to TRUE), then the function turn_on calls another model which is the start_engine model. The start_engine model represents just, as the name reveals, the start engine motor, which is responsible of admitting fuel and air into the motor and compressing it. A simple representation of this model can be stated as:

```
#-------------------------------------------------------------------
Model Start_Engine
        Properties
                Boolean turn_on
        Methods
                Function admit_air
                Function admit_fuel
                Function compress
End Car;
#-------------------------------------------------------------------
```

This model has a single property, turn_on (TRUE or FALSE), which is called by the turn_on function of the car, in order to run its functions, which are to admit air into the motor, admit the fuel and compress it, and after combustion the motor can work by itself, and the start_engine can be turned off.

Once the motor has started, it is time to accelerate so the car can move. This is done by the function car_drive in the car model, which may call other objects such as engine, differential, wheels, and so on. An encapsulated, object-oriented representation of this car model would be:

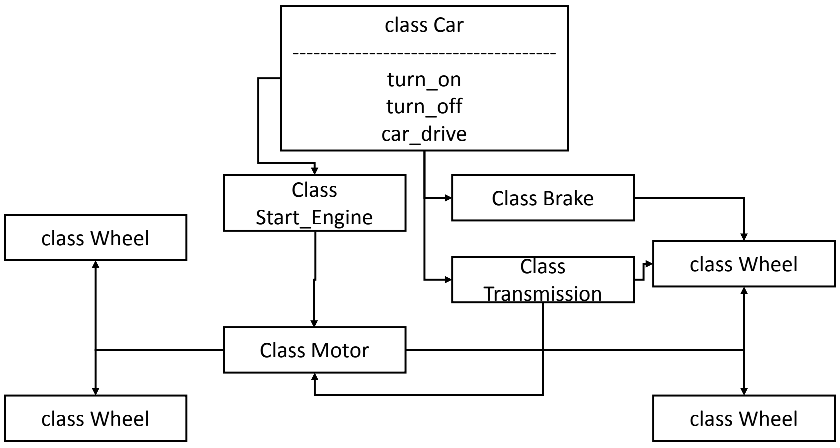


Figure 7: Example of Encapsulation – Car model

Each different object is not aware of what the other one is doing, but they receive or send tasks according the definitions on its structure.

Definition of Inheritance

Inheritance is the principle that an object incorporates all or part of the definition of another class, usually referred to as superclass or parent-class. This creates a hierarchy of structures, where some objects inherits from one superclass while other are derived from another class, and so on. The classes in this hierarchy are referred to as generalized/specialized structures. That is because, as we search for common characteristics of different objects, we arrive at some moment in a concept which is common to a variety of objects of interest.

These generalized, common objects can also produce derived objects by combination, which defines a multiple inheritance. For instance, a class **Daisy** can be defined as a plant, as well as a Flower, depending on the intention of the user. To define this class, one can use multiple inheritance concept.

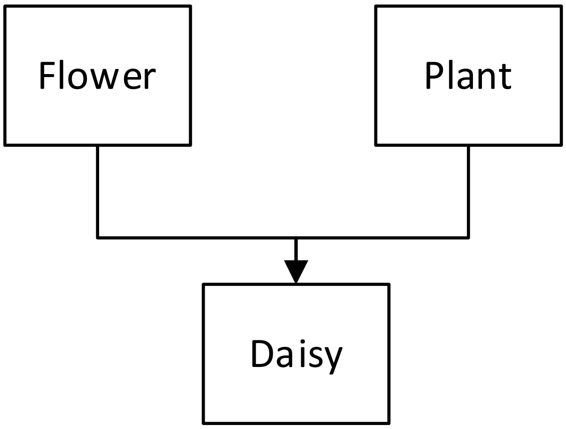


Figure 8: The Daisy Class as an example of Multiple Inheritance.

On the other hand, on single superclass can also generate multiple subclasses. For example, consider the different types of car: convertible car, mini SUV, urban car, sport car. All of this type of cars can be a class derived from a superclass Car, which describes the basic features common to all of these models, such as the fact of having 4 wheels, one steering wheel, and so on.

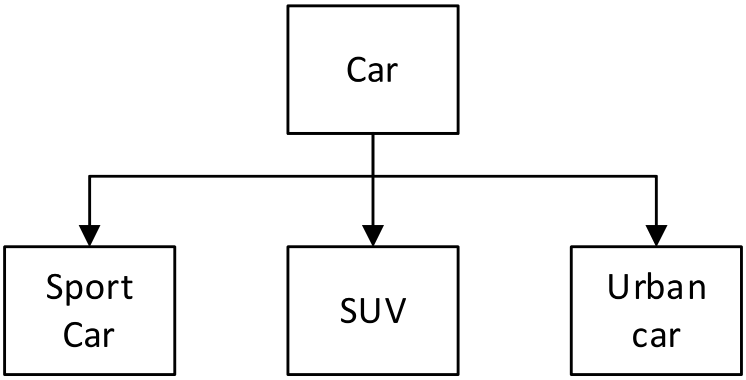


Figure 9: The Car Superclass as an example of Multiple Subclasses.

This concept makes it easier when programming, for it avoids the repetition of code already written. It also makes easier to analyze a system, for its is know what is common to each of objects on it, and what is different, by relating to their hierarchy

SECTION 2

SCIENTIFIC COMPUTING PRINCIPLES

INTRODUCTION TO SCIENTIFIC COMPUTING

Scientific computing is a tool used to solve a variety of problems in science and engineering fields. The basis of this process comes from a knowledge over the phenomenon under study, known as model. Knowing how the phenomenon behaves (model) and how to develop an algorithm able to do predictions of the phenomenon is the basis of the scientific computing.

According Bindel and Goodman (2009), the challenge of scientific computing draws of mathematics and computer science, being necessary discipline and practice in order to overcome them. The same problem can be solved using different algorithms and the testing of such involves breaking it procedure by procedure. Accuracy, stability, robustness and performance are some of the factors considered when developing algorithms and programs.

There are nowadays a variety of tools available for developing such algorithms. Programming environment and debuggers, visualization, profiling, pre-compiled libraries are some of these which helps one to create high-quality software solutions.

One important feature regarding scientific computing is that, in most problems involving continuous mathematics, as in derivatives, integrals or nonlinearities, the solution of a problem will not be exact using a finite number of steps. Therefore, an iterative process is required, which will converge to a solution with reasonable accuracy, depending on the admissible error. According Heath (2013), the challenge is to find rapidly convergent iterative algorithms, which will produce accurate resulting approximations. In some cases, the iterative algorithm may be so fast that it may even be preferable over analytical methods for linear systems which will require more computational resources.

The solution procedure using scientific computing usually involves the following steps:

- Development of a mathematical model of the system under investigation
- Development of the algorithms necessary to solve the problem numerically
- Implementation of the algorithm in a computer software, or development of a computer software to solve the problem
- Run the simulation
- Store the results of the simulation
- Represent such results in a way they can be analyzed and validated
- If the validation fails, repeat all or some of the steps above until the validation succeeds

Computational resources are not infinite, so the seeking of a solution to a mathematical problem usually involves fitting this problem to the availability of resources. This means simplifying a problem, translating a difficult and complicated system into a simpler one which provides the same solution, or a closely related one. According Heath (2013), some procedures for simplifying mathematical problems involve:

- Replacement of infinite-dimensional spaces into finite-dimensional spaces

- Replacement of continuous terms with discrete ones, such as the replacement of integrals, derivatives by finite sums and finite differences.
- Replace differential equations with algebraic equations
- Replace non-linear problems with linear ones
- Replace high-order systems with low-order systems
- Simplify complicated functions with simpler ones

COMMON MATHEMATICAL PROBLEMS

System of Linear Equations

The Linear systems of equations are widely spread in scientific problems in areas such as biology, chemistry, physics, and engineering. The fundamental problem involving systems of linear equations is to find the value of a set of unknown variables given a set of linear equations. If the system of equations is the same as the number of variables then the system has a unique solution for the variables. In the case that the number of equations is less than the number of variables, then the system has many different solutions. If the number of equations is higher than the number of variables, the system has no solution, but approximations can be found.

A linear system of equations can be mathematically expressed according the following equations:

$$a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n = b_1$$

$$a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n = b_2$$

$$\vdots$$

$$a_{n,1}x_1 + a_{n,2}x_2 + \ldots + a_{n,n}x_n = b_n$$

With n unknowns $x_{1,2,..n}$ .The same system can be better represented in the matrix form as follows:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Or in a compact form:

$$A.X = B$$

Where A is the $n x n$ matrix of coefficients, B is the right-hand side vector of length n, and X if the solution vector of length n.

To analyze how linear systems behave, consider a simple 2x2 system of linear equations:

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

If all the values of the coefficient matrix are different from zero, the system can be rewritten in the form of the following two equations:

$$x_2 = -\frac{a_{1,1}}{a_{1,2}} x_1 + b_1$$

$$x_2 = -\frac{a_{2,1}}{a_{2,2}} x_1 + b_2$$

Which defines the slope-intercept equations of two lines in a plane.

The set of solutions (values for x_1 and x_2) consists of all points in the plane where the two lines intersect. For this case, there are three possibilities:

- A unique solution – the lines intersect at a single point in space.
- No solution – the lines are parallel, therefore there is no intersection.
- Infinitely many solutions – the lines are parallel with the same intercept.

These conditions hold for any type of linear system of equations, independent on the size of matrixes. A system is called nonsingular if it has one and only one solution. On the other hand, it is referred to as singular if it has no solution or an infinite set of solutions.

To check if the system is nonsingular, the matrix A must satisfy any of the following conditions:

- A has an inverse, i.e, a matrix denoted A^{-1} exists, such that the following relation holds $A^{-1}A = I$ (identity matrix).
- $\det(A) \neq 0$
- $rank(A) =$ n the rank of the matrix A is the maximum number of linearly independent rows or columns it possesses.
- For any vector $y \neq 0$, $Ay \neq 0$

If any of the conditions above hold, then the system is no trivial, and it either possess no solution or an infinite set of solutions.

Solution using Cramer Rule

According the Cramer's rule, the value of the unknowns in a linear system is given by fractions which the denominator is the determinant of the coefficient matrix and the numerator is the determinant of the coefficient matrix replacing each column by the right-hand side vector of the system.

We exemplify the application by solving the following system of linear equations:

$$\begin{bmatrix} 1 & 2 & 3 \\ -1 & -2 & -4 \\ 5 & 6 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ -11 \\ 28 \end{bmatrix}$$

The first step to find the values of x_1, x_2 and x_3, is to find the determinant of the coefficient matrix:

$$\det(A) = det\left(\begin{bmatrix} 1 & 2 & 3 \\ -1 & -2 & -4 \\ 5 & 6 & 1 \end{bmatrix}\right) = -4$$

The next steps consist on calculating the determinant of the coefficient matrix replacing each column of it by the right-hand side vector. Replacing the columns, one can find the following determinants:

$$\det(Ax_1) = det\left(\begin{bmatrix} 10 & 2 & 3 \\ -11 & -2 & -4 \\ 28 & 6 & 1 \end{bmatrix}\right) = -12$$

$$\det(Ax_2) = det\left(\begin{bmatrix} 1 & 10 & 3 \\ -1 & -11 & -4 \\ 5 & 28 & 1 \end{bmatrix}\right) = -8$$

$$\det(Ax_3) = det\left(\begin{bmatrix} 1 & 2 & 10 \\ -1 & -2 & -11 \\ 5 & 6 & 28 \end{bmatrix}\right) = -4$$

The last step consists on obtaining the value for each unknown variable in the respective order by dividing the determinant of Ax_1, Ax_2 and Ax_3 by the determinant of A. By doing this one obtains the following values for the unknowns:

$$x_1 = \frac{\det A}{\det Ax_1} = \frac{-12}{-4} = 3$$

$$x_2 = \frac{\det A}{\det Ax_2} = \frac{-8}{-4} = 2$$

$$x_3 = \frac{\det A}{\det Ax_3} = \frac{-4}{-4} = 1$$

Curve Fitting

Curve fitting consists in methods used to approximate an unknown function using some sort of algorithm which derives from the hypothesis that the system being fitted behaves according this algorithm (at least in the region under analysis). There are two common methods for curve fitting:

- Interpolation
- Least squares

Interpolation consists on approximating an unknown function using a known one in some point in space contained by known values of the unknown function. For instance, consider a system governed by the following equation, where x are known values given as input to the system and y the output of the function:

$$y = \sin(x)$$

If one visualizes the values of y inside the range of x between 0 and 3, the following figure can be produced.

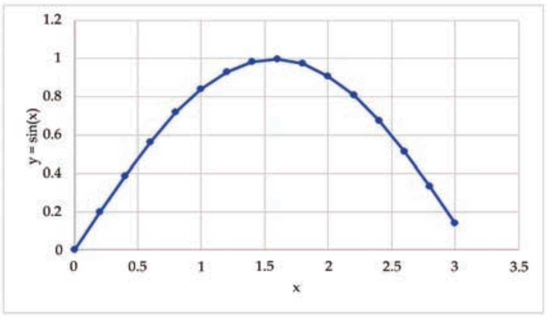


Figure 1: Function y = sin(x).

As already mentioned, the interpolation method is useful to found values at unknown regions of the function located between known values. The most simple interpolation algorithm consists into the linear one. A linear proportion is used to found the unknown value of the function, which can be stated as:

$$\frac{y-y_1}{x-x_1}=\frac{y_2-y_1}{x_2-x_1}$$

Where y is the unknown value of the function at the point x. The values y_1 and y_2 are known values of the function at the points x_1 and x_2 respectively. Rearranging the above equation to make y explicit:

$$y=y_1+(x-x_1)\frac{y_2-y_1}{x_2-x_1}$$

The accuracy of this algorithm for non-linear functions depends on the nonlinearity of the function and the range being interpolated. For instance, consider that the only known values for the function previously mentioned ($y = \sin(x)$) are for x = 0 and x = 1. The following figure can be used to represent this system.

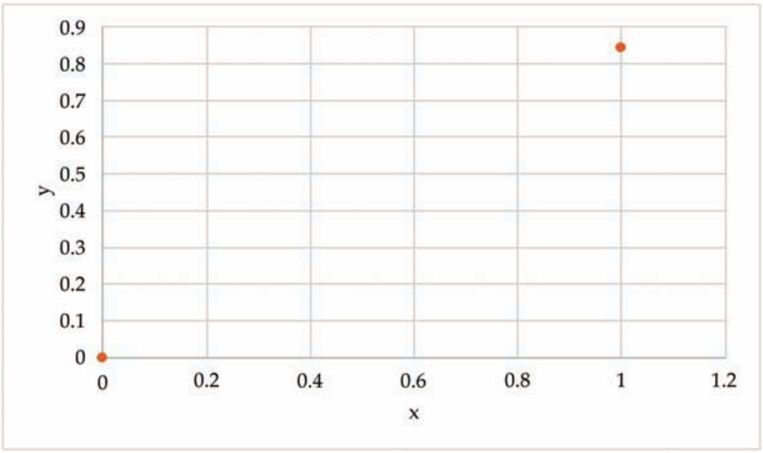


Figure 2: “Unknown“function values at x = 0 and x = 1.

At first different assumptions can be made regarding the behavior of the function between these two points. It may behave as a linear system, a parabolic one or an exponential one, as shown in the figures below.

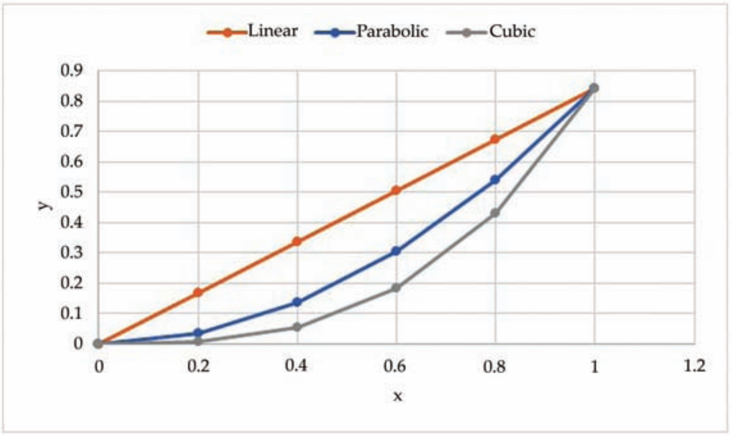


Figure 3: Different assumptions made on the “unknown” function y = sin(x).

Performing the linear interpolation using the previously described algorithm, one can obtain the value for y:

$$y = 0 + (0.5 - 0)\frac{(0.8414 - 0)}{(1 - 0)} = 0.4207$$

The following figure provides a comparison between the interpolated value and the real value of the function (y = sin(x)).

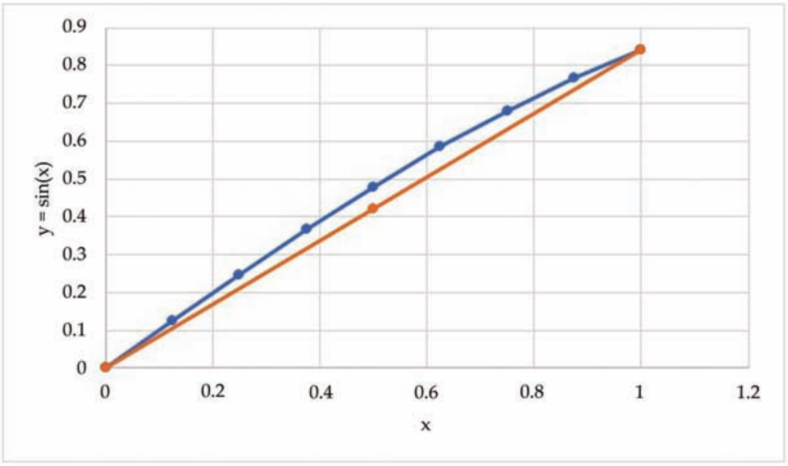


Figure 4: Difference between interpolated value and real value of the function y = sin(x) between x = 0 and x = 1.

Depending on the application, one can say that the approximation of the interpolated algorithm is reasonable and can be used inside this range of the "unknown" function. However, what happens if the interpolation is performed in an extended region of the function? Suppose now that the now value of the function is between x = 0 and x = 2, for which y = 0 and y = 0.909. One desires to know the value of y for x = 0.5 performing linear interpolation. The calculation is done as follows:

$$y = 0 + (0.5 - 0)\frac{(0.909 - 0)}{2 - 0} = 0.227$$

Now comparing the new approximation with the real value of function one can obtain the following figure:

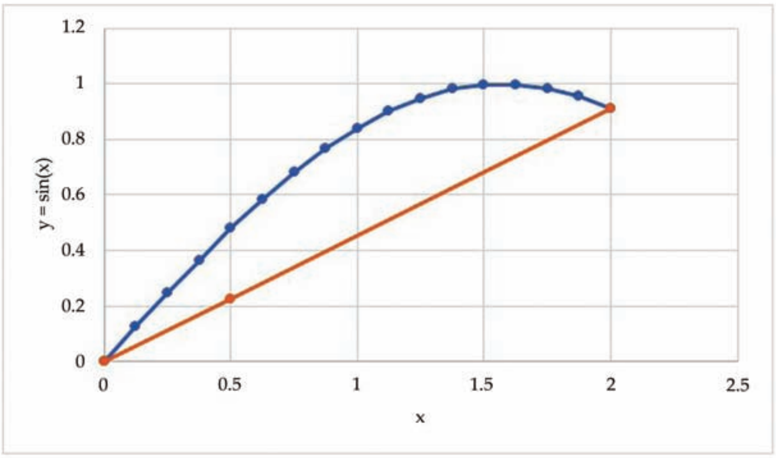


Figure 5: Difference between interpolated value and real value of the function y = sin(x) between x = 0 and x = 2.

The approximation gets much worse as the range of the values used to perform the interpolation increases. This feature is one of the drawbacks of the linear interpolation. Nonetheless, it is a powerful and simple tool for estimation and curve fitting.

SECTION 3 OBJECT-ORIENTED DEVELOPMENT AND PROGRAMMING

OBJECT-ORIENTED PROJECT

The field of knowledge responsible of planning, organizing, securing and managing resources in order to fulfill specific tasks and objectives is referred to as Project Management. The vital challenge of project management is to complete all the engineering project tasks and objectives, without violating the project scope, time and budget. These three are also called project constraints.

The secondary—and more ambitious—challenge is to optimize the allocation and integration of inputs necessary to meet pre-defined objectives (Thapa, 2011)

There exist different models which can support the development process such as Waterfall Model, Spiral Model, RAD Model, Incremental Model, Object Oriented Model etc. Each model has advantages and disadvantages, and the decision to choose a specific one depends mostly on the development team. A second possibility is a combination of different models in order to better fulfill project requirements.

When using an object-oriented design, the main building block of the software becomes the classes and their instances, or objects. The objects are specific things, while the classes are generalizations of the things. For instance, in a software for bioinformatics, a class Lion may define any type of lion, of any gender, age or size. On the other hand, an object Lion defines one specific lion with (possibly, depending on the necessities of the project) defined gender, age, name, size, and so on. In summary, every object has its own state and behavior, while the class describes which states belongs to the its type (without a defined value) as well as it holds the behavior definitions.

According Thapa (2011), the object- oriented approach to software development is decidedly a part of the mainstream simply because it has proven to be of value in building systems in all sorts of problem domains and encompassing all degrees of size and complexity. Furthermore, most contemporary languages, operating systems, and tools are object-oriented in some fashion, giving greater cause to view the world in terms of objects. Object-oriented development provides the conceptual foundation for assembling systems out of components using technology such as Java Beans or COM+. Constructing object – oriented systems is exactly the purpose of the Unified Modeling Language (UML).

Booch (1996) mentions that some advantages of the object-oriented project development is the fact that as a common unit of decomposition is employed, features such as incrementation and iterative process are naturally possible. It is also noticeable that, qualitatively, they demand different kinds of measures. The following four points are major benefits derived from an object-oriented approach:

- Better time to market
- Improved quality
- Greater resilience to change
- Increased degree of reuse

Nonetheless, to fully benefit from this project development approach, it is important to consider five habits (Booch, 1996):

- A ruthless focus on the development of a system that provides a well-understood collection of essential minimal characteristics;
- The existence of a culture that is centered on results, encourages communication, and yet is not afraid to fail
- The effective use of object-oriented modeling

- The existence of a strong architectural vision
- The application of a well-managed iterative and incremental development life cycle

Although this points may appear to be generic in the context of project management, the presence of object-oriented approach in the design process reinforces the attention that should be taken at each point.

The first point, ruthlessness, is special applied in object-oriented design, since a project is supposed to better react to changes in the understanding of the real problem. When this changes are applied to the project, not much effort should be required to tune the project, which means that no many parts of the software are to be necessarily rewritten, for it should support modularity and extensibility.

An efficient object-oriented software development organization is focused in the development, delivery and maintenance of the software products satisfying the users requirements and providing delighting features. Each activity performed during the software development: analysis, design, implementation, quality assurance and documentation are only important if they are ways of achieving the final goal.

A well-organized and efficient object-oriented project should have the following features:

- A collection of classes, well organized into hierarchies
- A well-defined set of collaborations that specifies the different ways that the classes communicate with one another, providing system functionalities.

Naturally, a huge collection of classes and definitions of interactions among them does not necessarily make a good and efficient software. This is only achieved by implementing just the necessary number of artifacts that the software requires. An excess of prototypes, test scaffolding, documentation and team will basically waste resources and decrease code readability and extensibility. The reusable artifacts of every software project include:

- Architecture
- Design
- Code
- Requirements

- Data
- Human interfaces
- Estimates
- Project plans
- Test plans
- Documentation

Jones apud Booch (1996) refers to these artifacts as reusable because they may surpass the lifespan of the project that created them. The same piece of code can be transported from one project to another. The design of an user interface of one application may be also used in a second one. Frameworks can be architectures serving as foundation for an entirely family of programs.

INTRODUCTION TO PROGRAMMING

A computer program can be seen as a series of instruction that defines how some electrical impulses flow inside a computer system. These impulses are not restricted to the computer's memory, but they interact with mouse, keyboard, screen and any other peripheral connected to the machine, in order to produce the tasks that the program issued. These tasks can be from a simple blank screen where the user is allowed to type some text, to high-level games where artificial intelligence is required to challenge the user of the machine.

Software

A software is a computer program itself. It contains not only one, but a collection of instructions, which may be organized into different files, that defines the series of task that the computer should perform. In order to be stored, some sort of medium must be used which is capable of recording the instructions and transmitting it to the computer. The simplest example of a medium storing such instructions would be a simple piece of paper with some code written on it. While this type of medium is easy for a human to read and maybe even to somehow understands what the computer should do, it is not the most viable way for a computer to understand the program. So floppy disks, CD, DVD, USB pen drive are all examples of mediums that can store a software, and are readable by a computer.

Before being used, the software must be read by the memory of the computer, also called Random Access Memory (RAM). This is normally achieved by storing the program in the hard-driver of the computer, which is accessible by the internal memory of the computer through electromagnetic reading of the patterns stored. Such patterns are not easily understandable by humans, for they are a sequence of zeros and ones, such as:

00111001000111

A computer is capable of translating such sequence in some sort of task that it should perform, such as the blink of a screen, the writing or reading of a document, etc. Each type of processor has its own *language*, or way of interpreting the series of ones and zeros. For example, in a Windows system the above sequence may tell the machine to blink the screen, while in a Mac OS it may mean that a document should be printed. This means that each processor has its own *machine language*.

Software development tools

Suppose a programmer wants that a computer sums two numbers, and print in the screen the result of this algebraic operation. How can this be performed with the series of zeros and ones mentioned earlier? That`s not an easy task, and it can become nearly impossible when it refers to complex series of tasks, performed by operating systems or high-level commercial software's.

For that reason, tools were developed, which software instructions are seen as a series of symbols or text that are easier for a human to manage than a binary sequence. These tools convert the textual or graphical instructions developed by the programmer into machine language, which can be read by the computer. For instance, C++ programming language allow developers to see the instructions to a computer in a way very similar to the English language. However, the syntax is developed in a much simpler way, for natural language has natural ambiguity and it may require a good background knowledge from the message recipient to clearly understand what is being said.

Coming back to the example mentioned in the beginning of this subsection, in which the developed wanted the machine to sum two numbers and print it to the screen. In C++ language, this can be simply done by writing the following set of instructions:

```
a = 1
b = 2
c = a + b
```

The example above is not a complete program, but it is an example of how the instructions can be much easier understandable using these development tools rather than the binary sequence, which is the *natural* language of a machine. This does not mean that a computer can directly understand the sequence written above, but it uses a compiler, or translator, to rewrite the instructions above in its language.

In language used to write the code (higher-level) is referred to as *source code*. The language that is read by the machine is referred to as *target code*. A source code can be translated to different target codes, depending on the processor used. This means that for a Mac OS system, or a Windows system, the same instructions written in C++ for instance, can perform the same tasks, although these tasks are internally translated to totally different target codes.

To develop softwares, a series of tools are available. Some of these are:

- Editors

An editor can be from a plain text editor such as Notepad, to a very detailed software with different tools that may help the developer to speed up the program development process. In the editor, the developer writes the instructions as if he was written in text, however following the rules according the programming language used, similarly to natural languages. For instance, the following statement in English:

"The big red train travels to the beautiful station of Amsterdam"

Is grammatically correct in English. However, if we rewrite it as:

"The train big red to the beautiful station travels of Amsterdam"

Is wrong and cannot be easily understandable by another person. In the case of a computer program, the "grammar" must be perfectly well written, otherwise errors are produced or the software does not perform the desired task exactly as wanted.

For example, in computer language, if one wants to create a variable called dog and to attribute the value of 10 to it, in C++ language as well as many other languages this could be written in an editor as:

```
dog = 10
```

On the other hand, is one wants to perform this task, but writes it in the following form:

```
dog is equal to 10
```

This will issue an error in a C++ program, although from a human point of view is clearly understandable that the dog variable is equal to 10. The programming languages have strict rules that have to be follow in order to work properly.

Compiler

A compiler is the piece of tool responsible of translating the code from the source code to the target code, which may not necessarily be machine-language. For instance, MATLAB, a high-level programming language has tools to compile some code into faster C code. The C code is then processed by a C compiler to produce and executable program. The process of compilation follows a set of instructions or procedures which are:

- Preprocessor: — as the name refers to, it processes some header and instructions to modify the content of the source file before the compiler begins.
- Compiler – translates the processed code into target code.
- Linker – It links or combines both the compiled code with the compiler-generated machine code to make a complete executable program.
- Debugger – A tool that makes it possible for a user or a developed to trace back errors in the program or in its execution, even in the level of line by line. The debugger keeps track of the value of variables, objects and other generated elements of the program, so the developed can see it these values are as

expected, and if not, he can know the piece of information that is generating the error.

- Profiler – A tool that keeps track of the performance of the program. With it, a developer can trace with elements are taking more memory and time, and optimize the code so it can run smoothly and faster, without non-usable code.

UNIFIED MODELLING LANGUAGE

The so-called Unified Modelling Language (UML) is a broadly known tool for developing software in general. It implements international industry standard graphical notation to describe and characterize software analysis and design. The use of a standardized notation for software development leaves little or no room for misinterpretation and ambiguity.

The UML derives from the unification of notations developed by Booch, Rumbaugh, Jacobson, Mellor, Shlaer, Coad, and Wirf-Brock, among others (Williams, 2004). It has been accepted as a standard by the Object Management Group (OMG), which is a non-profit organization responsible for distributing object-oriented computing.

The UML is specially useful for object-oriented scientific computing for, as it follows the basic notions of object-oriented programming, it can be used as a standard and a visual tool to analyze and design scientific software.

A model in UML is composed of three basic elements:

- UML building blocks
- Rules to connect blocks
- UML common mechanisms

Regarding the UML building blocks, they can be divided into three types:

- Things
- Relationships
- Diagrams

Things

The Things are the most important building block type in UML language. They can be subdivided into:

- Structural
- Behavioral
- Grouping
- Annotational

Structural things are static components of the model, representing physical elements or concepts that the model possess. The following is a description of the structural elements in UML:

Class: Using the same original concept of object-oriented language, a class defines the attributes and behavior of a set of similar objects in the system. Example: the Dog class defines the behavior of Rex, Daisy and Max, three different "objects" derived from the same class.

The UML representation of the Class is as follows:

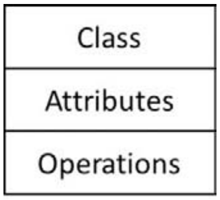


Figure 1: The Class thing representation in UML.

Interface: The Interface specifies the responsibilities of a class, by defining a set of operations attributed to it.

The following figure is the standard representation of the Interface block.

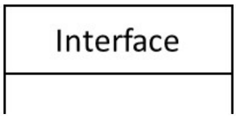


Figure 2: The Interface thing representation in UML.

Collaboration: A block that defines the interaction between two elements in the model. The representation of the Collaboration is as follows.

Figure 3: The Collaboration thing representation in UML.

Use case: This block represents a set of actions performed by a system to reach a specific objective.

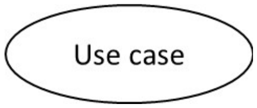


Figure 4: Use Case representation in UML.

Component: This block represents a physical part of the system. The graphical representation of a Component block is done according the following figure.

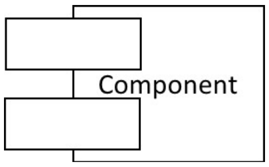


Figure 5: Component block representation in UML.

Node: The Node block is a physical element that exists during the execution of the model. Its representation is done as follows.

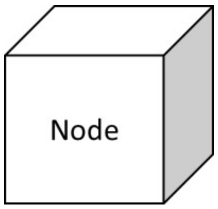


Figure 6: Node block representation in UML.

The Behavioral things are dynamic elements of the UML model. They represent interactions between structural things, as well as the current state of a thing. The following blocks are behavioral things in UML:

Interaction: Is a block consisting of one or more messages exchanged among blocks in order to perform a determined task or to reach a specific objective at simulation time. The following figure is a representation of this block.

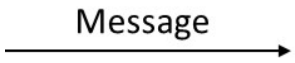


Figure 7: Interaction block representation in UML.

State Machine: This block is used to define the different states a block goes through during its lifetime (simulation time). These states are usually responses to different events issued in the model, generated by external factor.

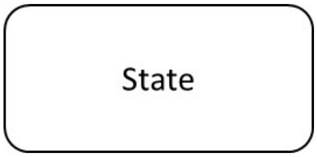


Figure 8: State Machine block representation in UML.

Behavioral and Structural things are grouped together using another type of building block in UML, referred to as grouping thing. There is basically one type of grouping thing, the Package. The UML representation of this building block is shown below.

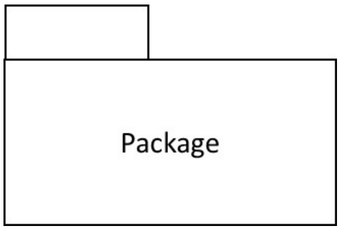


Figure 9: Package block representation in UML.

In many situations, it is useful to add comments, remarks, highlights or any sort of special information in the model, which does not necessarily interact with the model, but it is used as a small piece of information between human-machine. In UML, the An notational things are used to perform this type of procedure. The unique building block in this type of thing is the Note, and its representation in UML notation is shown below.

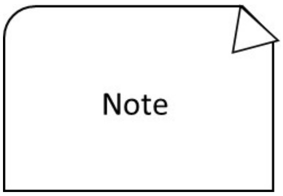


Figure 10: Note block representation in UML.

Relationships

Another very important concept in UML, the Relationship shows how elements are associated with each other. This association is an important feature that determines the functionality of the software.

- There are four different kinds of Relationship in UML:
- Dependency
- Association
- Generalization
- Realization

Dependency: This kind of relationship determined that the changes in one element causes transformation in the dependent element. The representation is done as follows.

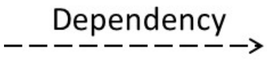


Figure 11: Dependency block representation in UML.

Association: It determines a collection of linked UML elements in the model, describing also how many objects are taking part in that relationship.

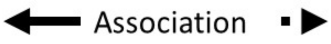


Figure 12: Association representation in UML.

Generalization: The generalization reveals a relationship of specialized – generic element in the UML diagram. It describes a relation of inheritance between the connected objects. For example, from a

"Plant" object to a "Flower" object and a "Tree" object. The following is the UML representation of such block.

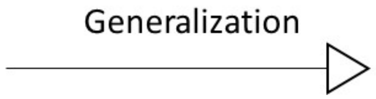


Figure 13: Generalization block in UML.

Realization: The realization block, as the name suggests, reveals a relationship in which one block tells the other what must be realized. The first block does not realize it, but the second one. It is especially useful for the development of interfaces. The following figure shows its representation.

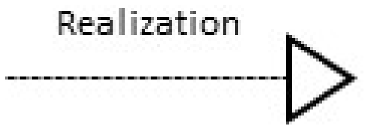


Figure 14: Realization block in UML representation.

Ways of modelling

UML, as a versatile language, allows different approaches for different model types. These different approaches define which tools will be used in which way. Three different ways of modelling, of model types are clearly defined in UML. They are:

- Structural Modelling
- Behavioral Modelling
- Architectural Modelling

Structural Modelling: As the name suggests, this approach is used to capture the static features of a system. It consists of the following elements:

- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Package diagrams

- Composite structure diagram
- Component diagram

The structural model of the system represents the very existence of it, with its components all clearly stated. Therefore, the class diagram, component diagram and deployment diagrams are part of structural modeling, representing the elements and the mechanism to connect them.

Behavioral Modelling: In contrast with the structural modelling, the behavioral one reveals the dynamics of the model, how things interacts and how they change during their lifecycle. It consists of the following diagrams:

- Activity diagrams
- Interaction diagrams
- Use case diagrams

These diagrams are used to show the flow of data and information in the model.

Architectural Modelling: The architectural modelling represents the combination of the structural modelling and the behavioral modeling. It can be defined as the blueprint of the whole system. The Package diagram comes under this modelling structure.

Diagrams

A diagram in UML is an element used to plot all the things and relationships, showing the entire behavior and relations inside the system under analysis. The visualization of these diagrams is the most important aspect of the entire UML model development.

Because the complexity of a system usually cannot be seen from a single perspective, UML possess 9 different diagrams with details specific to each that makes it capable of interpreting the majority of real and human-developed systems. These diagrams are:

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram

- Activity diagram
- Statechart diagram
- Deployment diagram
- Component diagram

Class diagrams

The Class diagrams are developed in two main phases of the model implementation: analysis and design phases. At the analysis phase, the conceptualization of the class is very high, and its representation until this moment may only be a symbolic name with some general description and pseudo-code describing the main operations performed by this class. It may be used to reveal the relationships in the problem domain, but the system implementation is still not clear at his phase.

At the design phase, the class diagrams already reach a more concrete state, with clear attributes and operations, detailing implementation procedures and the relationships existent among the various classes.

There are some conventions for the definition of class diagrams:

- The name of the class has to start with a capital letter (e.g class Dog and not class dog).
- The name of the class is written on the top of the class, in separated compartment (rectangle), in such a way that any one can clearly note that the name identifies that class.
- The second compartment (rectangle) is reserved to the attributes/ data menbers of the class (if any).
- The lower compartment states the methods/ operations performed by such class.
- Optionally, a last compartment can be added to refer to special features of the class that may not fit into attributes/ data or operations.

Taking the above conventions into account, a more concise notation of the class diagram is represented below.

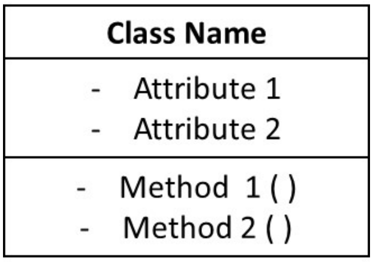


Figure 15: Notation of class diagram.

The former can be exemplified by a generic Box, or Volume, with attributes such as:

- Name, or Identification (not the name of the class, but a name or identification for each instance of the class)
- Height
- Width
- Depth
- Amount of things

And things can be added or removed from the box, so the following operations may be performed:

- Add thing
- Remove thing

Taking all of this into account, the class diagram Box is represented below:

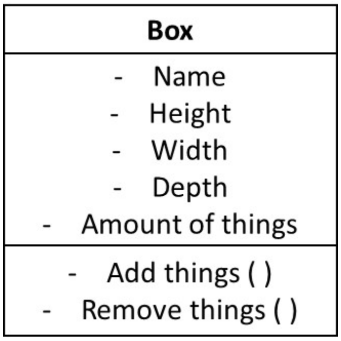


Figure 16: Representation of the Box class diagram.

Object diagrams

Object diagrams are representations of specific instances of classes developed before. The concepts applied to the class diagram are in this sense, the same applied to the object diagram. An important aspect of object diagrams is that they are a static view of the system being modelled, so they represent a snapshot of the system at a particular time of the simulation lifetime.

In a class diagram, the desired concept is represented as an abstract model consisting of classes and their relationship. On the other hand, object diagrams as specific representations of such abstract models, which exists in a particular moment of the simulation (or at any moment), with a concrete nature.

There are different purposes in using an object diagram, such as:

- Forward and reverse engineering of the system;
- Analysis and visualization of objects relationship in a system;
- Static view of interaction;
- To be able to acknowledge object behavior, as well as the relationship among them in a practical sense.

The following figure illustrates the difference between a class diagram and an object diagram. In the right side (the class diagram) an abstract polynomial class is defined. The polynomial class can define polynomials (instances of the class) with different orders, different constant values and so on. On the left side (the object diagram) a specific instance of the polynomial class is defined, which is classed poly12. An instance has defined properties and attributed values to it, although it may change at different simulation moments. Nonetheless, if a snapshot of the simulation is taken, the object at that specific moment has definite values and it highly concrete.

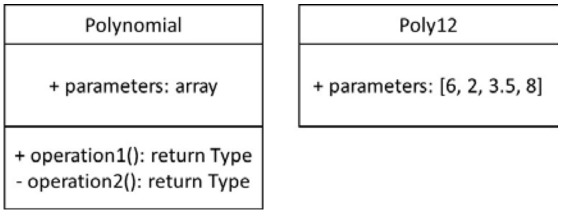

Polynomial
+ parameters: array
+ operation1(): return Type
- operation2(): return Type
Poly12
+ parameters: [6, 2, 3.5, 8]

Figure 17: Class diagram (left) and object diagram (right) in UML.

Use case diagrams

A Use case diagram refers to how the users (also denoted as actors) interact with the system. A set of special symbols and connectors are used to illustrate these relationships. The following purposes should be fulfilled when setting up such diagram:

- The scenarios in which the system interacts with people, organizations, or other systems;
- The objectives or tasks that such actors achieve;
- The scope of the system.

The purpose of an use case diagram is not to depict detailed interaction of the user at each simulation time with the system. Rather, the objective of it is to show a general overview of the relationship among actors, use cases and systems. In simple systems, a use case diagram can be replaced by a use case textual description, if the developer desires to do so.

In the diagram, each use case is symbolized with a labeled oval shape. Lines represent actors in the process, and each actor interaction with the system is represented with a line between the actor and the use case. The system is limited with a box around the use case.

The following figure illustrates a simple use case diagram of a simulator / equation calculator. The user interacts with the system by providing the necessary parameters, solvers, etc in order to configure the model or equation to be simulated.

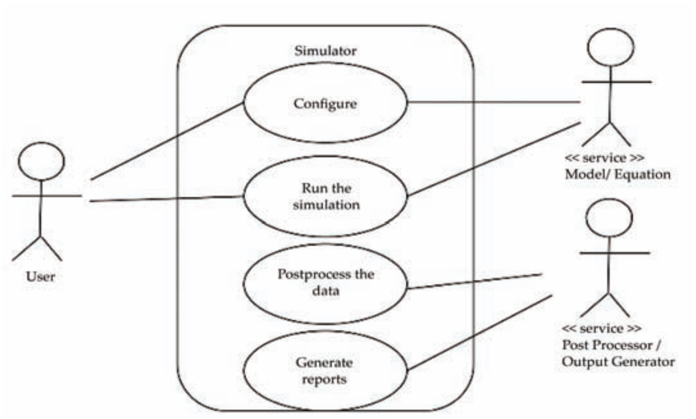

Simulator
Configure
Run the simulation
Postprocess the data
Generate reports
User
<< service >> Model/ Equation
<< service >> Post Processor / Output Generator

Figure 18: Example of a use case diagram (Simulator).

On the other hand, the Model / Equation external service captures the configuration provided by the user as well as the run simulation command in order to generate the results. A post process function is used to prost process the data and generate reports / outputs.

The ideal application of use case diagrams is in cases such as (Lucid Chart, --):

- To represent the goals or objectives of user-system interaction;
- Define and organize functional requirements in a system;
- Specify the context and requirements of a system;
- Modelling the basic flow of events in a use case.

The main components of a use case diagram are:

- Actors: The person, organization or external service that interacts with the system. They must be external and must produce/ consume data. The total set of actors in a use case model reflects everything that needs to exchange information with the system (Rosenberg and Scott, 1999).
- Scenario: Also referred to as system, is the sequence of actions and interactions between actors and the system.
- Goals: the result of most use cases. A good use case diagram is supposed to show the activities and variants to reach the necessary goal.

The relationship between an actor and the use cases is not a single one, rather it may be of different kinds. The default relationship if the «communicates» relationship. This type of relationship is used to show that the actor has issued a request to the use case, or vice-versa. The actor's issues requests usually requiring measurable outputs. A more abstract relationship may exist between different use cases. If a line without label is written in a use case diagram, then a relationship of «communicates» exists between the two entities involved.

Regarding use cases relationships, as mentioned earlier, they may be far more abstract than the relationship between actors and use cases. In UML, there are two possible types of relationships among use cases. These types are:

- <<include>>
- <<extend>>

The <<include>> relationship is used to extend the functionality of a use case, incorporating the behavior of the one in the other extreme of the relationship line. In this it is avoided the repetition of a behavior description through the inheritance of it. As an example, suppose that different functionalities in a simulator application requires the user performs the same action. In this case, these functionalities will all have a <<include>> relationship with this common use case task. Because this type of relationship is not the default one, a label must be provided over the relationship symbol to make it explicit.

The <<extend>> relationship is used when a use case may perform a different behavior or a specific one at a specific lifetime of the simulation, under certain stated conditions. This means that this task will not be performed for sure every time that the use case runs, but only on special occasions. For example, suppose that a system which an user provides a number for a “Root Calculator” use case, and it calculates the squared root of this number. It is supposed that the user will provide positive numbers, so the square root calculation is straight forward. However, what happens if the user provides a negative number? A <<extend>> relationship can be used to, for example, issue an error and to ask for the user to provide positive numbers only, or a different algorithm which is able of calculating complex results for the square root of a negative number.

Sequence diagrams

As the name shows, sequence diagrams describe a chronological sequence of events of the system, and are mainly used in analysis and design phases. When a sequence diagram is created, objects defined in use cases are identified and considered as participants of the sequence. Different pieces of the use case behavior are attributed to objects in the form of services.

Sequence diagrams can be used to refine the use case diagram, for during each implementation the developer can figure out at what specific time of the simulation lifetime each event occur, and if an interaction which was not depicted before can be used to refine the use case diagram. The same can be said for an interaction that it never happens in a sequence diagram, can also be eliminated from an use case diagram.

According Williams (2004), in a sequence diagram, objects are shown in columns, with their object symbol on the top of the line. Similar to the class diagram, the object name appears in a rectangle. If a class name is specified, it appears before the colon. The object name always appears after a colon (even if no class name is specified). If an external actor (see the preceding Use Case Diagram section above) initiates any interaction, the stick figure can be used rather than a rectangle.

The diagram is seen in two dimensions. The vertical dimension refers to the simulation time, or application lifetime. The horizontal dimension represents the objects existent in the system. The sequence of events start at the top-left corner of the diagram, and the time progresses from top to down. The vertical line is referred to as object's lifeline. The ordering of the objects in the horizontal is arbitrary.

The representation of a message sent from one object to another is done with a line from the sender to the receiver labeled at a minimum with a message name. Optionally, the message label can include the information (arguments) that are necessary to the sent to the receiver with the message. When the receiver gets the message, a corresponding operation is executed, and during the realization of such task, other messages may be sent to other objects. An object can also send a message to itself, represented by an arrow from the object line to the same line.

The following figure illustrates a simple sequence diagram of an elevator service. These diagrams are normally concrete and represent one scenario. It may be necessary for an application to have a series of sequences diagrams, each one showing a specific scenario, according the user requirement.

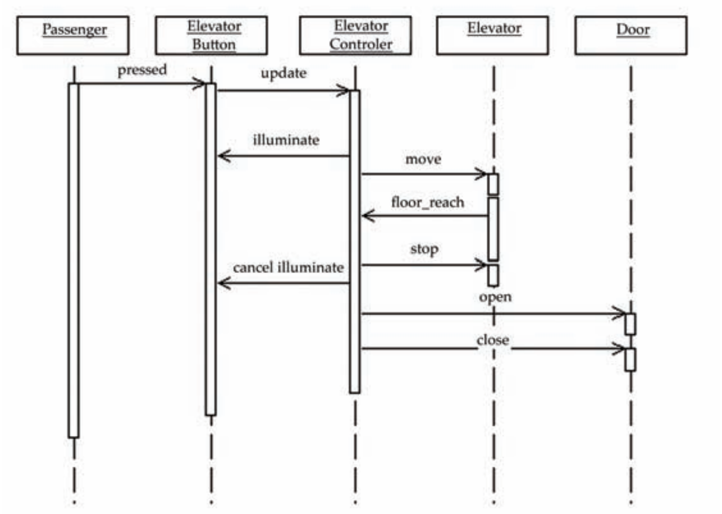


Figure 19: Use case diagram of an elevator service. (Source: http://www.web-feats.com)

Collaboration diagrams

The Collaboration diagram is similar to the Sequence diagram, in the sense that it represents the interactive behavior of objects in UML. The main difference relies in the fact that, while Sequence diagrams illustrates the time flow of messages in a model, the collaboration diagram shows the structural organization of the objects that send or receive messages.

State diagrams

This type of diagrams describe the behavior of nontrivial objects. It is mainly developed during the analysis and design phases. They are especially useful for describing the states of an object across different use cases, and also to identify object attributes as well as to refine the behavior description of an object (Williams, 2004).

To understand what the purpose of such diagram is, it is necessary to define what a state in UML is. A state is a condition that a object can be at some point during its lifetime, within a time frame. The state diagrams should be able to describe each and every condition (state) that an object may go through during its lifetime, as a result of interaction with different elements in the systems that generates a transition in the object's state.

The main purposes of the state diagram are:

- To model the dynamics of a system;
- To model the lifetime of a reactive system;
- To describe the different states that an object goes through during its lifetime.
- To outline a state machine capable of modelling the states of an object.

Different symbols are used to represent each feature of the state diagram. A state is represented by a rounded rectangle, as shown in the figure below.

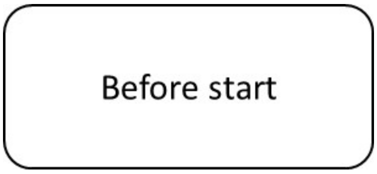


Figure 20: State representation in UML.

The initial condition (start state) is represented by a filled circle, as shown in the figure below.

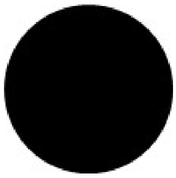


Figure 21: Initial state representation in UML.

The final condition is represented by a filled circle with another circle around it

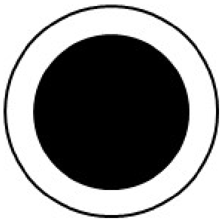


Figure 22: Final state representation in UML.

A transition or a change in the state is represented through arrow lines connecting different states, which is triggered by events, conditions or time.

Activity diagrams

An activity diagrams is mainly developed during the implementation of “complicated” methods that may require many different activities to be performed in specific sequences. The activity diagram can also be incorporated in a system model during the analysis to better understand the flow of a use case, by breaking it down. Through an activity diagram, the developer postulates the sequence of rules that the use case must perform.

Basically, an activity can be described as an action that must be performed, either by a human or a computer. Each activity block must be a single step. For instance, the task “cook rice” is not a single activity, for it involves a series of single actions. These actions are, a simple way, to boil the water, to pour the rice in the water, wait for free water evaporation and finally to remove the rice from the heat. The description of these activities in a flow sheet constitutes an activity diagram. This is an example of a series of activities that must be performed in a row. Alternatively, an activity diagram may show activities that may be performed in parallel to each other, and some symbols are used to guarantee synchronism, i.e, all the activities before have been finished before going to a next step. This can be exemplified in a more refined example of the “cook rice” simulation, for the water can be boiled while some garlic is being fried and rice is added and salted. When the rice is pre-prepared and the water is boiled (synchronism), then we add the water to the rice and next activities are the same as described above.

The main purposes of an activity diagram are:

- Draw the activity flow of a system;
- Describe activities in a sequential way;
- Described parallel, sequential, branched and concurrent flow of the system.

The following elements are involved in an activity diagram:

- Activites: the main core of the diagram.
- Association

- Conditions
- Constraints

The used symbols and representation in an activity diagram are very similar to a state diagram. Two additional symbols are used: the synchronization bar and the decision symbol.

As the name resembles, the synchronization bar indicates that all parallel activities above it must be finished before proceeding to the next activities. This allows concurrent activities to exist and to define the point where the output of them must be ready to serve as a trigger to the next activities.

The decision symbol is a diamond shape, with one or more incoming arrows and one or more outgoing arrows, each one labeled by a distinct condition (Williams, 2004). The condition is a Boolean value (true or false, 0 or 1) which defines all possible outgoings of a decision element.

Component diagrams

The main focus of a component diagram is to reveal the relations existent among different components of the system. The component is defined in UML as a module of classes which represent independent systems or subsystems with the capability of interfacing with the system as a whole. A system mainly based in this type of approach is referred to as component-based development. The component diagrams also allows an overview of the whole project, in such a way that the developer has a general outlook of what the system is supposed to do.

From an object-oriented point of view in scientific computing, the component diagram allows the main developer to arrange classes in groups based on a common purpose. In this way, other developers can also have a general outlook of the project in a higher level.

The main distinction between a component and a class is done in the header of the symbol. A component is also represented with the keyword <<component>> and/or the component symbol. This is very important for, the same symbol without this indication is a class object.

The following figure illustrates a component object in UML. As mentioned before, special attention is given to the header with the keyword and the component logo.

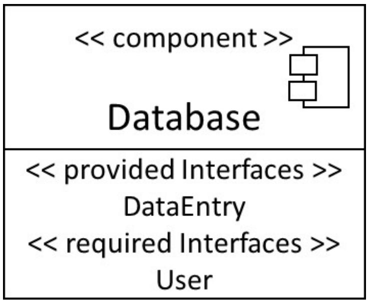


Figure 23: Component example in UML.

The interfaces of a component are represented in a similar way that is done with attributes and methods in a class object. These interfaces represent the location where the group of classes inside the component interacts with other system components. There are two common interfaces: the provided interfaces and the required interfaces.

The provided interfaces are represented as a straight line from the component box with an attached circle. The provided interfaces represent communication between data produced by the current component with an external one.

The required interfaces are represented using a straight line from the component box with an attached half circle. Alternatively, it can be represented as a dashed arrow with an open arrow. It indicates the interfaces used to obtain information for a component in order to perform the function it was designed for.

The component diagrams should generate communication between:

- The scope of the system;
- The overall structure of the application;
- Specific objectives that users or external services may need to achieve.

The following is an example of component diagram in scientific computing. It can be clearly seen that the use of UML notation to develop such system makes the development of Object Oriented programming much easier.

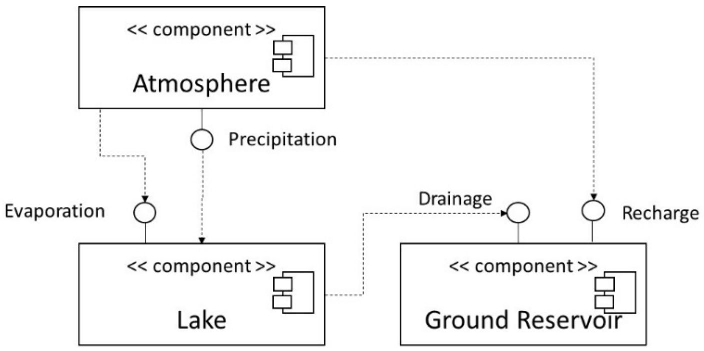


Figure 24: Example of Component diagram in OOP for scientific computing.

Deployment diagrams

In essence, a deployment diagram describes the physical deployment of information that the software generates. Each information generated by the software is called an artifact. The basic building block for this type of diagram is the node, which represents the basic software or hardware elements in the system. Lines between nodes indicate relationships and shapes inside nodes represent software artifacts that are deployed.

The deployment diagrams are applicable in:

- Showing which software elements are deployed by which hardware elements;
- Illustrating the runtime processing for hardware;
- Provide the topology of a hardware system.

The process of developing deployment diagrams involves tasks such as: identifying the scope of the system under development, whether it involves a single machine or a network or computers; identify the hardware limitation of the system and take this into consideration in the development of the system.

Applications of UML in Object-Oriented Scientific Computing

Recently there are different research lines in using UML for scientific computing. In this chapter some applications are reviewed, which also serve as ideas to help the reader in his own application of this tool.

- Back in 2007, Selic addressed the issue of software's which are developed using different programming languages and conventions, which inherits supported tools which are not very user friendly and not well designed as well. The intent was to develop an expressive domain specific language, still keeping the benefits of the existent programming languages and frameworks. The solution addressed by the author is to use UML profile mechanism to define your "expressive" domain specific modeling language. This confronts with the use of DSL in conformation with the UML standards. The new implementation is referred to as DSML (Directory Services Markup Language).

Regarding using cloud resources for scientific computing, Ostermann et al. (2009) revised the challenges and usuability of compute Clouds to extend a Grid workflow middleware and show on a real implementation that this can speed up executions of scientific workflows. This real implementation makes use of UML notation in order to have a standard for the implemented models, in such a way that no specific piece of software needs to be run in a cloud machine, but all the users involved uses the same one, i.e ASKALON which is a workflow library using UML standard notation.

Qin and Fahringer (2012) developed a workflow environment called ASKALON Workflow Hosting Environment (AWHE), a workflow library to develop models using the latest standard UML Activity diagram. The modelling environment incorporates predefined UML modelling elements and user-defined constructs used to generate the AWDL (Abstract Workflow Description Language) representations of scientific workflows and submit them to the ASKALON runtime system for execution.

In the same year (2012), Perez et al. developed pyOpt, an object-oriented framework for formulating and solving nonlinear constrained optimization problems in an efficient, reusable and portable manner. The framework uses object-oriented concepts, such as class inheritance and operator overloading, to maintain a distinct separation between the problem formulation and the optimization approach used to solve the problem. The framework is developed in the Python programming language. Although the software itself does not make use of UML notation, the authors show how UML notation was used throughout the process of software development and how the visualization of it facilitates the analyses and understanding of the whole system.

C++

Before starting to program in C++, it is necessary to have installed in the computer some editor, a compiler, preprocessor and a debugger. Fortunately, it is not necessary to search for each one of these, as there are softwares available, integrating all these tools into a single piece of program. Some examples that the reader can obtain are listed in the following table:

Software	License	Windows	Linux	Mac OS
C++Builder	Proprietary	Yes	No	Yes
Code::Blocks	GPL	Yes	Yes	Yes
Dev-C++	GPL	Yes	No	No
Eclipse CDT	EPL	Yes	Yes	Yes

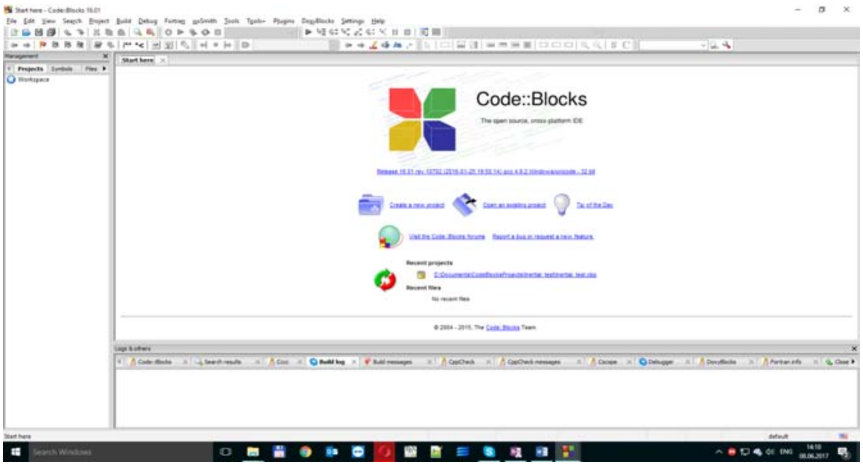
In the present book, we will focus on the development using Code::Blocks, because it is free and one of the easiest tools to develop in C++.

Installing an IDE (Code::Blocks)

Before we start any programming, it is necessary to download and install the IDE, which will help us to go through the tutorial. The Code::Blocks is a free software to develop in C, C++ and FORTRAN, with a clean and easy usable IDE. To obtain it, first navigate to the website: http://www.codeblocks.org

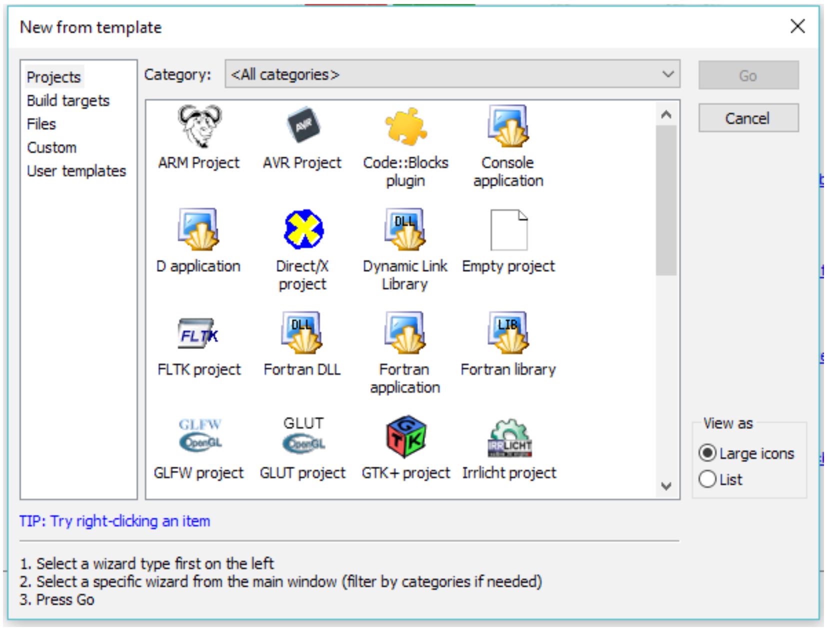
And under the sections "Download", look for the binary release, which is the easiest way to install the software. Once downloaded and installed, we can start with the programming.

One the software starts, the following screen is shown:



A new project can be created, by clicking in the link Create a new project, in the main program area. We will create a simple program, which will show a message in the screen and the user can click any button to close it. Although this piece of software is not clearly useful, it will help the reader to get a first contact with the C++ programming language.

After clicking in the Create a new project button, the program shows the following box:



TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

The project we would like to create is a simple console application. To create it, just click the icon Console application, on the top right corner, and the button Go will unblock. This button leads us to the wizard, or step-by-step helper to begin the project.

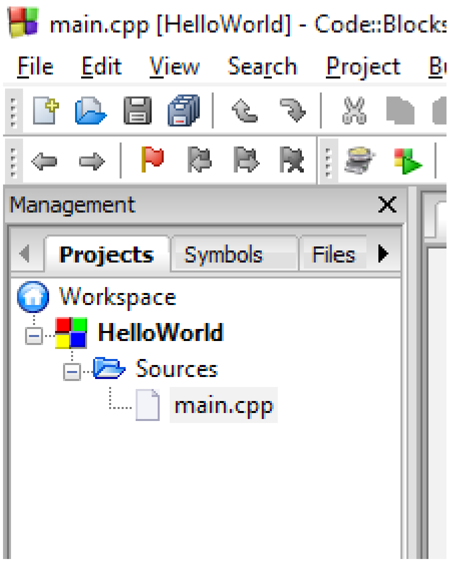
The next window is a welcome window, and a check box can be selected to skip it next time the user starts a project. Clicking on the Next button, the program asks with the developer wants to write a C or a C++ project. For our project, select C++ and click the Next button.

In this moment, it is necessary to define the project title, the folder where the project will be saved. In the Project Title box, type: "*HelloWorld"* (without the quotation marks), and select a folder to save your project. The program automatically fills the box Project filename, with what was written in the project title box, plus a ".cbp" ending. The last box shows the complete filename for your project, including the ".cbp" file.

Clicking Next, a new window is issued, and a compiler must be selected. Make sure to select one compiler already installed in the computer. If unsure, for Windows users, a free Windows C++ 2010 compiler can be downloaded from Windows© website (https://www.microsoft.com). There are two selection boxed below for Debug and Release configuration, selected by default. Just click the Finish button.

Developing a first program

Once a new project was created in the previous sub section, it is time to start the real programming. On the extreme upper left corner, a project tree can be seen. It is necessary to fully expand this tree to see the source code, as showed below:



Now double-click the “main.cpp” file to see its content. The following text is shown:

```
#------------------------------------------------------------------------------------
1 #include <iostream>
2
3  using namespace std;
4
5  int main()
6 {
7     cout << “Hello world!” << endl;
8     return 0;
9  }
#------------------------------------------------------------------------------------
```

We will not go through all the details and lines of the program, but some features must be highlighted:

The first line reads:

```
1 #include <iostream>
```

Is an information necessary for the preprocessor, to know which libraries, or additional code are necessary to be linked to the source code for the program to run. In this case, the library iostream read or write to the standard input/output streams. This library is necessary if we want to have some sort of input from the user or output to the screen, among other uses.

The second line is an empty line. Lines without text are just ignored by the compiler. So why they are used? For a better readability of the developer. In the same way that regular English text is organized in chapters, paragraphs and so on, the programs are organized in blocks and these blocks are better seen if some space is inserted between them.

The third line:

```
3  using namespace std;
```

Is a definition which allows the developer to use functions, classes and constants without always redeclaring the namespace from which it belongs. For instance, the same program above could be written without the statement on the third line in the following form:

```
#--------------------------------------------------------------------------------
----

1 #include <iostream>
2
3  int main()
4  {
5      std::cout << "Hello world!" << std::endl;
6      return 0;
7  }
#--------------------------------------------------------------------------------
----
```

Which issues exactly the same result of the program before, with the difference the wherever it is used the function **cout** or the function **endl**, it will be necessary to use the directive **std::** before. So, the namespace is used to spare typing and time for the developer.

The real program starts in line 5 of the original program:

```
5  int main()
```

The following line has an open bracket, and the line 9 closes it. Everything inside these lines belongs to the program itself. The function name main() is a reserved word in C++ so it knows where the main program is.

Finally, what the program was designed to perform is line 7:

```
7      cout << "Hello world!" << endl;
```

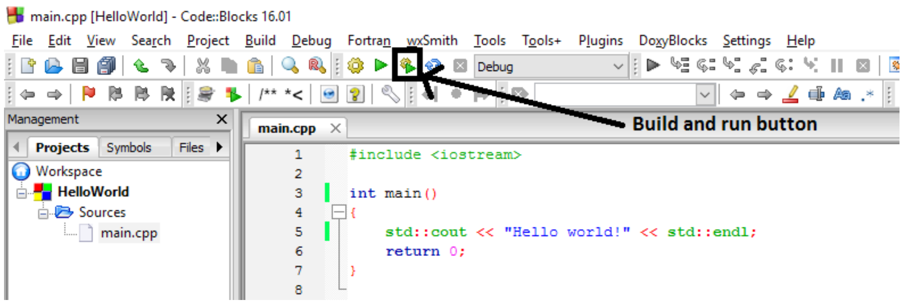
Which is to simply issue a message in the screen, and the message is: “Hello world!”. The function cout is used to print the expression. After << in the screen, and the directive endl ends the line, so what comes next in the program will be printed in the following lines.

The expression in line 8:

8 **return** 0;

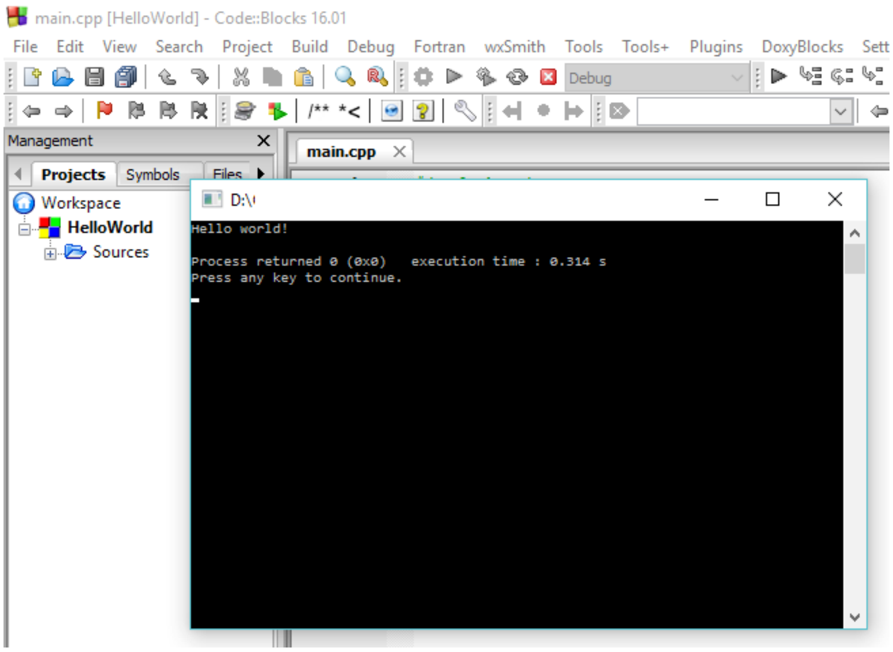
Is necessary to define what value will be returned to the function main(). The beginning of the line 5 says which type of value will be returned, in this case an integer (int), so because we are not interested in the value itself, we return 0 (zero) to the function, and the program terminates.

The program can be run by clicking in different ways in Code::Blocks. One option (and the easiest one) is to click in the Build and run button, which lies in the toolbar on the top of the screen.


main.cpp [HelloWorld] - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
Build and run button
Management
Projects Symbols Files
Workspace
HelloWorld
Sources
main.cpp

```
#include <iostream>

int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

A second option is to click first on the Build Button (the second button to left of the Build and Run Button), which will just compile the code. After click on the Run Button (the first button to the left of the Build and Run Button), which will run the compiled code. The result after following this procedure is shown in the following figure.


main.cpp [HelloWorld] - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Sett
Debug
Management
Projects Symbols Files
Workspace
HelloWorld
Sources
main.cpp
D:\

```
Hello world!

Process returned 0 (0x0)   execution time : 0.314 s
Press any key to continue.
```

Data types

When programming, it is necessary to store different things in memory, such as a number, a name of something, a matrix, etc. This is done by using different data types. Depending on the data type, the system allocate memory and decides what can be stored in reserved memory.

The following table shows the most basic primitive data types for C++ and what it is used for.

Table 1: Primitive data types for C++

Type	Keyword	Description
Boolean	bool	Store TRUE or FALSE values
Character	char	One byte of integer type.
Integer	int	Stores integers such as 0, 1, 2, 3, …
Floating point	float	A single-precision floating point value.
Double floating point	double	A double-precision floating point value.
Valueless	void	Represents the absence of data
Wide character	wchar_t	A wide character type.

The data types can be modified by using one of the following terms:

- signed
- unsigned
- short
- long

The amount of memory and range of data which is used by each type of data depends on these modifiers as well. The following table shows the memory consumption and the range of data for each type of data, unmodified and with the modifiers.

Type	Typical Bit Width	Typical Range
char	1 byte	- 128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	4 bytes	-2147483648 to 2147483647
unsigned int	4 bytes	0 to 4294967295
signed int	4 bytes	-2147483648 to 2147483647
short int	2 bytes	-32768 to 32767
unsigned short int	2 bytes	0 to 65,535
signed short int	2 bytes	-32768 to 32767
long int	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
signed long int	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long int	8 bytes	0 to 18,446,744,073,709,551,615
float	4 bytes	+/- 3.4e +/- 38 (~7 digits)
double	8 bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8 bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

Declaring variables

In C++, a variable must always be defined before being actually used. This definition will tell the machine that some amount of memory needs to be reserved to the variable. The declaration is a line containing the type of the data as well as its name, and more than one variable can be declared in the same line. For example:

```
int a, b;
```

Tells the compiler that memory needs to be reserved for two variables (variable “a” and “b”), and they are of the type integer.

After being declared, an optional step is to initialize the variable, that is to give it an initial value. This is done by writing the variable name, a signal sign and the attributed value, as in:

```
a = 1;
```

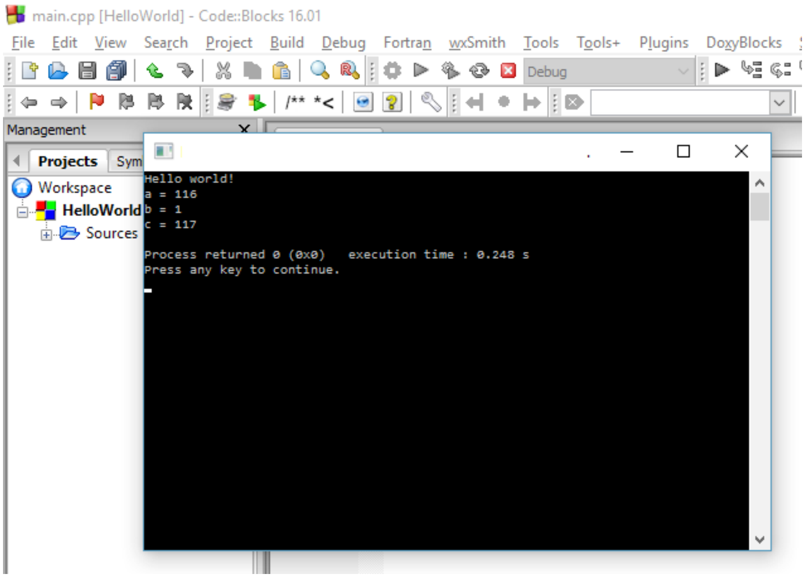
Means that the variable “a” was attributed a value of 1 (one), which is an integer. However, what can happen if the variable is initialized with the wrong type of value? For example, in the case above the variable “a” is supposed to just have integer values. If instead of writing “1”, we write:

```
a = 1.1;
```

This can be tested in the first program we wrote. After some modification, we arrive in a new program as shown below:

```
#include <iostream>

using namespace std;

int main()
{
   int a,b,c;
   a = ‘t’;
   b = 1.1;
   c = a + b;
   cout << “Hello world!” << endl;
   cout << “a” << a << endl;
   cout << “b” << b << endl;
   cout << “c” << c << endl;
   return 0;
}
```

In the example above, the variables “a”, “b” and “c” are declared as integers. However, “a” is initialized as a char (‘t’), b is initialized as a float (1.1) and c is the sum of them. What happens in such case. If we Build and Run the example, the result is the following:


main.cpp [HelloWorld] - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks
Debug
Management
Projects Sym
Workspace
HelloWorld
Sources

```
Hello world!
a = 116
b = 1
c = 117

Process returned 0 (0x0)   execution time : 0.248 s
Press any key to continue.
```

The value of a was modified to 116, b was changed to 1 and c was, as expected the sum of 1 with 116, which is 117. Why was attributed the value of 116 to "t" ? As already mentioned, the char type is actually an integer value hidden behind the character. These values come from the ASCII (American Standard Code for Information Interchange). As computers can only understand numbers, each character is converted to a number according the ASCII. In the present case, the value of "t" in ASCII is 116.

So, a wrong declaration of variables, as well as a wrong initialization can lead to errors that may be difficult to track. Another situation that can easily lead to errors as careful attention must be taken is the scope of the variables. In general there are three places that a variable can be declared:

- Inside a function or a block, being local variables,
- In the definition of function parameters, being formal parameters.
- Outside of all functions, being global variables.

The examples shown before use local variables declaration. This is because the declaration of the variables occurred inside the function. An example of formal parameter declaration is:

```
#include <iostream>

using namespace std;

int sum(int a, int b)
{
return a + b;
}

int main()
{
    int a,b,c;
    a = 161;
    b = 1;
    c = sum(a,b);
    cout << “Hello world!” << endl;
    cout << “a = “ << a << endl;
    cout << “b = “ << b << endl;
    cout << “c = “ << c << endl;
    return 0;
}
```

We declared a function, sum( ), which takes to 2 arguments, or formal parameters, integer a and integer b. In this case is clearly seen that the declaration of the variables followed the declaration of the function, with a defined data type. The output of this program is the same as the one before, with the difference that the variables are correctly initialized.

The following is an example of global variable declaration:

```
#include <iostream>

using namespace std;

```

```
int c;

int main()
{
   int a,b;
    a = 161;
    b = 1;
    c = a + b;
    cout << "Hello world!" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    return 0;
}
```

In this case, the variable integer “c” is declared outside of a function, and any following function that uses it can do so without declaring again. That’s the difference between the local and the global variable.

Operators

Operators are used in C++ in the almost the same way they are used in mathematics, either to perform some algebraic operation, to compare values, to assign a value and so on.

The following is a list of common operators for C++.

Operator	Description	Example
+	Adds two variables	10 + 40 will give 50
-	Subtracts one variable from the other	10 - 40 will give -30
*	Multiplies variables	10 * 40 will give 400
/	Divides numerator by denominator	10 / 40 will give 0.25
%	Modulus Operator and remainder of after an integer division	10 % 40 will give 10
++	Increment operator, increases integer value by one	A = 10 A++ will give 11

--	Decrement operator, decreases integer value by one	A = 10 A-- will give 9

Relational Operators

Relational operator, as the name suggests, compare two variables and returns a Boolean value, either true or false. For instance, if a = 10 and b = 10.1 and we want to know if "a" is equal to b, the result if false. The following is a list of relational operators in C++. Assume that A = 10 and B = 10.1

Operator	Description	Example
==	Checks if the values of two variables are equal. In positive case then returns true, otherwise false.	(A == B) is not true.
!=	Checks if the values of two variables are not equal. In positive case (if the variables are not equal) then returns true, otherwise (if the variables are equal) than false.	(A != B) is true.
>	Checks if the value to the left side of the operand is greater than the value to the right of the operand. In positive case then returns true.	(A > B) is not true.
<	Checks if the value to the left side of the operand is smaller than the value to the right of the operand. In positive case then returns true.	(A < B) is true.
>=	Checks if the value to the left side of the operand is greater than or equal to the value to the right of the operand. In positive case then returns true.	(A >= B) is not true.
<=	Checks if the value to the left side of the operand is smaller than or equal to the value to the right of the operand. In positive case then returns true.	(A <= B) is true.

Logical Operators

There is a special type of relational opeators, used to compare Booleans (true of false values) instead of any value as in the case above. The list below shows some operators used in C++. Assume that A is true (1) and B is false (0).

Operator	Description	Example
&&	AND operator. If both the operands are true (non--zero), then returns true.	(A && B) is false.

||	OR Operator. If any of the two operands is true (non--zero), then returns true.	(A || B) is true.
!	NOT Operator. If a condition is true, returns false.	!(A && B) is true.

Assignment Operators

These operators are used to attribute a value or string to a variable, and to modify it, summing, subtracting, multiplying and so on. The following list shows some assignment operators.

Opera-tor	Description	Example
=	Values from the right side are assigned to the left side.	C = A + B will assign value of A + B toC
+=	Used to add the right-side value to the left side AND attribute again to the left side.	C += A is the same as C = C + A
-=	Used to subtract the right-side value from the left side AND attribute again to the left side	C -= A is the same asC = C - A
*=	Used to multiply the right-side value with the left side AND attribute again to the left side	C *= A is the same as C = C * A
/=	Used to divide the left side value by the ri-ght side AND attribute again to the left side	C /= A is the same as C = C / A
%=	Used to get the modulus between the left side value and the right side AND attribute again to the left side	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Flow Control

Flow control structure are used to run a specific section of code, only when some condition(s) is met. This structure can include only a single condition, as well as a list of options if different conditions have to be considered, as well as a section if any of the conditions are met. The next flow diagram illustrates this structure:

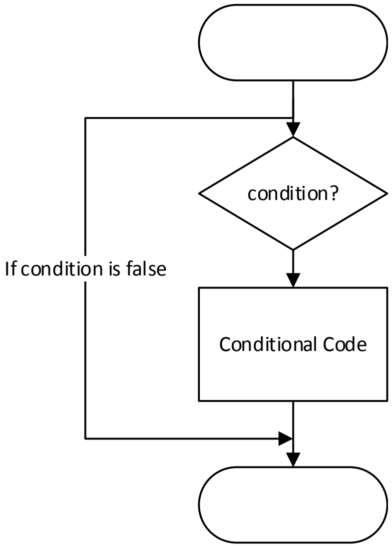


Figure 25: Flow diagram.

The most simple flow control structure is the if statement. It is used to test one condition, and if it is true, then run some code. In case it is false, it just jumps over the code and continue the code from below the if statement block. The following is an example of this structure:

```
int a,b;
a = 10;
b = 40;

if (a > b)
{
        cout << "a > b" << endl;
}
```

In the case above, the value of a is compared with the value of b using the relational operator Greater than. If it returns true, then the statement inside the if block (the single line with cout command) runs.

Otherwise, it just jumps this block and continues with the code outside it.

The if block can also be written in a more compact way as shown below.

```
if (a > b) cout << “a > b” << endl;
```

This type of declaration is useful for short structures, but it can difficult the reading for more complex programs, in which case the block style is preferable over the single line style.

It is possible to add a condition for the case that the relational operator returns false. To do that it is added a block else which just runs if the condition tested in the if block is false. The following is one example of such application:

```
a = 10;
b = 40;

if (a > b)
{
        cout << “a > b” << endl;
}
else // if a < = b
{
        cout << “a <= b” << endl;
}
```

In this case, the program will print the statement inside the else block (“a <=b”), since the condition tested returns false. Notice the comment used after the else keyword. Comments in such type of structures are always good programming practice, for they do not change the programming flow but provides the developer with some insight of what is being done in that part of the program.

It may be necessary to test not only one condition, but a set of conditions, and if that condition is met, then run the code inside the block. This is done by coupling else if blocks under the if block. An else block can also be used in such cases, in order to run some code if any of the tested conditions are met. The following code shows how to use this structure.

```
        if (statement 1)
        {
                Code if statement 1 = TRUE
        }
        else if (statement 2)
        {
                Code if statement 2 = TRUE
        }
        else
        {
                Code if statement 1 = FALSE and statement 2 = FALSE
        }
```

Example

Program a simple calculator to perform the four basic algebraic operations (sum, subtraction, multiplication and division). To achieve that, show a simple menu at the beginning of the program, from where the user can choose one of the operations.

Solution

The following code lets the user insert the two numbers to do the calculations, and to choose the mathematical operation in a menu, using the if-else condition block.

```
#include <iostream>

using namespace std;
int c;
int main()
{
   double a,b;
   a = 0.0;
```

```
b = 0.0;

int c;
c = 0;

cout << “My Calculator 0.0 !!” << endl;
cout << “====================” << endl;
cout << “Type the first number:” << endl;
cin >> a >> endl;

cout << “Type the second number:” << endl;
cin >> b >> endl;

cout << “Choose the operation” << endl;
cout << “0 - Sum” << endl;
cout << “1 - Subtract” << endl;
cout << “2 - Multiply” << endl;
cout << “3 - Divide” << endl;

cin >> c >> endl;

if ( c == 0 )
{
        cout << a << “ + “ << b <<” = “ << a + b << endl;
}
else if ( c == 1 )
{
        cout << a << “ - “ << b <<” = “ << a - b << endl;
}
else if ( c == 2 )
```

```
        {
                cout << a << " * " << b <<" = " << a * b << endl;
        }
        else if ( c == 3 )
        {
                cout << a << " / " << b <<" = " << a / b << endl;
        }
        else
        {
                cout << "No valid option selected" << endl;
        }

   return 0;
}
```

In the way presented above, the user can perform one mathematical operation as the example required, but after that the program ends. If the user wants to continue performing calculations, he should close the application and start it again. That is not very practical. To avoid this, we can use loops, which tells the program to return to some point of the code wherever it reaches the bottom of the block. One example of this structure is the while loop block. An example of this structure is shown below.

```
while ( condition is true )
{
        code to run
}
```

This type of block can also generate infinite loops, i.e the code runs eternally, if it never reaches a condition to leave the block. For example, the following code:

```
a = 1;
while ( a == 1 )
```

```
{
        cout << “Print line” << endl;
}
```

will generate an infinite loop, for the condition (a = 1) is always true and do not change inside the loop. Infinite loops can be dangerous inside a program and may cause crashes on it, so it is advisable to always check if loops are never infinite.

Basic Input and Output

An input operation is the flow of information (bytes) from the user to the main memory (program), using either a keyboard, a disk drive, a network connection, etc.

An output goes the other way around. It is a block of information generated by the program and displayed in a screen, recoded in memory, printed, etc.

C++ has a set of libraries or header files that deals with input/ output streams. The following table summarizes them.

Header File	Function and Description
<iostream>	This file defines the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, which are the objects cin, cout, cerr and clog respectively.
<iomanip>	This file declares services to use parameterized stream manipulators with formatted I/O, such as setw and setprecision.
<fstream>	This file declares services for user-controlled file processing.

Functions

A function in mathematical terms is one or more equations or algorithms that generates a result given some input. For instance, the function $f(x) = x^2$ is a function that, given an arbitrary value x, generates the square of it and give it as an output. So $f(1) = 1$ and $f(2) = 4$. In a similar way, a function in C++ is a series of procedures that performs some task given some input.

The program itself is a function called *main*. Besides that, smaller pieces of code can be collected together to summarize a function. This is specially useful in code that is repeated throughout the program. For example, suppose we need to calculate the sum of a number, than divide it by something and square it. To repeat the code for this procedure one or two times is not a big problem, but it can become inconvenient if it has to be done 100 times. So a function can be used to perform this operations, and the necessary inputs are given to it.

The following code exemplifies what is above mentioned:

```
#include <iostream>
using namespace std;
double function1 (double a, double b)
{
   double c;
   double d;
   c = a + b;
   d = c/10;
   return d*2.0;
}
int main()
{
   double a,b,c;
   a = 0.0;
   b = 0.0;
   c = 0.0;

   cout << “My Calculator 0.0 !!” << endl;
   cout << “====================” << endl;

   for(int count=0;count<=10;count++)
   {
      c = function1(a,b);
      cout << “a = “ << a << endl;
      cout << “b = “ <<  b << endl;
```

```
        cout << "c = " <<  c << endl;
        cout << "count = " <<  count << endl;
        a+=1;
        b+=1;
    }
}
```

The *function 1* is used to repeat a series of mathematical operations performed inside the loop *for*. This same operations can be performed anywhere inside the program without rewriting all the operations, but just by calling the *function1* which returns a number.

The expression

```
c = function1(a,b);
```

Is referred to as a function call. The information sent inside the parenthesis are the inputs for the function. The left-hand side of the "equation" above is the variable that will receive the value generated by the function.

To perform common math operations, C++ already contains a library, cmath, which provides the functionality similar to a scientific calculator. The following table enlists some of the functions available.

Function	Description
double sqrt(double x)	Computes the square root of a number: $\mathrm{sqrt}(x)=\sqrt{x}$
double exp(double x)	Computes the exponential of a number $\exp(x)=e^x$
double log(double x)	Computes the natural logarithm of a number $\log(x)=\ln x$
double log10(double x)	Computes the logarithm in a base of 10 $\log 10(x)=\log x$
double cos(double)	Computes the cosine of a number $\cos(x)=\cos x$
double sin(double)	Computes the sine of a number $\sin(x)=\cos x$

double pow (double x, double y)	Raises the first number to the power of the second $\text{pow}(x) = x^y$
double fabs(double x)	Computes the absolute value of a number $\text{fabs}(x) = \lvert x \rvert$

To use this library, simply type in the header the prerogative:
#include <cmath>

C++ Modules

Modules can be defined as "boxes" of code, which can be easily shared by different users without the necessity of understanding the whole algorithm implemented on it. The only thing required is an understanding of the interface of for what purpose the code was developed. In this sense, a module can be seen as a "black box".

A module may be a function or a set of functions defined to perform a task. If the module is composed by a collection of functions, one single function is used as in an interface, to collect all the inputs, make the calculations inside it using auxiliary functions and generate the necessary output. Suppose a module used to solve linear systems of equations of the following form:

$Ax = b$

Where A is a square, invertible matrix of dimensions is n x n. x is the solution vector of size n and b is a column vector of size n. A module may be implemented, taking these values as parameters and generating the necessary output using a simple prototype as follows:

SolveLinearProblem (double** A, double* x, double* b, int n);

The function SolveLinearProblem receives these arguments and implements all the necessary functions and methods to solve the problem and generate as output the solution vector x. This main function, as well as all those secondary methods called by it are reffered to as a module.

C++ Classes and Objects

Classes in C++ provide a series of advantages over the previously described modules, such as:

- Contains all the necessary functions to solve the desired problem;

- The functions inside a class can not be accessed by other parts of a program, except through defined interfaces;
- Can not itself access other parts of the program;
- Contains all the necessary data to solve the problem.

The data associated with a class is referred to as class members, and the functions as methods.

In the following example, it is defined a class of Process Data. Some basic attributes that each data may have are:

- A value (1, 2.8, 1e-2, etc);
- An unit (m^2, m/s, kg, etc)
- A date (for example a single number, 20081201 referring to the 1st of December of 2008).
- An identification code.

These attributes are connected to each instance of a process data by generating the following ProcessData.hpp. Each of the attributes mentioned above are class members.

```
#include <string>
class ProcessData
{
public:
std::string unit;
double value;
int identificationnumber, date;
};
```

The extension .hpp is used to indicate that the file is a header file associated with a C++ program. The keyword public allows external instances to access all variables of the class.

An usage of the class defined above is shows below. As a coding convention, The header file containing the class definition is enclosed within quotation marks, in contrast with system header files such as iostream, fstream and cmath with should ne enclosed with angle brackets.

This makes easier for the developer to distinguish between local include files and external ones.

```
#include <iostream>
#include “ProcessData.hpp”
int main (int argc, char* argv [])
{
          ProcessData distilled_flow;
          distilled_flow.identificationnumber = “F01”;
          distilled_flow.value = 10.00;
          distilled_flow.unit = “kg/s”;
          distilled_flow.date = 20170601;
          std::cout<<“Distilleddata”<<distilled_flow.identificationnumber
<< “obtained on ”
                    << distilled_flow.date << “ is”
                    << distilled_flow.value << distilled_flow.value << “\n”;
}
```

Inheritance

Inheritance is defined as the ability of extending classes by implementing them in a “family tree”. The data and methods of a superclass can be implemented and extended in a subclass. And not only one, but several subclasses can be derived from one superclass. The concept of inheritance arises two complementary concepts: extensibility and polymorphism.

Extensibility is the idea that a code can be easily extended, not by changing the original code, but only by adding functionality to it. Polymorphism the ability of implementing the same functionality in a variety of different types of objects.

For instance, the class developed above to hold Process Data can be extended to hold a specific type of process data, steam process data. Steam has specific data such as pressure and boiling point. So basically two additional class members are added:

- Absolute pressure
- Boiling point

The boiling point can be obtained from the pressure.

As the Steam Process Data is derived from the generic Process Data, this class definition in the .hpp file must be added in the header files.

```
    #ifndef STEAMPROCESSDATAHEADERDEF
    #define STEAMPROCESSDATAHEADERDEF
    #include "ProcessData.hpp"
    class SteamProcessData: public ProcessData
{
public:
SteamProcessData();
double AbsolutePressure, BoilingPoint;
};
#endif
```

All the public and protected members of the class ProcessData are available for the class SteamProcessData.

Polymorphism

Polymorphism is a highly useful feature when a variety of different classes are derived from one class, and for some of these classes it may be necessary to adapt one or more methods of the superclass. This redefinition can be done in C++ using the keyword virtual which defines methods that perform different tasks for different derived classes. The virtual keyword is a signal to the compiler that a method has the potential to be overridden by a derived class.

An example can be shown for a class developed representing a machine used to process some specific material. The machine refines the material by separating it from contaminants, obtaining in average 60% of the total mass of the raw material as the desired one, and the other 40% are contaminants and some material that could not be efficiently processed. So, a class defining the machine is shown below.

```
#ifndef MACHINEDEF
#define MACHINEDEF
#include <string>
```

```
class Machine
{
public:
std::string ID;
double RawMass;
virtual double RefineRawMaterial();
};
#endif
```

The virtual method RefineRawMaterial() defines the equation used to calculate the amount of refined material obtained from the RawMass amount of raw material. The RefineRawMaterial is implemented as a virtual method, showing that this method has a potential to be overridden by derived classes. The implementation of the method RefineRawMateiral is given below, where the amount of refined material is calculated as 60% of the total raw mass (40% is contaminant).

```
#include “Machine.hpp”
double Machine::RefineRawMaterial()
{
return RawMass * 0.6;
}
```

However, a special machine is able of separating more efficiently the same raw material, obtaining only in average 35% of contaminants in relation to the total mass of raw material, and the other 65% as the desired material. A subclass SpecialMachine can derived from the Machine class above.

```
#ifndef SPECIALMACHINEDEF
#define SPECIALMACHINEDEF
#include “Machine.hpp”
class SpecialMachine: public Machine
{
public:
double RefineRawMaterial()
}
```

Basically, the Special Machine implements the same class members and methods of the common machine. However, the method RefineRawMaterial is redefined as to calculated properly the more efficient separation of material as follows:

```
#include “SpecialMachine.hpp”
double SpecialMachine::RefineRawMaterial()
{
Return RawMass * 0.65;


}
```

Applications in Scientific Computing

In the following paragraphs, some literature review is given on application of C++ language in scientific computing using object-oriented approach. Specifically, we give a short review of the developed work and direct the reader to the article so a deeper understanding of a work can be retrieved.

Regarding programming language derived from C++, Kale & Krishnan (1993) developed Charm++, an explicit parallel object-oriented programming language, also extensible. The special features of the developed language involve multiple inheritance, dynamic binding, overloading, strong typing, and reuse for parallel objects. Charm++ provides. Specific modes for sharing information between parallel objects and extensive dynamic load balancing strategies are provided.

In 1993, Dubois Pe`lerin & Zimmermann developed an efficient object-oriented finite element programming in C++. The developed algorithm is extensive and because of that it is divided in different companions. The first one describes the governing principles of object-oriented finite element programming. The second companion described a prototype implementation written in Smalltalk, which proved that object-oriented programming is adequate for the design of easily maintainable software. In the third and last article, numerical efficiency is analyzed. The authors showed that achieved performance is comparable with Fortran.

Regarding applications in Bioinformatics, Thorthon (2003) developed a C++ library for evolutionary genetic analysis called libsequence. The library implements methods for data manipulation and the calculation of several statistics commonly used to analyze SNP data. The object-oriented design of the library is intended to be extensible, allowing users to design custom classes for their own needs. In addition, routines are provided to process samples generated by a widely used coalescent simulation.

Jasak et al. (2004) developed a C++ object-oriented toolkit called FOAM, a software designed to facilitate research in physical modelling, by mimicking in the code the continuum mechanics equations of the physical model. With this feature, it can handle separately physics from numerical discretization techniques. The authors explored the limitations of the toolkit applying it in the investigation of two in-cylinder combustion simulations.

Vukics & Ritsch (2007) created a framework for efficiently performing Monte Carlo wave-function simulations in cavity QED with moving particles. The developed framework uses object-oriented approach in in C++, with features such as extensibility and applicability for simulating open interacting quantum dynamics in general. The user is provided with several "elements", eg pumped moving particles, pumped lossy cavity modes, and various interactions to compose complex interacting systems, which contain several particles moving in electromagnetic fields of various configurations, and perform wave-function simulations on such systems. A great number of tools are provided to facilitate the implementation of new elements.

Heat transfer using object-oriented approach in C++ was investigated by Mangani et al. (2007) in the article "Development and validation of a c++ object oriented cfd code for heat transfer analysis". The code is based on the Field Operation and Manipulation C++ class library for continuum mechanics (OpenFOAM). The accuracy of the implementations was validate comparing results with experimental data available both from standard literature test cases and from in house performed experiments.

Ferrari et al (2016) developed an object-oriented library called LibHalfSpace C++ to evaluate the deformation and stress in elastic half-spaces. The study of such elastic half-spaces is employed in different areas, such as didactic, inversion of geodetic data. A collection of well-known models (Mogie source, penny shaped horizontal crack) are implemented in order to define the potential usage and the flexibility of the library.

MATLAB

MATLAB is a mixture of software package and programming language that allows the user to do mathematics and computation, to analyze data, develop algorithms, model real and simulate different types of models, producing graphical visualization and graphical user interface.

This chapter gives a short introduction to Matlab, for to who are not very familiar with this programming language and tool. Following the introduction, it is presented how object-oriented concepts are expressed in Matlab, as well how to use object-oriented concepts to solve simple problems.

It is assumed that the user has either the student edition of Matlab R2014b or newer, or a professional version. Before reading this chapter, the reader should set up the MATLAB software on the computer and start the application.

To run MATLAB in a PC, simply click in the MATLAB icon, either on the desktop or in the start menu. On a UNIX system, simply type *matlab* at the prompt.

When MATLAB starts, it shows two greather-than signs (>>) when it is ready to accept any command from the user. The program can be ended by typing quit or exit at the MATLAB prompt.

To easily reach help information, simply type one of the following commands:

help	Help on the meaning of a command. It provides a precise explanation of commands.
helpwin	Opens a MATLAB help GUI
doc	the same as helpwin (helpwin will be removed in future versions)
helpdesk	Opens a hypertext help browser
demo	Opens a GUI with a list of demos

Below are some useful commands to have at hand:

Version: To know which version of MATLAB are you running, type:

```
>> version
```

What: This command lists the .m, .mat and .mex files in the current working directory. The command can also be used to list the files in another directory, if the path of this directory is provided as second argument. For example, the command

```
>> what general
```

Provides a list of files in the directory general inside MATLAB path.

Who: provides a list of variables in the current workspace. *Whos* lists additional information about the variables. *Who global* and *whos global* list the variables in the global workspace.

For example, the following list of commands creates a variable with value one (1) and use the command who to see information about it:

```
>> a = 1;
>> who a
    Your variables are:
a
>> whos a
  Name      Size            Bytes  Class     Attributes
  a         1x1                 8  double
```

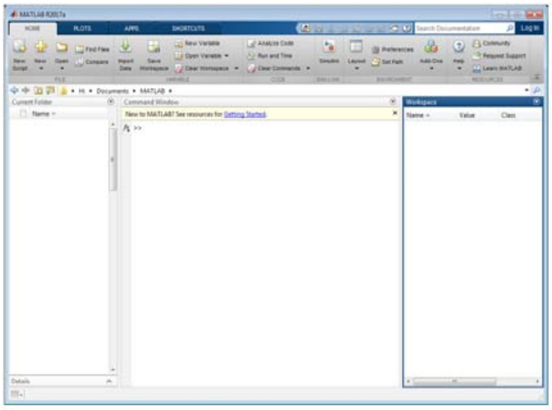
Clock: This command is used to print (or store) the current system time. The first number provided is a multiplier. The following numbers are [year, month, day, hour, minute, second]. For example:

```
>> clock
ans =
   1.0e+03 *
    2.0170    0.0070    0.0110    0.0120    0.0460    0.0316
```

Matlab Primer

Basic commands and arithmetic

Once the user have started MATLAB, the software opens with its default layout, as shown below:



New to MATLAB? See resources for Getting Started.

```
>>
```

Figure 26: MATLAB default window.

The desktop is divided into three main panels:

Current Folder – shows the files existent on the current folder.

Command window – the place where the user can do calculations and issue commands at the prompt (>>).

Workspace – lists the variables created or imported from files.

A variable can be created by typing a name to it, the equal sign and a value. For instance:

```
>> a = 1
```

Creates a variable a and attributes the value 1 to it. Mathematical operations can be directly performed using number or variables, or a mix of them. If the user wants to perform the sum 1+1, it can be performed in different ways:

```
>> a + a
ans =
    2
>> a + 1
ans =
    2
>> 1 + 1
ans =
    2
```

In MATLAB, the result of a command in the prompt is stored in the variable *ans*, so it changes wherever a new command is issued. In the above case, as the value of a is 1, the sum can be performed either by summing a with a, by summing a with 1 or the natural way of 1 + 1.

The following table shows a list of valid mathematical operators in MATLAB: The symbol in parenthesis can be used in place of the command. For example, the command 1 + 1 is equivalent to plus(1,1).

plus (+)	Addition
uplus	Unary plus
minus (-)	Subtraction
uminus	Unary minus
times (.*)	Element-wise multiplication
rdivide (./)	Right array division
ldivide (.\)	Left array division
power (.^)	Element-wise power
mtimes (*)	Matrix Multiplication
mrdivide (/)	Solve systems of linear equations xA = B for x
mldivide (\)	Solve systems of linear equations Ax = B for x
mpower (^)	Matrix power

Different operators can be used in the same command to perform a complex arithmetic operator, as in the following command:

```
>> 1+2.*4-3.^2
ans =
    0
```

The above calculation also shows the MATLAB respects the arithmetic order of operations (first power, than multiplication or division and lastly addition or subtraction). The number can also be grouped using parenthesis, so as to specify the order of the calculation, as in the following example:

```
>> (1+2).*4-3.^2
ans =
    3
```

Matrices and Arrays

MATLAB was primarily developed to work on whole matrices and arrays, instead of a number at a time. This means that all MATLAB variables are multidimensional arrays, no matter what type of data. A matrix is a two-dimensional array often used for linear algebra.An array can be created by using square brackets and separating each number with a comma or space. This type of array is a row vector.

```
>> a = [0, 1, 2, 3]
a =
    0    1    2    3
```

Another way of creating the same array vector is to use the linspace command or the colon operator. The linspace command and the colon operators are used to create equally spaced elements in the row vector. In this sense, the above array can be recreated using the following commands:

```
>> a = linspace(0,3,4)
a =
    0    1    2    3
>> a = 0:1:3
a =
    0    1    2    3
```

The command linspace(x,y,z) creates a row vector starting at x, ending at y with z equally spaced elements. The command x:dx:y creates a row vector starting at x, ending at y with dx step between each value.

To create a matrix with multiple rows, separate the rows using semicolons.

```
>> a = [0, 1, 2, 3; 5, 6, 7, 8; 9, 10, 11, 12]
a =
    0    1    2    3
    5    6    7    8
    9   10   11   12
```

Special commands can be used to directly create special arrays or matrices. The command *eye(x,y)* creates an identity matrix with x rows and y columns.

```
>> eye(2,3)
ans =
   1   0   0
   0   1   0
```

The command *ones(x,y)* or *zeros(x,y)* can be used matrixes of ones or zeros, respectively with dimension of x rows and y columns.

```
>> ones(2,3)
ans =
   1   1   1
   1   1   1
```

Operations can be directly performed in a matrix, using the arithmetic operators presented above. For example, to sum a value of 10 to each value in a matrix of zeros:

```
>> b = zeros (2,3)
b =
   0   0   0
   0   0   0
>> c = b + 10
c =
  10  10  10
  10  10  10
```

It is also possible to perform summation of two matrixes of the same dimension, in which case each element of the matrix will be summed accordingly:

```
>> b = eye(2,3)
b =
   1   0   0
   0   1   0
>> c = [1 2 3; 4 5 6]
```

```
c =
    1    2    3
    4    5    6
>> d = b + c
d =
    2    2    3
    4    6    6
```

Suppose that the desired matrix has 6 rows as a repetition of the first two rows. The user would have to type each number in a very inefficient way. For this case, the command repmat can be used to replicate the matrix. The command repmat(x,y,z) replicates the matrix x, y times in the row direction and z times in the column direction.

```
>> a = [0:1:3;1:1:4]
a =
    0    1    2    3
    1    2    3    4
>> b = repmat(a,3,1)
b =
    0    1    2    3
    1    2    3    4
    0    1    2    3
    1    2    3    4
    0    1    2    3
    1    2    3    4
```

Individual elements in a matrix can be referred to using subscripting. Each element is denoted by a row index and column index respectively. Suppose the user wants to obtain the element in the 3rd row and the 4th column:

```
>> b(3,4)
ans =
    3
```

Indexes can also be used to obtain multiple elements. For example, to extract the elements in the 3rd row, and columns 3 and 4:

```
>> b(3,[3 4])
ans =
    2    3
```

The operator semicolon (:) can be used to obtain all the elements of the row / column.

```
>> b(:,[3 4])
ans =
    2    3
    3    4
    2    3
    3    4
    2    3
    3    4
```

In the example above, the semicolon was used as row index, to obtain the elements of the matrix b in all the rows, in the columns 3 and 4.

Alternatively, a single index can be used to obtain an element of matrix. MATLAB counts the index down successive columns.

Typing in MATLAB

The following table shows a list of commands that can be used to save time when typing. If the user wants to repeat a long line typed before, he can use the arrow keys to repeat the same command:

↑	ctrl - p	Recall previous line
↓	ctrl - n	Recall next line
←	ctrl - b	Move back one character
→	ctrl – f	Move forward one character
ctrl - →	ctrl – r	Move right one word
ctrl - ←	ctrl – l	Move left one word
home	ctrl – a	Move to the beginning of the line
end	ctrl – e	Move to end of the line

esc	ctrl – u	Clear line
del	ctrl – d	Delete character at cursor
backspace	ctrl – h	Delete character before cursor
	ctrl – k	Delete to end of line

Saving and Loading of data

The command save can be used to save all the variables present in the workspace. The following code shows that MATLAB creates a file called matlab.mat, and inside the file it can be found the variables present in the workspace, in this case the variables a and b.

```
>> a = 1
a =
    1
>> b = 2
b =
   2
>> save
     Saving to: …..\matlab.mat
```

Alternatively, an additional argument can be used with the command save to specify the name of the file.

```
>> save filename
```

If more arguments are provided, they specify which variables are to be saved. In the following example, just the variables a and b are saved in the file my_variables:

```
>> save my_variables a b
```

The variables are saved with double precision binary. To load the variables from the file, simply type:

```
>> load my_variables
```

If it is necessary to save the variables in asci format, the user must add the argument "-ascii" when saving the file, also specifying the extension of the file name, as in the following example:

```
>> save my_variables.dat a b -ascii
```

Saves the variables in 8-digit asci to the file named my_variables.dat. By opening this file, it looks like:

```
  1.0000000e+00
  2.0000000e+00
```

Matlab Classes and Objects

In MATLAB, numbers, strings, arrays or any other type of variable is an object of an appropriate class. For instance:

```
>> b = 10;
>> c = 'Hello World';
>> d = [1 2 3];
>> s.data = 1;
>> whos
  Name      Size            Bytes  Class     Attributes
  b         1x1                 8  double
  c         1x11               22  char
  d         1x3                24  double
  s         1x1               184  struct
```

MATLAB has predefined classes, and user- defined classes. Pre-defined classes are basic classes used in MATLAB language. User-defined classes are those classes developed by users for perform desired operations, which can “replace” basic operations, change them or perform any type of task required by the user. This replacement is called overload.

There are hey terms in object-oriented programming in MATLAB, used to define the classes and related concepts, which are (MATHWORKS, 2017).

Class definition — Description of what is common to every instance of a class.

Properties — Data storage for class instances.

Methods — Special functions that implement operations that are usually performed only on instances of the class.

Events — Messages defined by classes and broadcast by class instances when some specific action occurs.

Attributes — Values that modify the behavior of properties, methods, events, and classes.

Listeners — Objects that respond to a specific event by executing a callback function when the event notice is broadcast.

Objects — Instances of classes, which contain actual data values stored in the objects' properties.

Subclasses — Classes that are derived from other classes and that inherit the methods, properties, and events from those classes (subclasses facilitate the reuse of code defined in the superclass from which they are derived).

Superclasses — Classes that are used as a basis for the creation of more specifically defined classes (that is, subclasses).

Packages — Folders that define a scope for class and function naming.

How classes are defined

Definition of classes in files

Classes in MATLAB can be defined in a similar way that functions or scripts are defined, using .m files. The name of the file must be the same as the name of the class followed by a .m extension.

A second option is to define classes in folders, instead of a single file. This type of class definitions is specially useful for long and complicated classes, with many properties and methods. There are two basic ways of creating folders that contains class definitions:

Path folder – a folder inside MATLAB path.

Class folder – besides being inside a folder in MATLAB path, the folder name starts with an @ character followed by the class name, as in:

@BasicClass

If the class is defined in the last way, a file must be dedicated to the class definition, and other files incorporate methods and properties of the class.

Class definition block

The “classdef” block is the MATLAB instruction containing the class on the beginning of the file, and starting with the keyword “classdef”, terminating with the “end” keyword.

```
classdef (ClassAttributes) ClassName < SuperClass
…
end
```

For instance, the following example is a sealed class named SecondClass that inherits from the FirstClass class. The sealed attributed means that other classes can not be derived from this class.

```
classdef (Sealed) SecondClass < FirstClass
…
end
```

Properties block

Data and attributes of the class are stored in the properties of it, defined in the properties block. There is one block of properties for each set of attribute specifications. The defined attributes and data can also incorporate initial data values. The properties block starts with the keyword “properties” and terminates with the “end” keyword.

```
properties (PropertyAttributes)
…
end
```

In the following example, a set of private properties (data only accessible by the class methods) is defined and a default value is given to “firstproperty”. A second set of property is also defined without special attributes (public by default) with a property “publicproperty” created without a default value.

```
properties (Access = Private)
        firstproperty = 0;
end
properties
```

```
        secondproperty;
end
```

Methods block

Functions and tasks performed by the class are defined in method blocks, one for each set of attribute specifications. This block starts with the keyword “methods” and ends with the keyword “end”.

```
methods (MethodAttributes)
…
end
```

The following example illustrates a public method (no argument is necessary by default) block with a function used to perform an arithmetic operation on the properties of the class.

```
methods
function obj = maths(obj)

obj.Prop3 = obj.Prop1 + obj.Prop2;

end

end
```

Events block

Events block are defined according a unique set of attribute specifications, and they contain the names of the events triggered in the body of the class. The event block starts with the keyword “event” and terminates with the keyword “end”.

```
events (EventAttributes)
…
end
```

The following is an example of a block of protected events with two events, “StateChange” and “NegativeValue”.

```
events (ListenAccess = protected)
        StateChange
        NegativeValue
end
```

Class Attributes

As mentioned earlier, classes may possess special attributes, declared in the "classdef" block, before stating the name of the class.

classdef (Attribute1 – value1, Attribute2 – value2,…) ClassName

The following is a list of valid class attributes for MATLAB (MATHWORKS, 2017).

Attribute Name	Class	Description
Abstract	Logical	An Abstract class can not be instantiated.
AllowedSubclasses	meta.class object or cell array of meta. class objects	Specify subclasses as meta.class objects that can subclass this class. In the case of multiple meta.class objects, use a cell array {}. An empty array is the same as a Sealed class.
ConstrutOnLoad	Logical	If true, the object is constructed when MATLAB loads the object from a MAT-file. The constructor should be implemented so it can have no error if called without arguments.
HandleCompatible	Logical	If true, the class can be used as a superclass for handle classes.
Hidden	Logical	The class does not appear in the output of the superclasses or help functions.
InferiorClasses	meta.class object or a cell array of meta. class objects	This attribute is used to establish a relationship of precedence among classes.
Sealed	Logical	If true, the class can not be subclassed.

Property Attributes

Attributes can be specified to define the behavior of different properties in a class. These customizations can be set up to define control access, data storage or visibility of property. An important aspect to mention is the fact that subclasses do not inherit the superclass member attributes.

The properties attributes, as mentioned earlier, are declared in the header of the property block, following the keyword "property".

properties (Attribute1 – value1, Attribute2 – value2,…)

…

end

The following points shows the valid attributes to all properties defined inside the block that specifies that attributed (MATHWORKS, 2017). The values inside parenthesis are the default values for the attribute, in case it is not specifically changed in the way described above.

Attribute: AbortSet

Class: Logical (false)

Description: In case it is true, MATLAB does not call the set method for the specified attributed if the same value that was already attributed is the new value ..

For handle classes, setting this attribute to true also avoid the call of PreSet and PostSet events.

Attribute: Abstract

Class: Logical (false)

Description: In case it is true, the property has no implementation, and a concrete subclass must redefine the property with the Abstract attribute set to false.

Additionally, abstract properties has special characteristics such as:

- They can not define set or get methods.
- They can not define initial values.
- The derived classes must define the same values as the super class for the SetAccess and GetAcess attributes.

Attribute: Access

Class: enumeration (public), meta.class object or cell array of meta. class objects.

Description: If it is set to public, any other class, superclass, subclass or object may have unrestricted access.

If it is set to protected, only classes and subclasses may have access to the property.

In the case it is private, then only class members are allowed to access the property.

Attribute: Constant

Class: logical (false)

Description: Fixes the value of a property for all instances (objects) of the class if this attribute is set to true. Special features of this attribute are:

Subclasses which inherits constant attributes cannot change them.

Constant properties cannot be Dependent.

SetAccess is ignored.

Attribute: Dependent

Class: logical (false)

Description: It can spare memory by not saving the property value, in case it is set to true. The set and get functions cannot access the property by indexation through the property name. This attribute is useful to calculate data on demand.

Attribute: GetAcess

Class: enumeration (public)

Description: A list of classes that have access to the property. Can be set to:

Public: Anything can access the property.

Protected: Only classes and subclasses can access the property.

Private: Access to the property is only allowed by class members (not from other classes or subclasses).

The attribute defines the classes that have get access to the listed properties. The classes may be specified as meta.class objects.

Attribute: GetObservable

Class: logical (false)

Description: This attribute allows the creation of listeners to access the listed properties, in case it is set to true. The listeners are called whenever property values are queried.

Attribute: Hidden

Class: logical (false)

Description: The attribute defines if the listed properties can be shown in a property list, as the Property Inspector, calls to set or get, etc. In this sense, hidden properties are not displayed by MATLAB in the command window, nor the value or the name of it.

Attribute: NonCopyable

Class: logical (false)

Description: The attribute determines if a property is copied if an instance of the class is copied. NonCopyable attribute can only be set to true in handle classes.

Attribute: SetAccess

Class: enumeration (public)

Description: Can be defined as:

- Public: Anyone have access to the listed properties.
- Protected: Access is permitted only by classes and subclasses (no functions or user).
- Private: Access to the listed properties is only permitted by class members (no subclasses or other classes).
- Immutable – The listed properties can only be set inside the constructor function.

This attribute is used to list the classes that have set access to the listed properties in the block. Classes should be specified as a single meta.class object or a cell array of meta.class objects.

An empty cell array {} of meta.class objects is the same as defining private access value to this attribute.

Attribute: SetObservable

Class: logical (false)

Description: If it is true and it is a handle class property, then listeners can be created to access the listed properties, wherever such properties are modified.

Attribute: Transient

Class: logical (false)

Description: This attribute specifies which properties are not to be saved when an instance of the class is saved to a file.

Methods Attributes

The behavior of methods can be changed by defining specific attributes to them. Such attributes change how access, visibility and implementation behaves during the lifetime of an instance of the class.

The attributes are specified in the methods block, after the keyword "methods".

```
methods (Attribute1 – value1, Attribute2 – value2,…)
…
end
```

The supported method attributes are mentioned below, and should be inserted in the class code according the convention referred above.

Attribute: Abstract

Class: logical (false)

Description: Methods with this attribute have no implementation, but subclasses can use the line containing arguments when implementing the method. Such subclasses do not need to necessarily implement the same number of input and output arguments, although it is recommended to use the same signature when implementing the abstract method. The method does not have the function or end keywords, rather only the function syntax, e.g [x,y] = AbstractFunction(a,b).

Attribute: Access

Class: enumeration (public)

Description: This attribute defines what code can call the listed method:

Public: Unrestricted access.

Protected: Method can only be accessed in classes or subclasses.

Private: Method is only accessible inside the own class members.

Optionally, a list of classes who can access the method can be provided. This are meta.class objects, listed using cell array. An empty cell array {} is the same as private access.

Attribute: Hidden

Class: logical (false)

Description: If the hidden attribute is false, the listed methods are shown using the methods or methodsview commands in MATLAB. The methods can be hidden from the user or any other object trying to observe the methods by setting this attribute to true. In this case, the methods will not be listed when using the commands referred above, and the command is method does not return true for the hidden method.

Attribute: Sealed

Class: logical (false)

Description: The attribute fixes the method, in the sense that it cannot be redefined by a subclass. Attempting to do so causes an error in the subclass.

Attribute: Static

Class: logical (false)

Description: A static method does not depend on an instance of the class and naturally does not need an object argument.

Typical Workflow to Develop Classes

Defining a Class

This section introduces how to develop a class to represent a familiar concept in scientific computing. The concept worked on is a generic storage box (could be a water tank, a boiler, distillation column, an ant colony, etc), used to store any generic material (or concept, in the most abstract sense).

The first step is to define the elements and the operations that forms the abstract storage box. For example, a storage box has:

- An ID (Identification Number), so the user knows to which

StorageBox object he is referring to specifically.

- An amount of thing inside (balance)
- Optionally a status (full, empty, open, closed, etc)

A list of different operations can be performed in the storage box, such as:

- Create an object for each storage box
- Add things
- Remove things
- Display the status of the box
- Save and load the StorageBox object

The StorageBox object will have an option that, if the user tries to remove things when it is already empty, it issues a notice to other elements that are designated to listen these notices. An Operator object will be designed to perform this operation over the StorageBox. He will determine the status of the StorageBox, assigning one of the following values:

- Full – StorageBox balance has 100 of things.
- Empty - StorageBox balance does not have things (0).
- Open - StorageBox balance has positive value of things.

With these features clear, the properties and methods of the objects StorageBox and Operator are clearly defined. It is recommended to only include functionality that meets the requirements or specific objectives of the program. For instance, if the StorageBox never really gets full, there is no reason to implement the status "Full" to it. The developed classes always should have room to be upgraded, so new functionalities can be added according the demand.

Specify Class Components

Some formal names have to be used in order to identify the properties that will store each data of the StorageBox. In the present case, we define the following properties:

- IdentificationNumber: This property is used to store an identification of the StorageBox object. MATLAB assigns a value to this property when you create an instance of the class.

Only StorageBox class methods can set this property. The SetAccess attribute is private.

- Balance: This property stores the current amount of “thing” inside the StorageBox (can be water, heat, ants, or whatever is being studied). The operation of insert or remove assigns values to this property. Only StorageBox class methods can set this property. The SetAccess attribute is private.
- Status — The StorageBox class defines a default value for this property. The Operator class methods change this value whenever the value of the Balance falls below 0 or rises above 100. The Access attribute specifies that only the Operator and StorageBox classes have access to this property.
- BoxListener — Storage for the NegativeAmount and OverflowedAmount event listener. Saving a StorageBox object does not save this property because you must recreate the listener when loading the object.

The operations performed by/at the StorageBox class are:

- StorageBox: Acceps an identification number and an initial balance to create an object that represents a storage box.
- insert: updates the StorageBox object balance by adding the specified amount of things.
- remove: updates the StorageBox object balance by removing the specified amount of things.
- getStatement: Displays information about the storage box.
- loadobj: Recreates the operator listener when you load the object from a MAT-file.

The events are triggered by the operator inside the methods of the class. In the present case, the StorageBox class triggers an event when the removal of things of the box results in NegativeAmount balance. Therefore, the NegativeAmount event occurs inside the remove method.

On the same principle, the StorageBox class triggers an event when the addition of things on the box results in OverflowedAmount balance. Therefore, the OverflowedAmount event occurs inside the insert method.

The definition of events is done inside an events block. The notify handle class method is responsible of triggering the event.

StorageBox Class Implementation

The StorageBox class should have only one set of data associated with one object. To do that, the StorageBox class has to be implemented as a handle class, as explained in MATHWORKS (2017). All copies of a given handle object refer to the same data. In this way, it is possible to generate different references to the same StorageBox, without creating duplicates or unlinked copies of the same object.

The StorageBox class can be created in a single .m file, as shown below. Each section of the code is explained using MATLAB comments (every line of text that starts with a "%").

StorageBox.m

```
classdef StorageBox < handle
   % STORAGEBOX inhherits from handle class
   % because it is allowed only one copy of
   % each box.

   properties (Access = ?Operator)
      Status = 'open'
      % The status is determined by the current balance.
      % Access is limited to the StorageBox class and the
      % Operator class
   end

   properties (SetAccess = private)
      IdentificationNumber
      Balance
   end

   properties (Transient)
      BoxListener
   end
```

```
events
   NegativeAmount
   OverflowedAmount
end


methods
   function obj = StorageBox(ID,InitialBalance)
      % Constructor method for StorageBox
      obj.IdentificationNumber = ID;
      obj.Balance = InitialBalance;
      obj.BoxListener = Operator.addBox(obj);
   end

   function insert(obj,amt)
      % Adds an amount amt to the balance of the
      % StorageBox obj.
      if obj.Balance > 100
         disp([‘Box ‘,num2str(obj.IdentificationNumber),...
         ‘ is full.’])
         return
      end
      newbal = obj.Balance + amt;
      obj.Balance = newbal;
      if newbal > 0
         obj.Status = ‘open’;
      end
      if newbal > 100
         notify(obj,’OverflowedAmount’)
```

```
    end
end

function remove(obj,amt)
    % Removes an amount amt to the balance of the
    % StorageBox obj.
    if (strcmp(obj.Status,'closed')&& ...
    obj.Balance < 0)
        disp(['Box ',num2str(obj.IdentificationNumber),...
        ' has been closed.'])
        return
    end
    newbal = obj.Balance - amt;
    obj.Balance = newbal;
    if newbal < 0
        notify(obj,'NegativeAmount')
    end
end

 function getStatement(obj)
     % Generates a short statement of the StorageBox.
     disp('-------------------------')
     disp(['Box: ',num2str(obj.IdentificationNumber)])
     bal = sprintf('%0.2f',obj.Balance);
     disp(['CurrentBalance: ',bal])
     disp(['Box Status: ',obj.Status])
     disp('-------------------------')
 end
```

```
    end % of methods

    methods (Static)

    function obj = loadobj(s)
        % function used to regenerate the obj from a file.
        if isstruct(s)
            id = s.IdentificationNumber;
            initBal = s.Balance;
            obj = StorageBox(id,initBal);
        else
            obj.BoxListener = Operator.addBox(s);
        end
    end

    end % of methods (Static)
end
```

Operator Class Implementation

The Operator is a class used to provide services to the StorageBox class. It is responsible for listening to insertions and removals in the StorageBox, also to attribute a status according the current balance of the box. When the StorageBox triggers the NegativeAmount or the OverflowedAmount events, the Operator resets the StorageBox status.

Because the Operator class has no data, it has no properties. The StorageBox object stores the handle of the listener object.

The Operator class performs two operations:

- Assign a status to each box as a result of a removal or a insertion of things
- Adds a box to the system

Operator Class Components

Basically, the Operator class performs two methods:

assignStatus — Method that assigns a status to a StorageBox object. Serves as the listener callback.

addBox — Method that creates the NegativeAmount and the OverflowedAmount listeners.

The Operator class can be created in a single .m file, as shown below. Each section of the code is explained using MATLAB comments (every line of text that starts with a "%").

```
classdef Operator

   methods (Static)

      function assignStatus(obj)
         if obj.Balance < 0
            if obj.Balance < -200
               obj.Status = 'closed';
            else
            obj.Status = 'overdrawn';
            end
         end
         if obj.Balance > 100
            obj.Status = 'overflowed';
         end
      end

      function lh = addBox(obj)
         lh = addlistener(obj, 'NegativeAmount', ...
         @(src, ~)Operator.assignStatus(src));
         lh = addlistener(obj, 'OverflowedAmount', ...
         @(src, ~)Operator.assignStatus(src));
```

```
        end


    end
end
```

Using the StorageBox class

The intent of this subsection is to demonstrate how MATLAB classes behave, by creating and manipulating a StorageBox object. First, create a box with ID “B01”and initial balance of 50 things.

```
>> BA = StorageBox(‘B01’,50)
BA =
  StorageBox with properties:
    IdentificationNumber: ‘B01’
                 Balance: 50
             BoxListener: [1x1 event.listener]
```

Remove 10 things of the box and check the status:

```
>> remove(BA,10)
>> getStatement(BA)
-------------------------
Box: B01
CurrentBalance: 40.00
Box Status: open
-------------------------
```

In order to test the change in the status of the BOX, remove the remaining 40 + 1 things in the Box and check the status again.

```
>> remove(BA,41)
>> getStatement(BA)
-------------------------
Box: B01
CurrentBalance: -1.00
Box Status: overdrawn
```

```
------------------------
```

Because the balance in the box became negative, the Operator changed the status of the box to “overdrawn”, so from now on every method that checks on the status of the box will know that the balance is a negative value. To check if the Operator is correctly closing the box if the balance reaches values below -200, remove 200 things of the box and check the status.

```
>> remove(BA,200)
>> getStatement(BA)
------------------------
Box: B01
CurrentBalance: -201.00
Box Status: closed
------------------------
```

It can be seen that the status of the box was correctly set to “closed”, which means that no more things can be removed from the box. To check if this implementation is correct, try to remove one thing from the box and check the status again.

```
>> remove(BA,1)
Box B01 has been closed.
>> getStatement(BA)
------------------------
Box: B01
CurrentBalance: -201.00
Box Status: closed
------------------------
```

When we try to remove one thing from the box now, it generates a message noticing that the box is closed, so the balance should not change. This can be checked as above, by generating the statement again and observing that the balance is the same as before trying to remove one more thing. The box can be reopened by adding enough things so as the balance becomes positive again. To do so, add 202 things in the box and check the status.

```
>> getStatement(BA)
-------------------------
Box: B01
CurrentBalance: 1.00
Box Status: open
-------------------------
```

It can be observed that the operator correctly changed the status of the box back to open, since the balance indicates that there is one (1) positive thing.

Other classes can be derived from this former StorageBox class, to represent specific volume elements, with additional properties such as the dimensions of the box, some internal dynamics in the box (e.g chemical reaction of animal reproduction) and any other property or method that may be of interest to the developer of the class.

An Example: The “Diffusive” Storage Box network

In this section, we will improve the developed StorageBox by implementing a system with connected storage boxes that can interact with one another. Suppose a simple network of three generic storage boxes (may be water tanks, storage facilities, ant colonies, etc) connected as the following figure

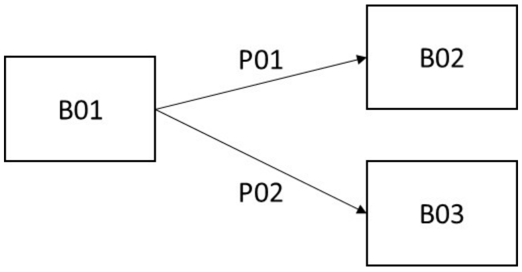


Figure 27: Simple storage boxes network.

Each box has an initial amount of generic things. Suppose box B01 has 100 things, B02 and B03 has 50 things. We may suppose that there are connections between the box B01 and B02, and B01 and B03. In these connections, the things are transferred from one box to another following a diffusive rule, which means:

$$Flux = Constant * \left(Box_{right} - Box_{left} \right)$$

Where Flux is the number of things that are transferred from the box with more things to the box with less things. The Flux is linearly proportional to the difference of amount of things in the Box, which means that the higher the difference of things in the connected boxes, the higher the flux of things.

In order to evaluate how material is transferred from one box to the other, it is necessary to simulate the system for some steps, and record the number of things in the boxes at each time step, as well as the fluxes of things. For the moment, it is assumed a discrete system, so time will not be incorporated to the system.

This simple example could be implemented in a spreadsheet such as Microsoft Excel ®. Supposing that the constant of the connector P01 is equal to 0.1, and the constant of the connector P02 is equal 0.2. The simulation of 10 steps results in the following table:

step	B01	B02	B03	P01	P02
1	100	50	50	5	10
2	85	55	60	3	5
3	77	58	65	1.9	2.4
4	72.7	59.9	67.4	1.28	1.06
5	70.36	61.18	68.46	0.918	0.38
6	69.062	62.098	68.84	0.6964	0.0444
7	68.3212	62.7944	68.8844	0.55268	-0.11264
8	67.88116	63.34708	68.77176	0.453408	-0.17812
9	67.60587	63.80049	68.59364	0.380538	-0.19755
10	67.42289	64.18103	68.39609	0.324186	-0.19464

The following figure illustrates the results for the balances of the boxes and for the fluxes. It can be seen, as expected for a diffusive system, that the amount of things in each box converges to the same value, that is, given enough time of simulation, all of the boxes will have the same amount of things, independently of the initial condition.

Regarding the fluxes, it reduces progressively as the difference between the balances of the boxes reduce, until it converges to zero, given enough simulation time.

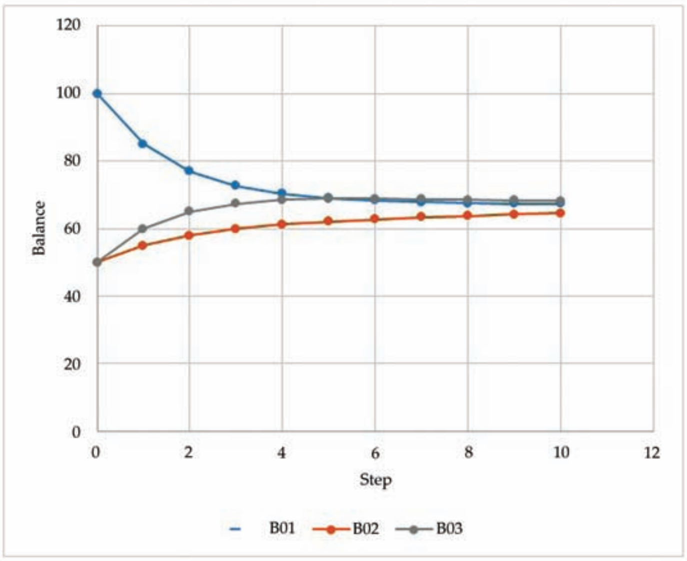


Figure 28: Balance of the three boxes network.

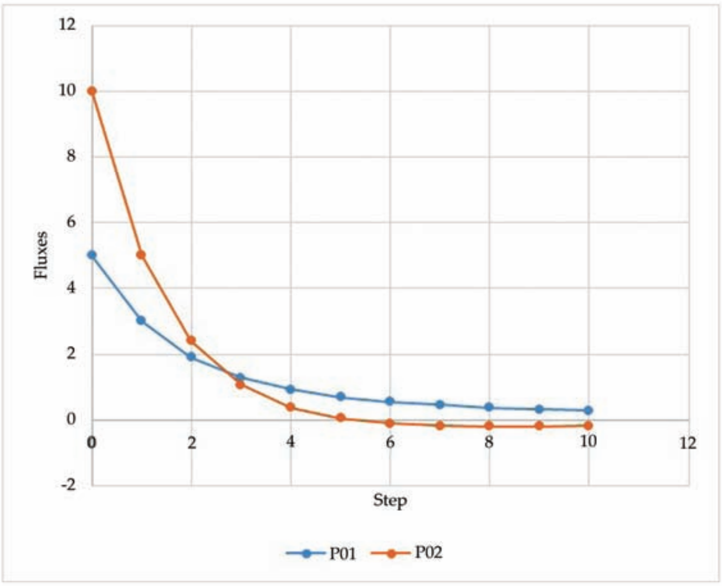


Figure 29: Fluxes of the three boxes network.

As mentioned before, the implementation of such problem in a spreadsheet is simple. However, a more complicated network with hundreds of boxes and connections may make this type of implementation unfeasible. That is the reason why it is exemplified here this network in MATLAB using object-oriented approach.

The StorageBox classes were developed earlier in this chapter, and it will not be changed. The connections are also a class that stores what is the box to the right and what is the box to the left, and it calculates the flux according the equation showed above.

The Pin class is the name that will be used to implement the connectors of boxes. It most store some information, which is:

- **Prev:** stores what box is to the left of the connector.
- **Next:** stores what box is to the right of the connector.
- **Const:** Is the constant used to calculate the flux. Can be any positive value. However, very high values may cause unstable fluxes for this discrete approach. For instance, if the initial balance of the box to the left is 100 and the box to the right 0, using a constant of 10. The initial flux will be equal to

$$Flux = 10 * (100 - 0) = 1000$$

So, the initial flux will be equal 1000 things, which means that 1000 things will be removed from the box with more things (right one with 100 things) generating a balance of -900 things!

- Flow: Used to store the calculate flow according the equation mentioned above.

The methods to be implemented in the Pin class are:

- Pin: the constructor method, used to create the object.
- equation: the method that calculates the flux.

The implementation of the Pin class is done in a single .m file Pin.m, as follows:

```
classdef Pin < handle
  % Pin class used to connect Boxes


  properties (SetAccess = private)
    Prev % stores the box to the left
    Next % stores the box to the right
```

```
        Const % proportional constant of the flux
        Flow
    end

    methods
        function obj = Pin(Input,Output,Const)
            % Constructor method
            obj.Prev = Input;
            obj.Next = Output;
            obj.Const = Const;
            obj.Flow = obj.Const.*(obj.Prev.Balance - obj.Next.Balance);
        end

        function equation(obj)
            % Method used to calculate the flow
            obj.Flow = obj.Const.*(obj.Prev.Balance - obj.Next.Balance);
        end
    end % of methods
end
```

We also inherit this class from the handle class, since it is not desirable to have different copies of the same Pin object.

It is recommendable to test the developed class, in order to find and correct any bugs in the code. To do so, create two boxes and connect them. After that, test if the equation is calculating the correct flux. In the following example, it is implemented two boxes, B01 and B02, with an initial balance of 100 and 50, respectively. The boxes are connected with a Pin P01, with constant of value 0.1. The initial flux should be:

$$Flux = 0.1*\left(100-50\right) = 0.1*50 = 5$$

```
>> B01 = StorageBox('B01',100)
B01 =
  StorageBox with properties:
```

```
  IdentificationNumber: 'B01'
             Balance: 100
         BoxListener: [1x1 event.listener]
>> B02 = StorageBox('B02',50)
B02 =
  StorageBox with properties:

  IdentificationNumber: 'B02'
             Balance: 50
         BoxListener: [1x1 event.listener]
>> P01 = Pin(B01,B02,0.1)
P01 =
  Pin with properties:
    Prev: [1x1 StorageBox]
    Next: [1x1 StorageBox]
   Const: 0.1000
    Flow: 5
```

Before implementing the simulation itself, it is necessary to be able to record the data generated during the simulation. To do so in an object-oriented approach, it is developed a Recorder class, which is responsible of, at each step of simulation, to record in a matrix the values of the boxes balances, as well as the generated fluxes between them. To know which boxes or connector generates data to the recorder, it should have properties that enables to store such elements. Also, the recorded data is another property of the class. In summary, the properties of the Record class are:

- **Boxes:** it stores the boxes in the system.
- **Pins:** it stores the connectors in the system.
- **States:** a matrix to store the recorded balances of each box.
- **Flows:** a matrix to store the recorded flux of each connector.

The only important method of the Recorder class, besides the constructor itself, is the record function. This method goes through each box and connector, collecting the current balance or flow data and storing in an appropriate matrix.

The following code shows the Recorder class, which is written in a single .m file:

```
classdef Recorder < handle
  % Pin class used to Record data from simulation

  properties (SetAccess = private)
    Boxes % collection of the boxes in the system
    Pins % collection of the connectors in the system
    States % matrix with balances of the boxes at each step
    Flows % matrix with fluxes of the connectors at each step
  end

  methods
    function obj = Recorder(Boxes,Pins)
      % Constructor method
      obj.Boxes = Boxes;
      obj.Pins = Pins;
      obj.States = zeros(1,length(Boxes));
      obj.Flows = zeros(1,length(Pins));
    end

    function record(obj,count)
      % The “main” method of the recorder, used to save in the matrix
      % the recorded data of box balances and connector fluxes
      for i=1:length(obj.Boxes)
        obj.States(count,i) = obj.Boxes(i).Balance;
      end
```

```
            for i=1:length(obj.Pins)
               obj.Flows(count,i) = obj.Pins(i).Flow;
            end
        end
    end % of methods
end
```

It is recommendable to test the Recorder class in order to know if has any bug and if all the methods and properties are working as expected. In the following code, we use the previously developed boxes B01 and B02, with the connector P01 and record the data available in the line 10 of the matrices of States and Flows of the Recorder R01 object.

```
>> R01 = Recorder([B01 B02],[P01]);
>> record(R01,10)
>> R01.States
ans =
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
   100    50
>> R01.Flows
ans =
     0
     0
     0
     0
```

```
0
0
0
0
0
5
```

The next step to simulate the network of boxes is to develop the Simulator class itself. One again, it is undesirable to have copies of the same Simulator in different parts of the program, so this class also should inherit from the handle class. The Simulator, for the moment, may have only one data, that is the Recorder object. To simulate for a given number of steps, a method is used with the Simulator object and the number of steps as arguments for the function.

The simulate function uses a for loop to, at each step, simulate the system, record the data using the recorder and update the balances of the boxes.

The following code shows the implementation of the Simulator class, in a single .m file.

```
classdef Simulator < handle
   % Simulator class inhherits from handle class
   % because it is allowed only one copy of
   % each simulator.

   properties (SetAccess = private)
      Recorder
   end

   methods
      function obj = Simulator(Recorder)
         % Simulator constructor method
         obj.Recorder = Recorder;
      end
```

```
        function simulate(obj,numsteps)
            % The “main” function of the Simulator
            % numsteps are the number of steps used to simulate
            for j=1:numsteps
                for i= 1: length(obj.Recorder.Pins)
                    % simulate the system using the connectors equation
                    equation(obj.Recorder.Pins(i));
                end
                % record the data using the Recorder object
                record(obj.Recorder,j);
                for i= 1: length(obj.Recorder.Pins)
                    % update the balance of the boxes by adding / removing
                    % the flows
                  remove(obj.Recorder.Pins(i).Prev,obj.Recorder.Pins(i).Flow)
                    insert(obj.Recorder.Pins(i).Next,obj.Recorder.Pins(i).Flow)
                end
            end
        end
    end % of methods
end
```

To test the Simulator class, it may be directly implemented the example mentioned at the beginning of this subsection. In resume, three StorageBox objects are implemented, B01, B02 and B03 and two Pin objects, P01 and P02 with 0.1 and 0.2 as the flux constant respectively. A recorder R01 is used to record the data of the simulation. A simulator S01 is used to simulate the system. In order to approach the time that all the balances are equal, the simulator uses 100 steps.

This simulation is named test1.m

```
B01 = StorageBox(‚B01‘,100);
B02 = StorageBox(‚B02‘,50);
```

```
B03 = StorageBox('B03',50);
P01 = Pin(B01,B02,0.1);
P02 = Pin(B01,B03,0.2);
R01 = Recorder([B01 B02 B03],[P01 P02]);
S01 = Simulator(R01);
simulate(S01,100);
States = S01.Recorder.States;
Flows = S01.Recorder.Flows;
figure
plot(States,'-+')
ylabel('Balances')

figure
plot(Flows,'-o')
hold on
ylabel('States')
```

The generated results for the balances of the boxes is shown in the figure below, followed by the calculated fluxes between the boxes.

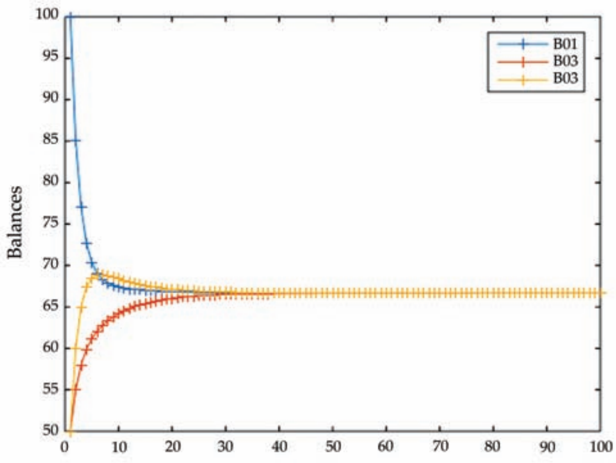


Figure 30: Balances of the three boxes network example – Object-Oriented implementation.

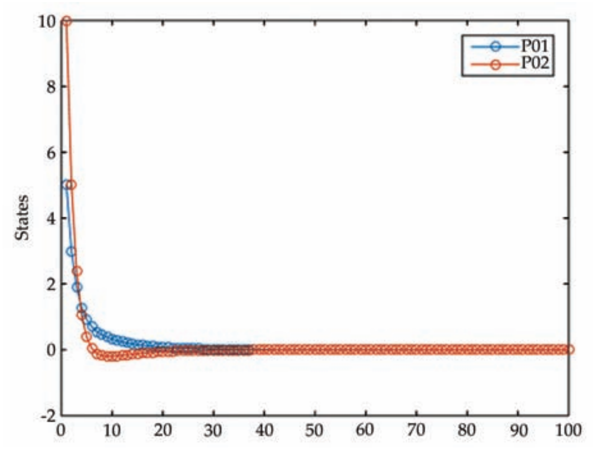


Figure 31: Fluxes between the three boxes – Object-Oriented implementation.

For this case, it can be seen that approximately in the step 40 (forty), the fluxes are already almost zero (less than 0.01) and the balances of the boxes converges to a similar value (in this case, approximately 66.6667).

JAVA

While not by nature a scientific computing language, Java has grown in use since it is naturally object-oriented. According Boisvert (2001), Java is portable at both the source and object format levels. Both the source format for Java (a .java file) and the object format (the bytecode in a .class file) are expected to behave the same on any computer with the appropriate Java compiler and Java virtual machine. Second, Java code is safe to the host computer. Java implements a simple object-oriented model with important features (e.g., single inheritance, garbage collection) that facilitate the learning curve for newcomers. But the most important thing Java has to offer is its pervasiveness, in all aspects. Java runs on virtually every platform Universities all over the world are teaching Java to their students. Many specialized class libraries, from three-dimensional graphics to online transaction processing, are available in Java.

However, according to the same author there are still some issues to be addressed. It fails to provide some of high-level numerical features, such as complex numbers and true multidimensional arrays.

Basic Java language characteristics

The first main characteristic of Java language is that it is a compiled language. This is an important feature when compared with some other languages used for scientific computing such as MATLAB, which is an interpreted language. The fact that Java is compiled means that the code is converted into bytecodes, which makes it very fast.

The syntax in Java is free formatted. This means that the use of indentation in the code and blank lines are free and does not change the significance in the code. The coding blocks are formed using clear delimiters, such as ";" and "{}".

Variables in Java must be declared before use, and any error in the program can be identified during compilation rather than during the program run. These features make Java a self-documenting language, and highly suitable for large-scale software systems.

The fact already mentioned that Java is naturally object-oriented facilitate the programming procedure and organization of the software, as the program parts are separated in modules called classes, and objects are created in runtime, to define a specific problem domain.

Java primer

To start developing java programs in our machine, it is necessary to have a compiler so the text code can be translated and run. For simple programs, contained in a single file, an available option it to use online compilers. Such compilers are web pages where the user provides the code, the web site has a compiler embedded with translates the .java file to the .class and .jar file. One example of a free online compiler is https://www.compilejava.net/.

A second option, which is better specially for more complex programs is to obtain and install a compiler in the computer. The following are the major Java compilers:

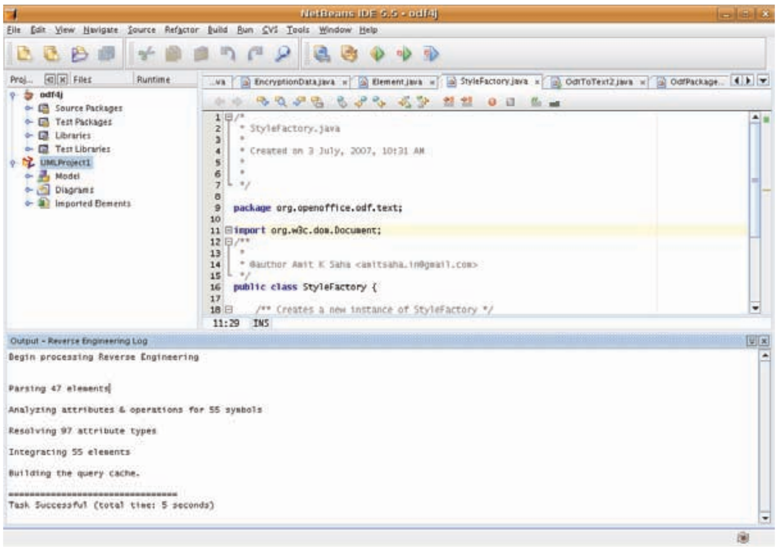
- The Java Programming Language Compiler (javac), included in the Java Development Kit from Oracle Corporation, open-sourced since 13 November 2006.
- GNU Compiler for Java (GCJ), a part of the GNU Compiler Collection, which compiles C, C++, Fortran, Pascal and other

programming languages besides Java. It can also generate native code using the back-end of GCC.

- Eclipse Compiler for Java (ECJ), an open source incremental compiler used by the Eclipse project.

A third option is, instead of obtaining only the compiler, to obtain an Interactive Development Environment (IDE), where the developer is able to write, compile and run the code in a single piece of software, making the development much faster. Again, there are a series of options available:

- NetBeans: open source solution for coding Java. It supports development of all Java application types (Java SE, JavaFX, Java ME, web, EJB and mobile applications). It has an advantage of being modular by design, so it can be extended through plugins to enhance functionality. Besides that, it also supports other languages (PHP, C/C++ and HTML5). It is multi-platform (Microsoft Windows, Mac OS X, Linux, Solaris and other platforms supporting a compatible JVM) and can be used for work in Cloud applications, integrated with the Google App Engine.

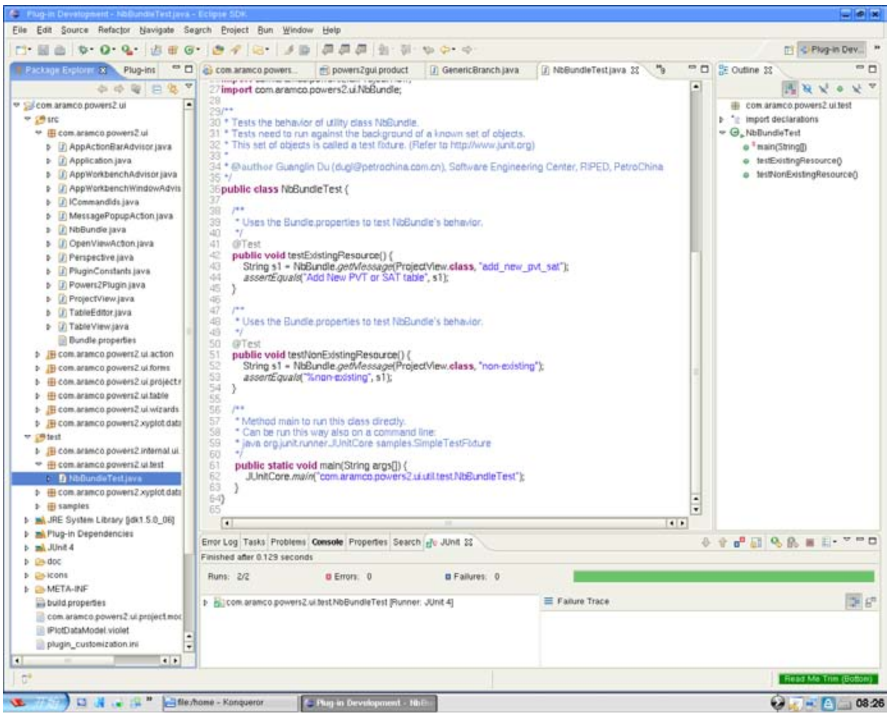

NetBeans IDE 5.5 - odf4j
File Edit View Navigate Source Refactor Build Run CVS Tools Window Help
Proj... Files Runtime
odf4j
Source Packages
Test Packages
Libraries
Test Libraries
UMLProject1
Model
Diagrams
Imported Elements
...va EncryptionData.java Element.java StyleFactory.java OdfToText2.java OdfPackage...
11:29 INS
Output - Reverse Engineering Log

```
/*
 * StyleFactory.java
 *
 * Created on 3 July, 2007, 10:31 AM
 *
 *
 */

package org.openoffice.odf.text;

import org.w3c.dom.Document;
/**
 *
 * @author Amit K Saha <amitsaha.in@gmail.com>
 */
public class StyleFactory {

    /** Creates a new instance of StyleFactory */
```

```
Begin processing Reverse Engineering


Parsing 47 elements

Analyzing attributes & operations for 55 symbols

Resolving 97 attribute types

Integrating 55 elements

Building the query cache.

================================
Task Successful (total time: 5 seconds)
```

Figure 32: NetBeans screenshot. Source: http://wiki.netbeans.org/NetbeansUML

Eclipse: Besides providing the separate compiler, Eclipse is a cross-platform IDE which allows software development either for mobile, desktop, web or enterprise domains. The software contains a base workspace with an extensible plug-in system to customize the IDE. The plugins also enable the development of softwares in a series of other programming languages (C, C++, JavaScript,, Perl, PHP, Prolog, Python, R, Ruby).

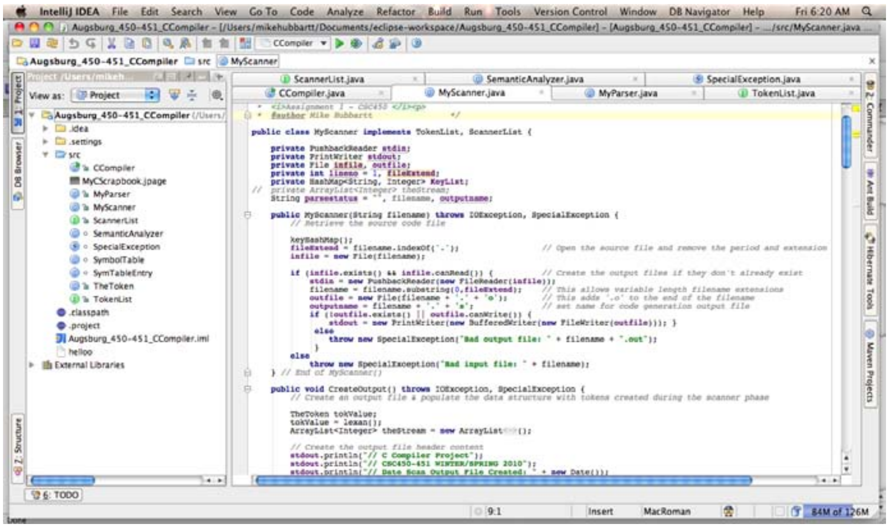

Plug-in Development - NbBundleTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Plug-ins
com.aramco.powers...
powers2gui.product
GenericBranch.java
NbBundleTest.java
Outline
com.aramco.powers2.ui.test
import declarations
NbBundleTest
main(String[])
testExistingResource()
testNonExistingResource()
Error Log Tasks Problems Console Properties Search JUnit
Failure Trace

```
import com.aramco.powers2.ui.NbBundle;

/**
 * Tests the behavior of utility class NbBundle.
 * Tests need to run against the background of a known set of objects.
 * This set of objects is called a test fixture. (Refer to http://www.junit.org)
 *
 * @author Guanglin Du (dugl@petrochina.com.cn), Software Engineering Center, RIPED, PetroChina
 */
public class NbBundleTest {

   /**
    * Uses the Bundle.properties to test NbBundle's behavior.
    */
   @Test
   public void testExistingResource() {
      String s1 = NbBundle.getMessage(ProjectView.class, "add_new_pvt_sat");
      assertEquals("Add New PVT or SAT table", s1);
   }

   /**
    * Uses the Bundle.properties to test NbBundle's behavior.
    */
   @Test
   public void testNonExistingResource() {
      String s1 = NbBundle.getMessage(ProjectView.class, "non-existing");
      assertEquals("%non-existing", s1);
   }

   /**
    * Method main to run this class directly.
    * Can be run this way also on a command line:
    * java org.junit.runner.JUnitCore samples.SimpleTestFixture
    */
    public static void main(String args[]) {
       JUnitCore.main("com.aramco.powers2.ui.util.test.NbBundleTest");
    }
}
```

```
Finished after 0.129 seconds
Runs: 2/2      Errors: 0      Failures: 0
com.aramco.powers2.ui.test.NbBundleTest [Runner: JUnit 4]
```

Figure 33: Eclipse screenshor. Source: http://www.eclipse.org/screenshots/#.

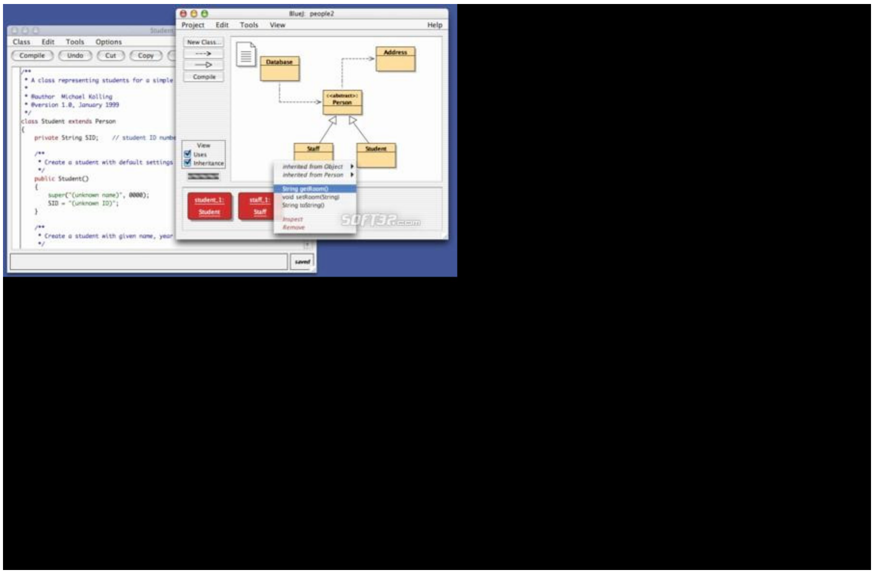
IntelliJ IDEA Community Edition: A free Java IDE mainly used for Android App development, Scala, Groovy, Java SE and Java programming. It has special features such as a visual GUI builder, code completion, code inspection among others. There is a commercial edition with additional features and it can be purchased if the developer requires such resources.



```
 * <I>Assignment 1 - CSC455 </I><p>
 * @author Mike Hubbartt           */

public class MyScanner implements TokenList, ScannerList {

    private PushbackReader stdin;
    private PrintWriter stdout;
    private File infile, outfile;
    private int lineno = 1, fileExtend;
    private HashMap<String, Integer> KeyList;
//  private ArrayList<Integer> theStream;
    String parsestatus = "", filename, outputname;

    public MyScanner(String filename) throws IOException, SpecialException {
        // Retrieve the source code file

        keyHashMap();
        fileExtend = filename.indexOf('.');                  // Open the source file and remove the period and extension
        infile = new File(filename);

        if (infile.exists() && infile.canRead()) {           // Create the output files if they don't already exist
            stdin = new PushbackReader(new FileReader(infile));
            filename = filename.substring(0,fileExtend);     // This allows variable length filename extensions
            outfile = new File(filename + '.' + 'o');        // This adds '.o' to the end of the filename
            outputname = filename + '.' + 's';               // set name for code generation output file
            if (!outfile.exists() || outfile.canWrite()) {
                stdout = new PrintWriter(new BufferedWriter(new FileWriter(outfile))); }
             else
                throw new SpecialException("Bad output file: " + filename + ".out");
             }
        else
            throw new SpecialException("Bad input file: " + filename);
    } // End of MyScanner()

    public void CreateOutput() throws IOException, SpecialException {
        // Create an output file & populate the data structure with tokens created during the scanner phase

        TheToken tokValue;
        tokValue = lexan();
        ArrayList<Integer> theStream = new ArrayList<>();

        // Create the output file header content
        stdout.println("// C Compiler Project");
        stdout.println("// CSC450-451 WINTER/SPRING 2010");
        stdout.println("// Date Scan Output File Created: " + new Date());
```

Figure 34: IntelliJ IDEA Community Edition screenshot. Source: https://mhreviews.files.wordpress.com.

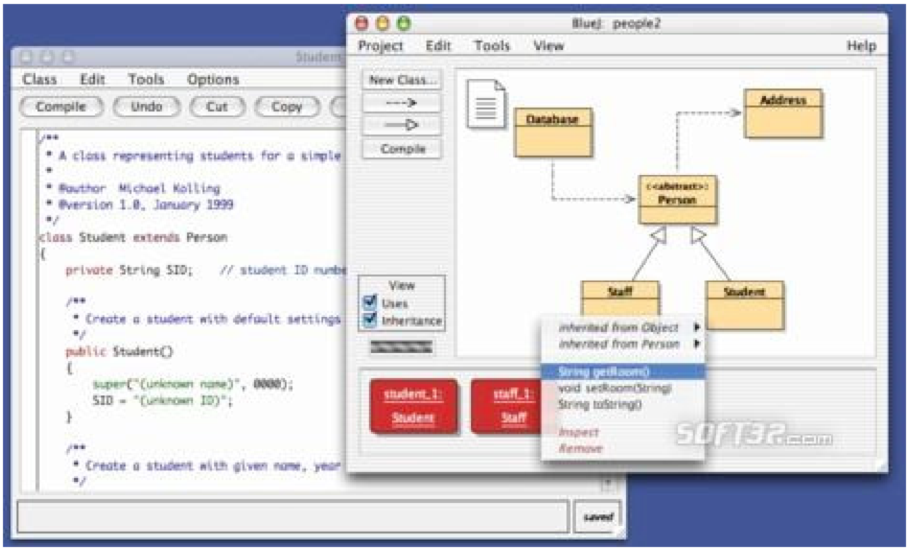
AndroidStudio: This Java IDE from Google is specifically designed for development of Android apps. Nevertheless, it is capable of running and editing some Java code.



```
/**
 * A class representing students for a simple
 *
 * @author  Michael Kolling
 * @version 1.0, January 1999
 */
class Student extends Person
{
    private String SID;    // student ID numbe

    /**
     * Create a student with default settings
     */
    public Student()
    {
        super("(unknown name)", 0000);
        SID = "(unknown ID)";
    }

    /**
     * Create a student with given name, year
     */
```

Figure 35: Android Studio screenshot. Source: https://img.utdstc.com.

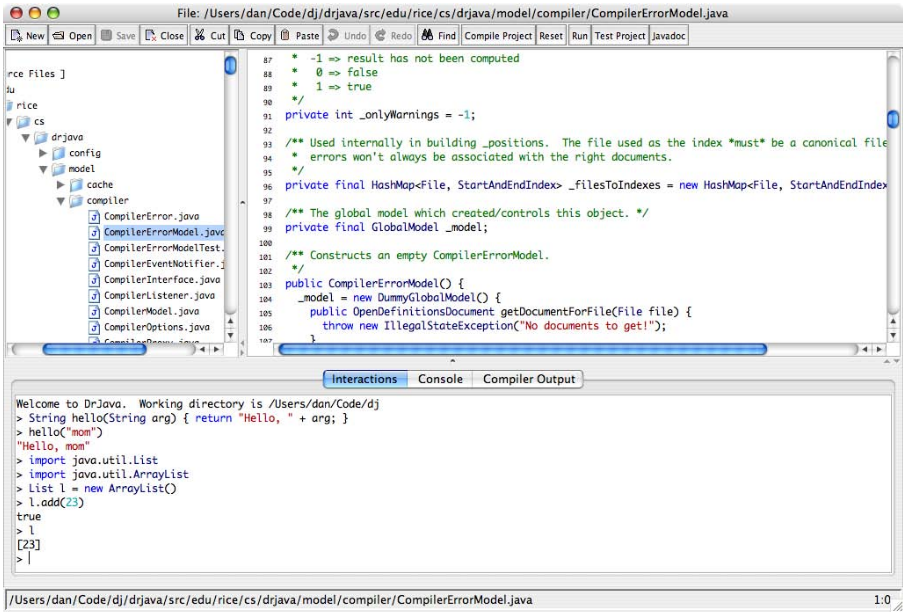
BlueJ: A software directed for education purposes. Nonetheless, it is also appropriate for small scale software development. It uses JDK (Java Development Kit) as a support tool to run. The main screen graphically shows the class structure of an application under development and objects can be interactively created and tested. This interaction facility, combined with a clean, simple user interface, allows easy experimentation with objects under development and this allows beginners to get started more quickly, and without being overwhelmed (IDR Solutions, __).



```
/**
 * A class representing students for a simple
 *
 * @author  Michael Kolling
 * @version 1.0, January 1999
 */
class Student extends Person
{
    private String SID;    // student ID numbe

    /**
     * Create a student with default settings
     */
    public Student()
    {
        super("(unknown name)", 0000);
        SID = "(unknown ID)";
    }

    /**
     * Create a student with given name, year
     */
```

Figure 36: BlueJ screenshot. Source: https://bluej.soft32.com/

DrJava: A free Java Ide designed mainly for educational purposes. It is extremely lightweight and it provided an intuitive interface and the possibility to interactively evaluate the code. Its main feature is for it to be used as a unit testing tool, a source level debugger, an interactive pane for evaluating text of the program, intelligent program editor and can be used for more depending on your requirements (IDR Solutions, ___).



```
 *  -1 => result has not been computed
 *   0 => false
 *   1 => true
 */
private int _onlyWarnings = -1;

/** Used internally in building _positions.  The file used as the index *must* be a canonical file
 *  errors won't always be associated with the right documents.
 */
private final HashMap<File, StartAndEndIndex> _filesToIndexes = new HashMap<File, StartAndEndIndex

/** The global model which created/controls this object. */
private final GlobalModel _model;

/** Constructs an empty CompilerErrorModel.
 */
public CompilerErrorModel() {
  _model = new DummyGlobalModel() {
    public OpenDefinitionsDocument getDocumentForFile(File file) {
      throw new IllegalStateException("No documents to get!");
    }
```

```
Welcome to DrJava.  Working directory is /Users/dan/Code/dj
> String hello(String arg) { return "Hello, " + arg; }
> hello("mom")
"Hello, mom"
> import java.util.List
> import java.util.ArrayList
> List l = new ArrayList()
> l.add(23)
true
> l
[23]
> |
```

Figure 37: DrJava screenshot. Source: http://www.drjava.org.

To develop the code shown in the following sections, the book authors chose to make use of DrJava IDE. The reasons are its intuitive and easy use, as well as it lightweight, which makes it possible to use even from a USB stick, without any installation. The software can be downloaded directly from the website http://www.drjava.org.

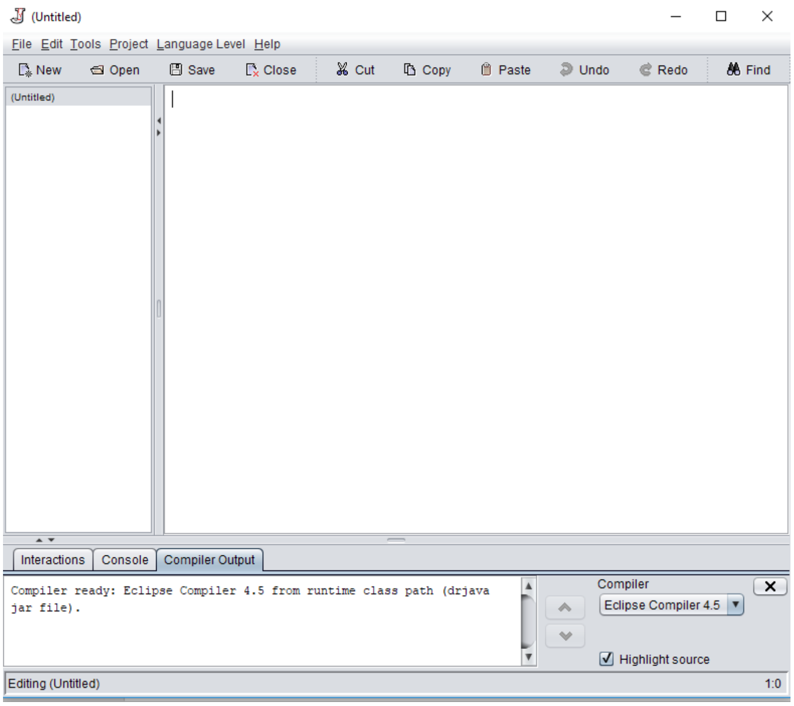
Developing the first program in Java

This sections introduces Java programming by showing a simple and classic example, the Hello World program. The main task of the program is to print a message in the screen (Hello World! Or any other message that the developer desires) and exit the program. Although without being of any practical use, this program helps the reader to grasp some principles in Java programming. The authors used Windows platform and DrJava IDE to develop the code.

To start an empty folder is created, in any place that the reader desires (e.g C:\Java). We create an empty text file inside this folder by navigating to it, right clicking anywhere and choosing the options to create a text file. Initially Windows gives a default name “New Text Document.txt”. It is necessary to change not only the name of the file but also the extension. Because it is being developed a java program, the extension must be .java. So, the file is renamed to "HelloWorld.java". After that, open the file and type the following code:

```
public class HelloWorld {
  public static void main(String[] args) {
    // Prints "Hello, World" in the terminal window.
    System.out.println("Hello, World");
  }
}
```

Save the file and close it. The program Hello World was just written, but in order to run, it is necessary to compile the code- The compilation can be done using the DrJava. First start the IDE by clicking on it wherever it was downloaded. Once the program starts the following window is shown (Windows platform).


(Untitled)
File Edit Tools Project Language Level Help
New Open Save Close Cut Copy Paste Undo Redo Find
(Untitled)
Interactions Console Compiler Output
Compiler
Eclipse Compiler 4.5
Highlight source
Editing (Untitled)
1:0

```
Compiler ready: Eclipse Compiler 4.5 from runtime class path (drjava
jar file).
```

Figure 38: Initial window of DrJava (Windows platform).

To open the HelloWorld.java file in the IDE, use the Open button on the toolbar and search the file, or simply drag and drop it from the folder to the IDE. Once the field is imported, the text code is shown in the main window of DrJava. To compile the code, go to Tools > Compile Current Document. If this is no error, a message appears on the bottom window of the program with the message "Compilation Completed". With this, a new file is created at the same directory of the .java file, with the same name but the .class extension.

To see if the program runs, go to Tools > Run Document. With this, the bottom window shows some messages and the result of the program after the line "> run HelloWorld". It printed the message and exited, as it was expected.

Alternatively, after compiling the .class program can be run in a command prompt, by issuing the command "java HelloWorld" and see the output on the screen.

Java basic language elements (Data types)

A native language element or a data type is a set of values, with determined operations performed on them. In Java, there are primitive types and reference types. The primitive types are summarized in the following table

Type	Set of values	Sample values
int	Integer	1 20 98754
double	Floating-point numbers	2.9191 2.02e45
boolean	Boolean values	true false
char	Characters	'A' '1' '\n'
String	Sequence of characters	"AB" "Hello" "2.98"

And the reference types are:

- Reference to objects;
- Type name is the same as class name;
- Variables hold references to dynamically allocated memory space.

Integer

An integer is any whole number between -2^{31} and $2^{31} – 1$ (−2,147,483,648 to 2,147,483,647).

Floating-point number

The double data type represents floating-point numbers which can be used in scientific applications. This type of data can be represented using a point as decimal separator (i.e 1.22) and accepts scientific notation (i.e 1.2e3 for $1.2*10^3$). A series of different operations with this type of data is native in Java, as illustrated in the following table:

Expression	value
1.27 + 2.1	2.37
1.27 – 2.1	-0.83
1.27 / 2.1	0.60476…
2.1 % 1.27	0.83
1.0 / 0.0	NaN
Math.sqrt(3.0)	1.732….
Math.sqrt (-3.0)	NaN

Boolean values

This data type can assume only two values: true or false. The basic operations with Booleans are shown in the table below.

Operator Name	Operator symbol	Definition
And	&&	a && b is true if both a and b are true.

Or	||	a || b is true if either a or b is true.
Not	!	!a is true if a is false, otherwise is true

Comparison operators are a special type of operators that generates Boolean results by comparing int or double data types. The following table shows some examples of these operators.

Operator	Operator symbol	True	False
Equal	==	2 == 2	2 == 2.1
Not equal	!=	2 != 3	2 != 2
Less than	<	2 < 3	2.1 < 2
Less than or equal	<=	2 <= 2	2 <= 1
Greater than	>	3 > 2	2 > 3
Greater than or equal	>=	3 >= 3	2 >= 3

Strings

A string is a sequence of characters. They may be concatenated using "+" operator.

String str1 = "anystring";

String str2 = "12";
String str3 = str1 + str2;

In a Java program, the basic data types are used to construct objects that can communicate with each other via their methods. There are four main concepts in the object-oriented programming in Java:

- **Object:** Structures with state(s) (variable or not during the program lifetime) and behaviors. For instance, a specific car has a color, number of wheels, power and speed. It has behaviors such as change gear, accelerate/ brake, reserve, stop, turn on or off.
- **Class:** A template to generate objects. The car example mentioned above, as a template for specific car objects, is a class.
- **Methods:** The methods of a class define its behavior. A class may contain a single method or different methods. Every data manipulation, executed action and state change is done through the methods of the object / class.

- **Instance variables:** Objects defined using the classes have a specific set of instance variables.

Java basic methods

A statement is declared using the following rule

<type> identifier;

<type> identifier = <initial value(s)>;

Java is case sensitive. This means that the identifiers x and X have different meaning. The following are some examples of statements:

int x = 24;

int y,z = x, 25;

double[] exarray = {1.1, 2.4, 9};

Point2D p1 = newPoint2D(1.2,5.6);

A class is identified using the keyword class and a name of the class starting with upper case. Conventionally, each first letter of the words inside the class name should also be in upper case.

class MyClassTemplate

The methods defined inside the class start with a lower case and conventionally each first letter of the words inside the methods name is in upper case.

public void myJavaMethod

The name of the program file is exactly the same as the name of the class defined inside it, using the same upper and lower cases as the class name, with the extension .java at the end of the name. For instance, if there is a class called SportCar, then the name of the file containing this class should also be SportCar.java.

Java program processing always starts from the following method:

public static void main(String args[])

The following table gives a list of reserved words in Java syntax. These keywords can not be used to generate variables, constants or any other identifier names.

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

Comments can be used throughout a program to give information regarding the code. The comments are ignored by the compiler, so they are used only for informative purposes. In Java, comments in a single line starts with a "//". In the case that it is necessary to do a multi-line comment, the first line starts with the "/*", and every following line starts with the "*". The last line of the comment ends with the "*/". The following gives example of commenting in Java.

```
// this is a single line comment.
int x = 24;
/* this is second way of writing a single line comment. */
int y = x;
```

```
/* this is
* a comment that
* spans three lines. */
```

Conditional statements and loops

Conditional statements define block of codes that only runs with a certain condition or set of conditions are met.

Loops are special language structures in programming used to repeat a block of code for a certain amount of times, according defined condition(s).

An if statement is used to run a block of code, as long as condition(s) is (are) met. The Java syntax for this type of conditional statement is exemplified below:

```
if ( a > b )
{
        String mystring = "abcd";
        b = a + 1;
}
```

A while block is used to repeat a block of code, while a specified condition is met. The use of this syntax allows the developer to execute a grouped statement as many times as necessary, without rewriting the same code over and over again.

```
int a, b = 10, 1; // initialize the necessary variables
while ( a > b )
/* the loop only starts and repeats as long as
* the conditions are met. */
{
        String mystring = "abcd";
        b = a + 1;
}
```

An important thing to keep in mind is that infinite loops should be avoided. Infinite loops happen when the program enters in the loop block and the program never reaches the condition necessary for it to exit.

The for loop is used to run a specified block of code in a determined amount of times. A variable is used to start the loop, and a loop continuation condition is defined, as well as the rule of incrementing/ decrementing the variable.

```
int n = 10; // initialize necessary variable
int inc = 2;
for (int i = 0; i <= n; i++ )
{
        inc = inc * i;
}
```

Additional constructs and statements can be used in order to elaborate more on the conditional blocks. If it is necessary to exit a loop statement without necessarily reaching its end condition, a break statement can be used.

There are also situations when it may be necessary to jump to the next iterations of the loop. In this case, a continue statement can be used inside a loop, transferring the flow of control directly to the increment statement of the next iteration loop.

In the case that, instead of only one or two conditions are to be considered, there a list of options may be necessary, the switch statement can be used.

The analysis if a condition is met can be calculated before the loop or the conditional statement, using Boolean values. The conditional operator "?": is a ternary operator (three operands) that enables you to embed a conditional within an expression. The three operands are separated by the "?" and ":" symbols. If the first operand (a boolean expression) is true, the result has the value of the second expression; otherwise it has the value of the third expression (Sedgewick & Wayne, 2007).

Object-Oriented characteristics of Java

Inheritance

Inheritance is the way that a class acquires properties and methods from another. It enables the development of programs in a hierarchical order.

The class above, possessing definitions of the inherited properties and methods is called superclass, parent class or base class while the class that acquires such properties and methods is referred to as subclass, derived class or child class. This type of relationship is called is a relation. For example, a Sport Car is a (specialized type of) Vehicle. So, Vehicle would be a superclass defining general methods and properties, while the Sport Car inherits all these properties and may have some specific ones, which does not interfere with the Vehicle class.

In Java, a subclass is defined by using the keyword extends followed by the name of the superclass. Example:

```
    Class MySuperClass {


}
    Class MySubClass extends MySuperClass {
}
```

To clarify, in the following example it is written two types of data, a generic Data and a specialized version of Data, Vectorial Data. The only method that it has is to print the data. The Vectorial Data runs the same method as the superclass, and additional code is added to its method. The superclass Data is defined as:

```
class Data {
  int datetime;
  double value;

  void showdata() {
    System.out.println("Date Time:"+datetime);
    System.out.println("Value:"+value);
  }
}
```

There are two attributes of this class, datetime and value. These attributes are also inherited to the following child class, which overrides the method showdata by adding more code. The overriding characteristics of Java will be described in the next subsection.

```
class VectorialData extends Data {
 String direction;

  void showdata(){
    super.showdata();
    System.out.println("Direction:"+direction);
  }
}
```

In the same file, a class named DataInheritance runs the main program. It generates one instance of the generic data and prints, and another instance of the specialized vectorial data, and prints. The file must be saved with the same name as this main class (DataInheritance).

```
class DataInheritance
{
 public static void main(String[] args) {
   Data data1 = new Data();
   data1.datetime = 20170101;
   data1.value = 1;
   data1.showdata();

   VectorialData vecdata1 = new VectorialData();
   vecdata1.datetime = 20170102;
   vecdata1.value = 1.5;
   vecdata1.direction = "south";
   vecdata1.showdata();
}
}
```

Once the code is compiled and run, the following output is obtained.

```
> run DataInheritance
Date Time:20170101
```

```
Value:1.0
Date Time:20170102
Value:1.5
Direction:south
```

So, the Vectorial Data class inherited from the superclass its attributes, and extends it to have additional attributes and possibly methods. Other classes could also be inherited from the same superclass, without affecting the Vectorial Data subclass. Additionally, subclasses may also be derived from the subclass Vectorial Data, producing even more and more specialized classes, as the following example:

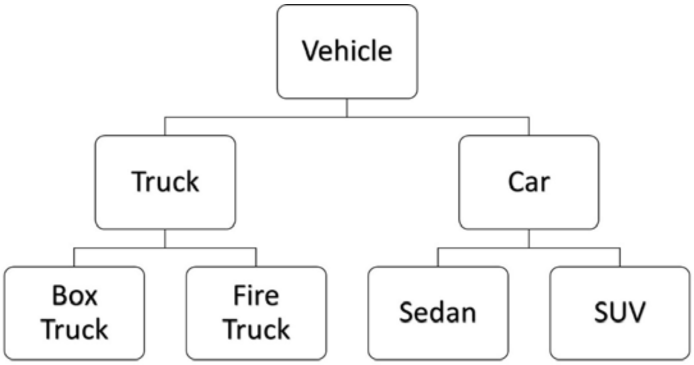


Figure 39: Example of inheritance of classes.

Overriding

Overriding consists on the ability writing over a previously defined method, redefining or extending it.

There are some advantages on overriding, such as to define a behavior according specific characteristic of the subclass, which means that a subclass implements a superclass method according its requirement.

Rules are well established for methods overriding:

- The argument list should be the same as that of the overridden method. Ex: the method run (double km, double velocity) should be overridden with a function that also admits two arguments.
- The type which is returned should be the same or a subtype of

the return type declared in the superclass method.

- Access level can be extended, but not restricted. This means that a superclass method declared as public cannot be overriding by a private or protected method.
- Only inherited methods can be overridden.
- The keyword final avoids a method to be overridden.
- A static method may not be overridden, but can be re-declared.
- Constructors can not be overridden.

Polymorphism

Polymorphism is defined as the capability of an object to assume different forms. A common case of polymorphism in OOP is when a superclass reference is used in order to refer to a subclass object.

Polymorphism is tested by applying more than one IS-A test. If the object can pass this test, then it is said to be polymorphic. All objects in Java are polymorphic since any object will pass the IS-A test. The following is an example of such case:

```
public interface Carnivorous{}
public class Animal{}
public class Lion extends Animal implements Carnivorous{}
```

Lion is considered to be polymorphic, since it has multiple inheritance. The following IS-A tests can be successfully applied to the Lion class:

- Lion IS – A Carnivorous
- Lion IS . A Animal
- Lion IS – A Lion
- Lion IS – A Object

Having passed on these tests, the following statements can be applied to the Lion object reference without errors:

```
Lion l = New Lion();
Animal a = l;
Carnivorous c = l;
Object o = l;
```

The reference variables a, c and o refer to the same Lion l object.

Abstraction

In Object-Oriented programming, abstraction is the capability of hidden from the user the process of implementation, providing only the functionality necessary. In summary, the user has the information on what the object does instead of how it does it. This process is achieved in Java using Abstract classes and interfaces.

Abstract classes can contain abstract methods, but not necessarily. An abstract method is a method without body. Nonetheless, if the class contains at least one abstract method, then it must be declared as an abstract class. Abstract classes can not be instantiated. Once a subclass is developed from an abstract class, it must provide implementation to all the present abstract methods in the superclass. The following is an example of an abstract class.

```
/* File name : Shape.java */
public abstract class Shape {
  private String name;
  int pointx;
  int pointy;

  public Shape(String name, int pointx, int pointy) {
    System.out.println("Constructing a Shape");
    this.name = name;
    this.pointx = pointx;
    this.pointy = pointy;
  }

  public double computeArea();

  public void computePerimeter();

  public String getName() {
    return name;
  }
}
```

Encapsulation

Encapsulation is the ability of packing data (variables) and methods (function) into a single unit. When encapsulated, data from one class is hidden from other classes, and can only be accessed though the methods of their current class. Because of that, encapsulation is also referred to as data hiding.

Encapsulation can be done in Java by declaring variables of a class as private, and providing public setter and getter methods to modify and view the variables values.

Advantages of encapsulation are, among many, the ability of making some specific attributes read-only or write-only and the class can be over total control of what is stored in its fields.

Interfaces

An interface is a similar data type to classes, in the sense that it is a collection of abstract methods. A class implements an interface, inheriting the abstract methods of the interface. An interface may also have constants, default methods, static methods and nested types. Only default and static methods may have bodies. All the others are declared only.

An interface defines the functions or behaviors that a class implements. Nonetheless, writing an interface is analogous to writing classes. Once a class implements an interface, it must also define all its abstract methods, or be also defined as abstract.

The definition of an interface is done inside a .java file, with the name of the file equal to the name of the interface. The compiled code of the interface is generated in .class file. Like classes, interfaces may appear in packages, and their bytecode file must be in a directory structure that has the same name as the package name. Like abstract classes, an interface can not be directly instantiated.

To declare an interface, use the keyword interface, as in the following example:

```
/* File name : myInterface.java */
import java.lang.*;
// Any number of import statements
public interface myInterface {
  // Any number of final, static fields
  // Any number of abstract method declarations\
}
```

Packages

Packages are groups of related types (classes, interfaces, enumerations and annotations) which can provide access protection and namespace management. They are especially useful to prevent naming conflicts, to make searching/locating and to define the use of classes, interfaces, etc simpler.

Examples of existing packages in java that may be mentioned are:

- java.lang – bundles the fundamental classes
- java.io – classes for input, output functions are bundled in this package

When creating a package, the developer must follow some guidelines. At the top of every source file, a package statement must be clearly stated in the first line of the source file, and each source file can contain only one package statement with the classes, interfaces, enumerations, and annotations that are included in the specific package.

To use packages in the program, the file has to be compiled with a directive specifying the package that has to be included. With this, a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

To use classes in defined in different files but still in the same package, the package name must be used in the top of the file. In case one class need to refer to another class in a different package, there are three ways of accomplishing this:

The package keyword followed by a dot and the full name of the class can be used as in:

package.myClassName

Another option is to directly import the class using the keyword import and the wild card (*).

import package.*

A third way is to use the keyword import and the name of the class itself, with the keyword package and a dot preceding it.

import package.myClassName

Once a class is placed on a package, the name of the package becomes part of the name of the class (mypackage.MyClass for example),

Data structures

Data structures are important Java utilities and can be used to perform a variety of operations, especially for scientific computing. Some of them which may be mentioned are:

- Enumeration
- Bitset
- Vector
- Dictionary
- Hashtable
- Properties

Enumeration

Enumeration is an interface which allows organize elements in such a way that methods can be used to retrieve them as a collection of objects. It is a useful type for iterations where it may be necessary to go through each element of an array of objects. The enumeration interface defines two methods:

- Boolean hasMoreElements () – As the name shows, this method is used to check if all elements of the enumeration where already extracted(false), or if there are still elements to be extracted (true).
- Object NextElement () – This method is used to extract the next element in the queue of the enumeration.

BitSet

This class implements a set of bits or flags that can be treated individually (set or deleted). This is particularly useful if the program makes use of a large set of Boolean values and they need to be set and cleared as appropriate.

There are two ways of generating a BitSet, using one of the two constructors mentioned below:

BitSet () – Default BitSize object creation.

BitSize (int size) – An initial size of the object can be defined in the constructor, specifying the number of bits that it can hold. The initial value of the bits is zero.

Vector

The vector is a flexible class which works in the same ways as an array in Java, although it can grow to accommodate more elements. The accessibility of elements in a vector is done through indexing. Vectors are constructed using one of the following directives:

- Vector () – Default constructor, with an initial size of 10 elements.
- Vector (int size) – Optionally the developer can assign the initial size of the vector.
- Vector (int size, int cr) – Besides the initial size, the developer can specify the increment, which defines the number of elements that the vector grows every time it exceeds its maximum capacity.
- Vector (collection C) – This constructor creates a vector and apply each element in the collection c to a position in the Vector instance.

In the following table is listed some of the methods of the vector class.

Method	Return (void me-ans no return)	Description
add (int index, object element)	void	Inserts the element in the position specified.

add (Object element)	boolean	The element is added to the end of the vector.
addAll (Collection C)	boolean	Inserts all the elements of the collection at the end of the vector.
addAll (int index, Collection C)	boolean	Adds all the elemennts of the collection in the specified position.
capacity ()	Int	Returns the size of the vector.
clear ()	void	All the elements of the vector are removed.

Applications in Scientific Computing

Square root Calculator

The following application is a program developed to calculate a reasonable approximation of the square root of any positive number using object-oriented programming in Java. The algorithm used is the Newton-Raphson method for root finding, which reads:

$$x_{i+1} = x_n + \frac{f(x_n)}{f'(x_n)}$$

Where x_{i+1} is the approximation of the root of $f(x_n)$. The function to calculate the square root of any number can be expressed as:

$$f(x_n) = x^2 - S$$

Where S is the number which it is desired to know the root, and x is the root itself. Applying this function in the Newton-Raphson algorithm, one obtains:

$$x_{i+1} = x_n + \frac{f(x_n)}{f'(x_n)} = x_n + \frac{x_n^2 - S}{2x_n} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

This iterative algorithm is used to find an approximation of the root of any number, as long as it is positive.

The main Java program used to solve this problem is divided into 3 classes. The first class prints the values of :the current iteration;

- the current root approximation value;
- the approximation error

The following code illustrates this class, which is here called ScreenOutput:

```
class ScreenOutput {

  void updateScreen(int i, double xi, double x0)
  {
    System.out.print(i);
    System.out.print("   ");
    System.out.printf("%.3f",xi);
    System.out.print("                    ");
    System.out.printf("%.3e",Math.abs(xi - x0/xi));
    System.out.println(" ");
  }

  void printHeader()
  {
    System.out.println("It    x                    Error");
  }
}
```

The class has no data, and two methods. The first one (updateScreen) is used to print in the screen the value of the variables of interest at each iteration step. The second method (printHeader) prints a header which should be placed before starting the iterations.

The second class used in the program calculates at each iteration the root approximation using the Newton-Raphson algorithm mentioned above. The code reads:

```
class SqrtAlgorithm
{
  double x0;
```

```
 double xi;
 double epsilon;

 void updateFcn()
 {
   xi = (x0/xi + xi) / 2.0;
 }
}
```

The class has three properties, or states, which are used to calculate the root approximation at each iteration. The property epsilon is the acceptable error of the calculation.

The third class, which forms the core of the program, accepts two arguments from the user: the number that the user wants to know the root, and the acceptable error. The program than writes a small reader to indicate that it has started and creates instances of the two other classes. These instances are used to do the calculations and the necessary screenoutput. The code of this class reads:

```
public class Sqrt {
   public static void main(String[] args) {
      // read in the command-line argument
      double x0 = Double.parseDouble(args[0]);
      double epsilon = Double.parseDouble(args[1]);
        // repeatedly apply Newton update step until desired precision is
achieved
      System.out.println("==================");
      System.out.println("SQUARE ROOT CALCULATOR");
      System.out.println("==================");
      ScreenOutput scrout1 = new ScreenOutput();
      SqrtAlgorithm algo = new SqrtAlgorithm();
      algo.xi = x0/2;            // estimate of the square root of c
      algo.epsilon = epsilon;       // maximum error
      algo.x0 = x0;
      scrout1.printHeader();
      if ( algo.x0 >= 0 )
      {
       int i = 0;
       scrout1.updateScreen(i, algo.xi, algo.x0);
      while (Math.abs(algo.xi - algo.x0/algo.xi) > algo.epsilon*algo.xi) {
         i++;
         algo.updateFcn();
         scrout1.updateScreen(i, algo.xi, algo.x0);
       }
       // print out the estimate of the square root of c
       System.out.println(algo.xi);
      }
      else
      {
       // print out an error message
       System.out.println(„Error. No root of negative value!“);
      }




  }


}
```

The next step is to test the developed algorithm. A good methodology to start doing is to use values which the analytical square root is already know. In this way, we can measure the analytical error and if the program has any bugs.

We may start by using the program to calculate the square root of the number 9. It is already knowing that:

$$\sqrt{9} = 3$$

It is expected that the program may not fall exactly in the analytical square root of the number, as numerical approximation is used, which admits some error. So, we may start admitting a relatively large error of 0.1. This means that the calculated square root should fall somewhere inside the interval [2.95,3.05]. One the program runs, it generates the following output:

```
> run Sqrt 9 0.1
====================
SQUARE ROOT CALCULATOR
====================
It   x              Error
0   4.500              2.500e+00
1   3.250              4.808e-01
2   3.010              1.920e-02
3.0096153846153846
```

In the program, we fixed the initial guess to be always half of the value inputted by the user, which may be good in some cases, but not so much for others. The program takes 2 iterations to arrive in the value 3.0096, with an error less than 2E-2, or 0.02.

Next, we test the algorithm to find a much better solution using a smaller acceptable error, 1E-15 (the number 1 with 15 zeros to the left side). The output of the program is the following:

```
> run Sqrt 9 1e-15
====================
SQUARE ROOT CALCULATOR
====================
It   x              Error
0   4.500              2.500e+00
1   3.250              4.808e-01
2   3.010              1.920e-02
3   3.000              3.072e-05
4   3.000              7.864e-11
```

```
5   3.000                   0.000e+00
```

3.0This time, as expected, it took some more iterations of the program to arrive in a value inside the admissible interval of error. The program ran through five iteration, until it reaches a value so next to three that the error is almost zero.

We may also test the program with a trickier case. Specially we can try to calculate the root of the number 1 (one), which is equal 1. Using the same small admissible error as the case above (1e-15), the program output is:

```
> run Sqrt 1 1e-15
===================
SQUARE ROOT CALCULATOR
===================
It   x              Error
0   0.500                   1.500e+00
1   1.250                   4.500e-01
2   1.025                   4.939e-02
3   1.000                   6.097e-04
4   1.000                   9.292e-08
5   1.000                   2.220e-15
6   1.000                   0.000e+00
```

1.0In this case, the program starts using 0.5 as the initial guess for the square root of 1. It takes 6 iterations to achieve a value inside the admissible interval. It can be seen that this admissible error is very strict and in more flexible situations the program could even stop in the 3rd or 4th iteration, where it has already been reached a value very near to the real analytical root of the desired number.

Extending the Square root Calculator: Generic Root Finder

In this section, the Square root calculator developed above is extended, so it may find the root of any function. In this first version, two classes must be defined by the user, which are the function which he wants to find the root(s) and the analytical derivate of the function. As arguments to the program, the user provides the initial guess of the root and the admissible error.

The class function has no properties and a single method, which is used to calculate the function at the desired point. To test the developed code, we use the function to calculate the square root of 9, which we can analytically obtain (3). The code of this class reads:

```
class Function {

 double calculate(double x)
 {
  return x*x-9;
 }
}
```

The class dfunction calculates the analytical derivative of the function at the desired point. Again, the class has a single method and no properties. The method calculate is used to find the derivate of the function defined in the class mentioned above. It is important in this case that this derivative is the correct expression according to the function defined above. Otherwise it will not work properly. We use the appropriate derivative expression of the function defined above.

```
class DFunction {

 double calculate(double x)
 {
  return 2*x;
 }
}
```

The class used to generate output to the screen (ScreenOutput) is changed to show the error as the difference between the previous value of the root approximation in the iteration with the current value of it. The code for this class now reads:

```
class ScreenOutput {
 void updateScreen(int i, double xiold, double xinew)
 {
  System.out.print(i);
  System.out.print("   ");
  System.out.printf("%.3f",xinew);
  System.out.print("                    ");
  System.out.printf("%.3e",Math.abs(xiold - xinew));
  System.out.println(" ");
 }

 void printHeader()
 {
  System.out.println("It    x                    Error");
 }
}
```

A class called NewtonRaphson is used to calculate the root approximation at each iteration step. This class receives as properties the initial guess of the root, the function to be evaluated and the derivative of the function. These last two properties are previously defined instances of the classes Function and DFunction mentioned above. The only method of this class is to calculate the newton Raphson algorithm at each iteration step. The code of it reads:

```
class NewtonRaphson
{
 double xi;
 Function function;
 DFunction dfunction;

 void updateFcn()
 {
   double f_x = function.calculate(xi);
   double df_x = dfunction.calculate(xi);
   xi = xi - f_x / df_x;
  }

}
```

The main class is rename to RootFinderv1 (it is the version 1), and the code inside was modified so as to accommodate the changes made in the structure of the program. The code of this part is written below:

```
public class RootFinderv1 {
   public static void main(String[] args) {

      // read in the command-line argument
      double x0 = Double.parseDouble(args[0]);
      double epsilon = Double.parseDouble(args[1]);

      // repeatedly apply Newton update step until desired precision is
achieved
      System.out.println("==================");
      System.out.println("ROOT FINDER CALCULATOR");
      System.out.println("==================");
      ScreenOutput scrout1 = new ScreenOutput();
      Function f1 = new Function();
      DFunction df1 = new DFunction();
      NewtonRaphson algo = new NewtonRaphson();
      algo.xi = x0;           // estimate of the square root of c
      algo.function = f1;
      algo.dfunction = df1;
      scrout1.printHeader();
      int i = 0;
```

```
        double xi = x0*1000;
        scrout1.updateScreen(i, xi, algo.xi);
        while (Math.abs(xi - algo.xi) > epsilon) {
          xi = algo.xi;
          i++;
          algo.updateFcn();
          scrout1.updateScreen(i, xi, algo.xi);
        }
        // print out the estimate of the square root of c
        System.out.println(algo.xi);
    }


}
```

The next step, as it was done in the previous example, is to test the developed code. One point to be mentioned first is to respect with the different roots that a function may have. Suppose a linear function y = 2x+1.The root of this function can be easily checked by rearranging the equation as follows.

$$x = \frac{y-1}{2}$$

To obtain the root of such function, it is only necessary to replace y = 0, in which case we obtain x = -1/2. A plot of this function is shown below, where it can be seen the point that y=0 (in this point x is the root of the equation).

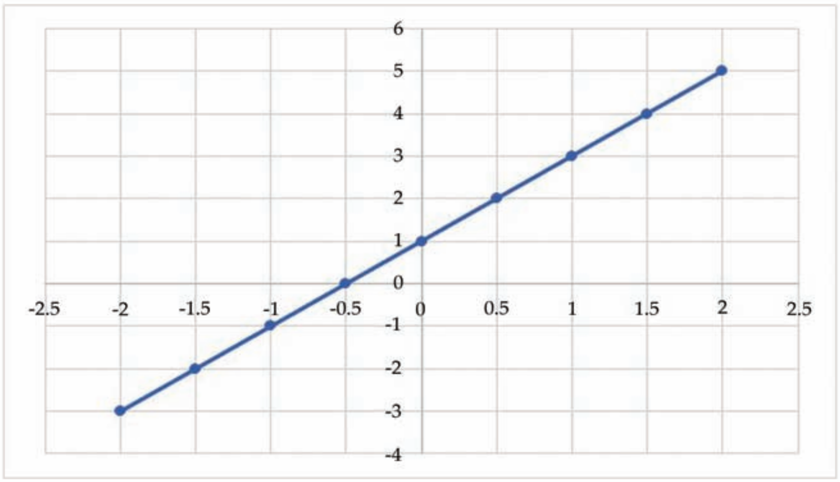


Figure 40: Graph of the function y = 2*x + 1.

From the plot above, it can also be clearly seen that, as the function infinitely increases to the right-hand side or infinitely decreases to the left-hand side, this function has a single root. However, consider now the function used to calculate the square root of number 9. The function is stated below.

$$y = x^2 - 9$$

Again, it is possible to find the root by directly manipulation of the above equation, isolating the x on one side of the equation, as it follows:

$$x = \sqrt{y + 9}$$

By assuming y = 0, one can see that two values can be obtained for x, -3 (3 negative) and +3 (3 positive). A plot in this region of the function is shown below.

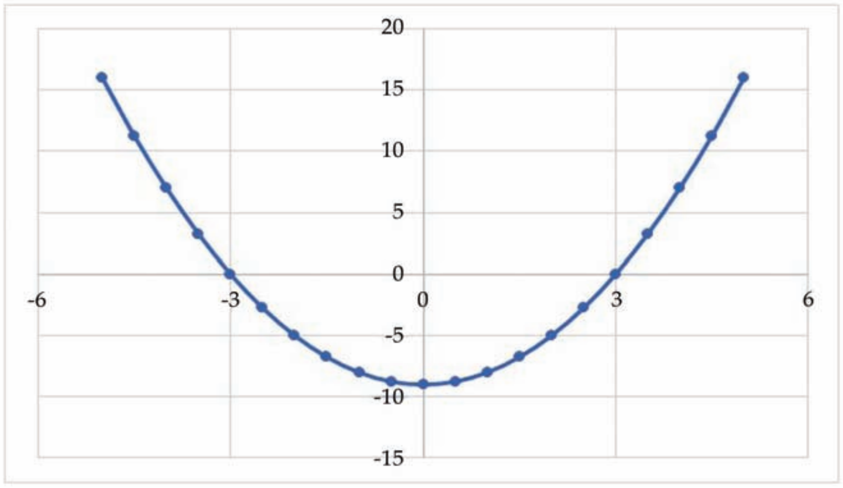


Figure 41: Plot of the function y = x^2-9

In the present section, it is applied a numerical procedure to calculate the root(s) of the provided function. The numerical method being used here is Newton-Raphson. This method is very well known for its efficiency and for its fast convergence. However, the method can only find one root at a time. The root that will be found depends mainly on the initial guess used, which will give the direction of the gradient.

Now we test the RootFinderv1 program to find the root(s) of the equation used to calculate the square root of 9, mentioned before. The following results are obtained if an initial condition of 4.5 is used and an admissible error of 1e-8.

```
> run RootFinderv1 4.5 1e-8
====================
ROOT FINDER CALCULATOR
====================
It   x            Error
0  4.500          4.496e+03
1  3.250          1.250e+00
2  3.010          2.404e-01
3  3.000          9.600e-03
```

```
4  3.000              1.536e-05
5  3.000              3.932e-11
3.0
```

The program took 5 iterations to find the root of the desired function within the admissible interval. The root that was found was the positive one (+3), although we know that there is also another root in (-3). To check that, run the code again using as initial guess (-4.5) and the same error.

```
> run RootFinderv1 -4.5 1e-8
===================
ROOT FINDER CALCULATOR
===================
It   x              Error
0  -4.500             4.496e+03
1  -3.250             1.250e+00
2  -3.010             2.404e-01
3  -3.000             9.600e-03
4  -3.000             1.536e-05
5  -3.000             3.932e-11
-3.0
```

As expected, the program converged to -3, which is another root of the function. The algorithm may be used for a function with different roots, but the root found will depend on how the initial guess directs the gradient during the iteration process. As a last test, we modify the equation to a third order polynomial of the form:

$$y = 4x^3 + 12x^2 - 24x - 32$$

From the function above we can obtain the analytical derivative as follows:

$$y' = 12x^2 + 24x - 24$$

The plot below shows the 3 roots of the function:

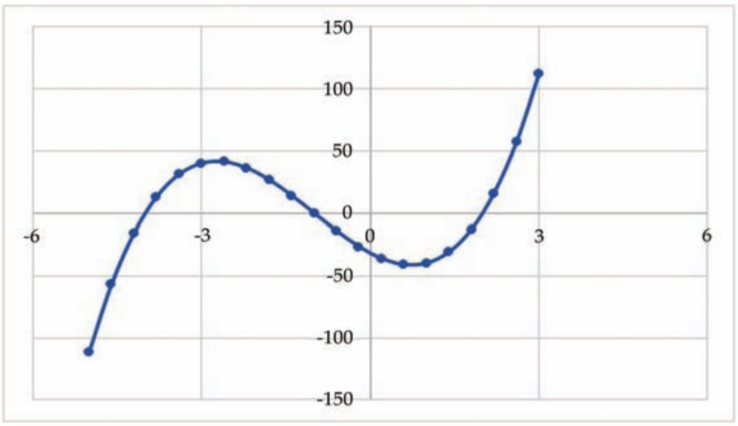


Figure 42: Plot of the function $y = 4x^3 + 12x^2 - 24x - 32$.

Replacing the function and the analytical derivative in the respective classes of the program, and using as initial guess -5 with admissible error of 1e-8, the program produces the following output:

```
> run RootFinderv1 -5 1e-8

====================

ROOT FINDER CALCULATOR

====================

It   x              Error

0  -5.000           4.995e+03

1  -4.282           7.179e-01

2  -4.033           2.494e-01

3  -4.001           3.211e-02

4  -4.000           5.191e-04

5  -4.000           1.348e-07

6  -4.000           9.770e-15

-4.0
```

The program arrives the extreme left root of the equation (-4). This value can be checked by replacing the value in the function, which should produce the value 0 (zero). As a second tentative, use as initial guess the value -2.7 and the same error.

```
> run RootFinderv1 -2.7 1e-8
====================
ROOT FINDER CALCULATOR
====================
It   x              Error
0  -2.700            2.697e+03
1  28.776            3.148e+01
2  18.918            9.858e+00
3  12.380            6.538e+00
4  8.072            4.308e+00
5  5.277            2.795e+00
6  3.529            1.747e+00
7  2.537            9.926e-01
8  2.102            4.350e-01
9  2.005            9.695e-02
10  2.000            4.786e-03
11  2.000            1.146e-05
12  2.000            6.570e-11
2.0
```

An interesting situation happens in this case. Because the derivative in this point is very low, the algorithm jumps on the 1st iteration from -2.7 and 28.7, and then it starts to converge to the extreme right root of the equation, jumping the root which is in the middle of these two roots. We test the program one last time, by using as initial guess -2 and the same error.

```
> run RootFinderv1 -2 1e-8
====================
ROOT FINDER CALCULATOR
====================
It   x              Error
```

```
0   -2.000              1.998e+03
1   -0.667              1.333e+00
2   -1.009              3.419e-01
3   -1.000              8.547e-03
4   -1.000              1.388e-07
5   -1.000              0.000e+00
-1.0
```

As expected, the algorithm converged relatively fast to the root in the middle, taking only 5 iterations to arrive inside the admissible interval.

Extending the Generic Root Finder with Numerical Derivative

In many common applications in scientific computing, one desires to calculate the root of a function, but it may be so complicated that it is unfeasible to obtain an analytical one. In this case, a numerical derivative can simplify the problem, by assuming an approximation as follows:

$$y' = \frac{dy}{dx} \approx \frac{y(x + \Delta x) - y(x)}{\Delta x}$$

Which is referred to as Forward Euler approximation. We can implement this finite derivate in out Root Finder program, in order to extend it to function which an analytical derivative is not directly available. First it is necessary to change the DFunction to use the above equation instead of the analytical one, as it follows:

```
class DFunction {

 double dx = 0.01;
 Function function;

 double calculate(double x)
 {
 return (function.calculate(x+dx)-function.calculate(x))/dx;

 }
}
```

In this case, a property to the DFunction was added, dx, which is the size of the finite step given to calculate the derivative. Ideally this step should be very small, so we fix a value of 0.01. Another added property is the function, so at each time that the derivative is called, it calculates the finite difference using the function defined in an instance of the class Function.

Another minor modification is to add during the program an attribution of the generated instance of the function to the property function of the DFunction object, according:

```
df1.function = f1;
```

Now it is necessary to test this new version, which we call RootFinderv2. We test with the same third-order equation defined in the previous case, an initial guess of -2.7 and an admissible error of 1e-8. The following results are obtained:

```
> run RootFinderv2 -2.7 1e-8
===================
ROOT FINDER CALCULATOR
===================
It  x           Error
0  -2.700        2.697e+03
1  24.570        2.727e+01
2  16.128        8.441e+00
3  10.540        5.588e+00
4  6.874        3.666e+00
5  4.519        2.355e+00
6  3.084        1.435e+00
7  2.322        7.622e-01
8  2.043        2.796e-01
9  2.001        4.161e-02
10  2.000        1.079e-03
```

```
11  2.000             5.953e-06
12  2.000             2.967e-08
13  2.000             1.478e-10
2.00000000000074
```

Although this initial guess is not a very good one, for the very small derivative generates a big jump in the root approximation, the algorithm with finite differences works well, taking 13 iterations to find the root, while the one with the analytical derivative took 12, just one less.

PYTHON

Introduction

Python is a high -level, object-oriented programming language with simple to use syntax. Python is a multi-purpose language, it may be used for Graphical User Interface development, Data Analysis, Web development, Scientific Computing, etc.

As an interpreted language, when one runs a python program, an interpreter will parse the code line by line. This is a major drawback when compared with Java or C++, which are compiled languages. This feature makes Python slightly slow.

Python is also an extensible language, and a variety of packages are available with many functionalities already implemented, so the new developer does not need to start from the scratch when developing a software.

Comprehensive online guides on how to get started in Python can be found, among other sites, in the following addresses:

https://www.programiz.com/python-programming

https://www.python.org/about/gettingstarted/

https://wiki.python.org/moin/BeginnersGuide

http://thepythonguru.com/getting-started-with-python/

Obtaining Python

Python is a free software, so it can be directly obtained in the internet. The official homepage for Python is https://www.python.org/. To a Python distribution, navigate to the

Downloads in the top toolbar, and choose the platform (Windows, Linux, MacOS). A new webpage will be opened and one has only to choose one of the links according the desired release.

In the present book, we focus on the Python 3. However, there are many applications using Python 2 also.

A second option to obtain Python is to download an IDE (Integrated Development Enviroment). This type of software enables the developer to write and to run the code in the same place, making it easier for continuum software development. In this regard, WinPython is a portable free IDE for development in Python on Windows. It is specifically aimed at educational and scientific purposes. It comes with the following pieces of softwares:

- IDLEX (Python GUI): IDLE stands for Integrated Development Environment.It is a simple and suitbale IDE for development in Python, with features such as multi-window, syntax highlighting, integrated debugger with stepping, breakpoints, etc. IdleX is a collection of over twenty extensions and plugins, developed for additional functionality.
- IPython Qt console: It was developed to be a replacement for the standard Python shell, or it can be used as a complete working environment for scientific computing (like Matlab or Mathematica) when paired with the standard Python scientific and numerical tools. Among its features, it may be mentioned dynamic object introspections, numbered input/output prompts, a macro system, session logging, session restoring, among others.
- Jupyter Notebook: An open-source web application that allows the developer to write and share documents that contain live code, equations, visualizations and explanatory text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning, etc.
- Spyder: The name of this software stands for Scientific Python Development Environment. Is an interactive testing, debugging and developing framework with the possibility of using scientific computing tools such as numpy, scipy and

matplotlib.

- WinPython command prompt: A command prompt with integrated python environment.
- Winpython powershell prompt:A PowerShell is a more powerful, feature-rich and more customizable shell than the WInPython command prompt.
- WinPython Control Panel: It can be used to manage installed packages, such as numpy or pandas. And it also allows advanced tasks, such as registering extensions, icons and Windows explorer context menu to a specific Python distribution.
- Winpython Interpreter: Use to assicate extensions to WinPython..
- Qt designer: A tool to develop Graphical User Interfaces using Qt components. The development of the interface can be done in a what-you-see-is-what-you-get (WYSIWYG) manner, and test them using different styles and resolutions.
- Qt linguist: A tool to translate Qt C++ and Qt Quick applications into local languages.

Python primer

To start developing Python code, one can open an empty text file and start writing it. Another option is to start a Python shell and write the code, which is computer interactively. While this second approach is more straight forward for short code and simple calculations, the former one is much more useful for complex code and more "perennial" code.

The first code presented here is a classic example, the Hello World. It simple prints in the screen the phrase "Hello World", or any other phrase that the developer may want, just by changing the words. To do so, one has to create an empty text file, naming it as "HelloWorld.py" (note the extension .py instead or the .txt extension). Once the file is opened, simply type the following code:

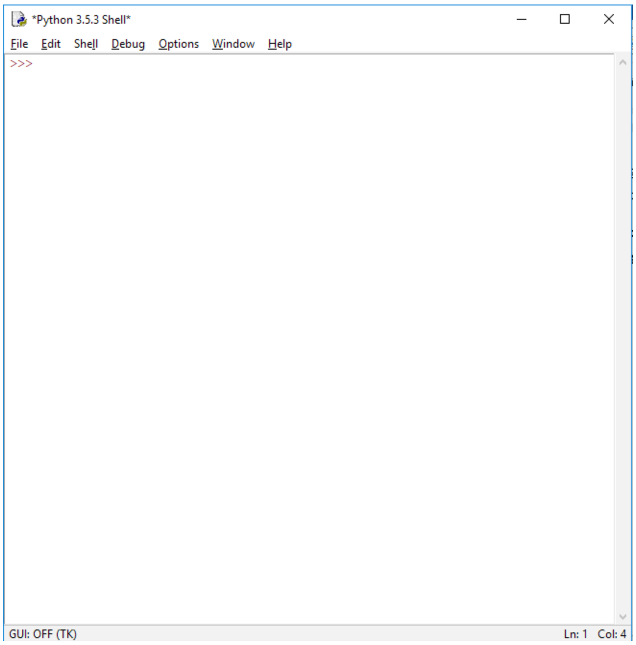
```
print("Hello World")
```

Save and close the file. There are two ways of running this program. One is by calling the command prompt, moving it to the folder where the file HelloWorld.py is saved and typing the following command:

python HelloWorld.py

If the python is correctly installed and everything works well, the window should print the phrase “Hello World” and then return so the user can continue typing.

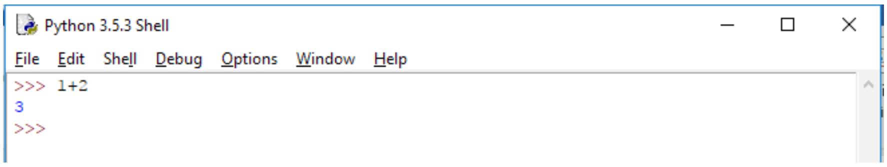
A second option is to run the code in an IDE, for example using the IDLEX in WinPython package. One the IDLEX software is started, the following windows appears in the screen



```
>>>
```

Figure 43: IDLEX window.

Every code typed in the IDLEX windows will appear after the >>> symbol. This IDE can also be used as a calculator. For example, the following calculation can be performed using IDLEX:



```
>>> 1+2
3
>>>
```

Figure 44: IDLEX as a calculator.

In the figure above, the Python language was used to calculate the sum of two numbers in a similar that is done using any calculator.

To run the HelloWorld program using the IDE, navigate to the File > Open… and find the file "HelloWorld.py". It opens in a similar way that a text editor opens, but there are a variety of options in the toolbar, as well as it is noticeable that the code is syntax highlighted. To run this code inside IDLEX environment, simply navigate to Run < Run Module. The result of the program is printed in the IDELX main window.

Syntax basics

Any variable defined in Python store references to objects in the memory. The names attributed to the variables are called identifiers. In python, the following set of rules defines how to proceed in order to give a valid name to a variable:

- The first character of any identifier can be a letter (ex: a or A) or an underscore (_). The use of a number is not valid.
- Identifiers can be generated by a combination of letters, digits and underscores.
- Identifiers can be of any length.
- The following expressions are keywords in python, and therefore can not be used to generate identifiers.

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
pass	else	import	assert	break
except	in	raise		

Values are assigned to variables using the "=" in a similar way that is done in mathematics. For instance, to attribute the value 10 to a identifier x, simply type x = 10. Python is able to dynamically identify the type of a variable. This means that it is not necessary to explicit declare x as a int variable, for the Python can automatically recognize it. In Python each and everything is an object or an instance of a class.

Comments are typed starting with a # symbol. Anything written after such symbol is ignored by the interpreter, and only used for code documentation and sharing.

One line of code can be used to assign multiple variables. For example, the following code statement is valid in Python:

Var1, var2, var3 = val1, val2, val3

- Python has 5 basic data types:
- Numbers
    - o Int: integer values (1, 2, 100)
    - o Float: for floating point values (1.1, 3.14, 100.0)
    - o Complex: complex numbers (1+2j)
- Strings: series of characters
- List: a series of numbers
- Tuple: a fixed series of numbers
- Dictionary: store key value pairs
- Boolean: true or false values

Basic mathematical operations can be performed in Python using the following list of symbols:

Symbol	Definition	Application	Output
+	Addition	3 + 1	4
-	Subtraction	3.0 - 0.1	2.9
*	Multiplication	3.0 * 0.1	0.3
/	Division	3.0 / 0.1	30.0
//	Integer division	1 // 2	0
**	Exponentiation	4 ** 0.5	2.0
%	Reminder	10 % 3	1

Similar mathematical operations can be performed in strings. For example, strings index starts at 0 (zero). To access one element of a string, one can do the following operation:

```
>>> mystring = “Hello World”
>>> mystring[0]
‘H’
```

The + operator is used to concatenate strings as follows:

```
>>> mystring_part1 = “Hello”
>>> mystring_part2 = “World”
>>> mystring_part1 + mystring_part2
‘HelloWorld’
```

The * operator is used to generate a repetition of the string for a defined amount of times, such as in the following example:

```
>>> mystring_part1 * 2
‘HelloHello’
```

If one tries to use the * operator between two strings, then Python generates an error.

The slicing operator [] can be used to retrieve a single character of a string, as shown before, as well as to get part of a string using the following syntax:

```
Mystring [start : end]
```

Which will return the part of the string starting in start and ending in end-1, as in the following example.

```
>>> mystring_part1[0:3]
‘Hel’
>>> mystring_part1[2:]
‘llo’
```

Strings can be compared using the following set of operators. The strings are compared lexicographically, i.e Python compares using ASCII values of the strings.

Symbol	Definition	Application	Output
<	Less than	"John" < "Mary"	True
>	Greater than	"John" > "Mary"	False
<=	Less or equal than	"John" <= "Mary"	True
>=	Greater or equal than	"John" >= "Mary"	False
==	Equal	"John" >= "Mary"	False
!=	Not Equal	"John" >= "Mary"	True

A List in Python can be defined as a collection of numbers which somehow are naturally grouped together. For example, all the measures of mass of different samples may be gathered together in a single list. The use of list implements flexibility since one can work with all numbers at once, or with numbers individually.

A list can be generated by writing the values inside square brackets and separating the numbers by comma.

mylist = [1, 2, 3.2, 700, 9.3]

The single variable mylist refers to a list of five elements. An index associates the position of the elements in the list with its value individually or in a subgroup. Like a string, the first index of a list is 0, and the following elements are monotonically increasing (0, 1, 2, 3, 4, 5, …).

Alternatively, a list can be composed of strings, as in the following case:

mylist = [“string 1”, “string 2”, “string 3”]

Or even a mixture of strings, numbers or any other different types, such as in the following example:

mylist = [“string 1”, 4, [1,2,3]]

There are other ways of creating lists:

mylist = list() # create an empty list

mylist = list([1,2,3]) # create an empty list

Dictionaries are a special type of python data, which associates values with keywords, enabling quick retrieve, addition, removal or modification of the keys. While lists are created with square brackets, the dictionaries are created with curly brackets.

Each item in the dictionary consists of a key, followed by a “:” symbol and the value associated with it. The pairs are separated by comma.

mydictionary = {

’key1’: 1.12,

’key2’: “thing”

```
}
```

The exemplified dictionary above has two keys (key1 and key2) with the attributed values for it (the first one a floating number and the second a string). Optionally an empty dictionary can be created by using the curly brackets without any argument inside it.

```
    My_empty_dictionary = {}
```

The value associated with a key can be retrieved by using key inside square brackets. The following code exemplifies this method.

```
>>> mydictionary = {
        'key1': 1.1,
        'key2': "any text here"
        }
>>> mydictionary['key2']
'any text here'
```

Items can be deleted from a dictionary using the del keyword. In the previously created dictionary, writing "del mydictionary['key1']".

Tuple is a special type of Python list, in which the values inside it can not be modified, deleted or added, replaced or reordered. This means that Tuples are immutable.

A Tuple can be created by inserting values inside a parenthesis ( ). An empty tuple can be generated by opening and closing the parenthesis without any value inside it.

```
>>> mytuple = ( ) # empty tuple
>>> mytuple = (1 ,2 3 ) # tuple with three arguments
```

Some common operations that are performed in lists can also be performed in tuples, such as obtaining the maximum value (max( )), the minimum value (min( )), indexing using slicing operator among others.

Loops

While loop

The while loop is used to repeat a set of statements as long as a condition is true (Langtangen, 2009). In Python, this type of programming block is implemented using the keyword while and indenting the code inside the block, as in the following example:

```
T = 20
dT = -5
while T > 0:
# here the block performs some mathematical operations
T = T + dT # the value of T is updated at each time the
loop runs
```

An important feature in Python language is code indentation. Any code inside a block must be with the same indentation, otherwise the program does not run, or it does not perform what it is expected for. The first statement coinciding with the indentation of the while loop runs only after the loop has completely executed.

For loop

For loops are programming structures similar to the while loop, in the sense that a block of code is repeated until a certain condition is met. But for loops are easier to be used when walking through each element in a list to run the same block of code. For example, suppose it is necessary to print in the screen all the elements of a list of voltage data. The following code can be used:

```
voltage = [1.0, 2.0, 7.0, 12.0]
for V in voltage:
print('The voltage of the element is:',V,'V')
print('------------------')
```

The for V in voltage construct generates a loop over each element in the voltage list. At each time the loop restarts, the variable V refers to an element in the list, starting with voltage[0], voltage[1] and so on. The loop repeats until it the reaches the last element (voltage[n-1] with n being the number of elements in the list).

Branching

Branching or flow control statements are programming structures used to dictate if a block of code should run or not, according one or more conditions that should be met. In many programming languages, as well as in Python, a grammatically similar to human language structure is used, the if.. else block.

If… else blocks are used to test if a condition is true, and in positive case it runs the block of code inside the if part. The else block is used in case the tested conditions is false. For example, consider the following code:

```
voltage = 220.0
if voltage < 220.0:
print('voltage is low')
else:
print('voltage is not low')
```

The code above prints the statement "the voltage is low" if the variable voltage carries a value which is less than 220. On the other side, in case the voltage is anything other than less than 220, it prints the statement "the voltage is not low". Maybe it is high, or maybe it is exactly equal 220. The only thing that the condition block above tests is IF the voltage is less than 220, anything different from that is thrown to the else code block. In summary:

```
if <condition>:
        # block of statements if condition is TRUE
else:
        # block of statements if condition is FALSE
```

Additionally, more conditions can be tested. Suppose in the voltage example above, that we need to test not only if the voltage is less than 220, but also if it is exactly equal to 220. The if else block can be extended with the elif keyword, which means else if, as in the example below.

```
voltage = 220.0
if voltage < 220.0:
        print('voltage is low')
elif voltage == 220.0:
        print('voltage is exactly equal to 220')
```

```
else:
        print('voltage is not low')
```

Optionally the else block can be skipped. In this case, if no condition is met, then the program jumps to the code after the if else block. If the if else block is very simple, it can be written in a condensed form using a single line, as in the following case:

```
# <code to run> if <condition> else # <code to run>
```

File handling

File handling refers to the technique of operating over files, opening, reading and/ or writing and closing afterwards. The syntax for opening a file is:

```
fileidentifier = open(filename, mode)
```

the fileidentifier is a file handler or file pointer generated using open with filename (a string with the path of the file) and the mode used to open the file. After operations on a file, it is important to close the file. The syntax for doing so is:

```
fileidentifier.close() # fileidentifier is the file pointer
```

The different modes that a file can be opened are summarized in the following table.

Mode	Syntax	Description
Read-only	"r"	Open a file for read only
Write-only	"w"	Open a file for writing. All the data in the file is cleared after the file is opened. If the file does not exist, then it is created.
Append	"a"	Write data at the end of the file.
Binay Write	"wb"	Open a file to write in binary mode.
Binary Read	"rb"	Open a file to read in binary mode,

Functions

A function in computer programming can be defined as a set tasks performed according some inputs, and generating some output. In Python, functions are defined using the keyword def. The keyword return is used to express which value(s) is(are) returned as output after the code runs. For example, suppose the following equation

$$s = s_0 + v_o t + \frac{at^2}{2}$$

Which is the equation used to calculate the space where an object moving uniformly accelerated is located. A function can be defined where the user provides the initial position (s_0), the initial velocity (v_0), the constant acceleration (a) and it gets as result the actual position (s) as well as the actual velocity ($v = v_o + at$), defined as:

$$v = v_o + at$$

In Python a function to obtain this values can be written as:

```
def func_s(s0,v0,a,t):
    i = 1
    v = list([v0])
    s = list([s0])
    while i<t+1:
        v.append(v0 + a*i)
        s.append(s0 + v0*t + (a*i**2)/2)
        i += 1
    return s,v
```

The first line defines the name of the function (func_s) followed by the arguments that the function receives as inputs: the initial position (s0), the initial velocity (v0), the constant acceleration and the time of simulation.

The second line initializes the variable which will be used to iterate and calculate the position and the velocity at each time step. Following the list of velocities (v) and position (s) is also initialized with the initial position and velocities.

The while loop is used to obtain the value of position and velocity at each time step. The last line of the function with the keyword return says which variables will be returned by the function, which are the lists of velocities and positions (s and v).

To use the function is necessary to give the necessary inputs and to use the outputs somehow. A complete program which calculated the positions and velocities and print them in the screen is given below.

```
print('-------------')
print('Movement Calculator')
print('-------------')
def func_s(s0,v0,a,t):
    i = 1
    v = list([v0])
    s = list([s0])
    while i<t+1:
        v.append(v0 + a*i)
        s.append(s0 + v0*t + (a*i**2)/2)
        i += 1
    return s,v
s0 = 0.0
v0 = 0.0
a = 2
t = 10
s,v = func_s(s0,v0,a,t)
print('Position: ',s)
print('Velocity: ',v)
```

Once this code (named as calculatespace.py) runs, the following output is obtained:

```
calculatespace.py
-------------
    Movement Calculator
-------------
Position:  [0.0, 1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0]
Velocity:  [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0]
```

The function used to calculate the position and velocity are useful for reuse, in the sense that different parameters (input arguments) can be given to the function to analyses different outputs obtained through it. The following code exemplifies the evaluation of this code for two different accelerations.

```
print('-------------')
print('Movement Calculator')
print('-------------')
def func_s(s0,v0,a,t):
   i = 1
   v = list([v0])
   s = list([s0])
   while i<t+1:
      v.append(v0 + a*i)
      s.append(s0 + v0*t + (a*i**2)/2)
      i += 1
   return s,v

s0 = 0.0
v0 = 0.0
a = [2.0, 1.0]
t = 10.0

s,v = func_s(s0,v0,a[0],t)
print('Acceleration: ',a[0])
print('Position: ',s)
print('Velocity: ',v)
s,v = func_s(s0,v0,a[1],t)
print('Acceleration: ',a[1])
print('Position: ',s)
print('Velocity: ',v)
```

Note that, by transforming the acceleration variable, from a single value to a list of values (2 and 1) it was only necessary to add one line of code to test the function with the new conditions (s,v=func_s(s0,v0,a[1],t)) and the following three lines are screen output. When this code is run, one obtains the following output:

```
calculatespace.py
-------------
Movement Calculator
-------------
Acceleration:  2.0
Position:  [0.0, 1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0]
Velocity:  [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0]
Acceleration:  1.0
Position:  [0.0, 0.5, 2.0, 4.5, 8.0, 12.5, 18.0, 24.5, 32.0, 40.5, 50.0]
Velocity:  [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

This type of problem exemplifies the flexibility and extensibility of the functions in Python, especially for cases where it is reused with difference input arguments.

A convention in Python is to develop documentation for user-defined function. This is done by adding comments to the begging of functions after its definition. The documentation string, known as a doc string, should contain a short description of the purpose of the function and explain what the different arguments and return values are. Interactive sessions from a Python shell are also common to illustrate how the code is used. Doc strings are usually enclosed in triple double quotes “””, which allow the string to span several lines (Langtangen, 2011).

```
def func_s(s0,v0,a,t):
   “””
   Function calculates space and position
   Of an object in uniform accelerated motion
   s0 - Initial position
   v0 - Initial velocity
   a - constant acceleration
   t - time span
```

```
return:
s - velocity vector
v - velocity vector
“””

i = 1
v = list([v0])
s = list([s0])
while i<t+1:
    v.append(v0 + a*i)
    s.append(s0 + v0*t + (a*i**2)/2)
    i += 1
return s,v
```

Classes and objects

As already mentioned in the previous chapter, a class packs a set of data with a collection of methods or function which operates in this data. In this way, data and methods are grouped to achieve more modularity and because of that, a better organization in complex and continuously growing projects.

In many cases, the use of classes is not mandatory. However, the implementation of such a technique allows more elegant code development and the fact that the code becomes modular makes it easier to grow at a later stage.

In scientific computing, the most common application of classes is to represent function, incorporating its parameters as part of the class members and the equation itself as one or more methods inside the class. To exemplify one may consider the equation of uniformly accelerated motion used in the previous section. In this case, the initial position and velocity as well as the acceleration may be considered parameters of the class.

In Python, a class usually has a constructor, which is a special function that runs when an instance of the object is created. The syntax of a constructor is given in the following form:

__init__

The class for the uniformly accelerated motion can be represented using UML notation, as shown below:

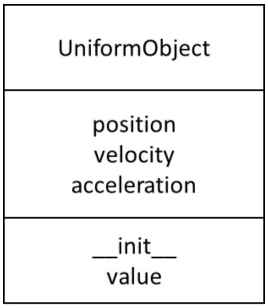


Figure 45: A Class for the uniformly accelerated motion.

In the above representation, we chose to call the developed class as UniformObject, for it represents a generic object that behaves according the uniform motion equation described above. This class has as attributes its position (which will be the initial position when the equation is calculated), its current velocity (which will be the initial velocity when the equation is calculated). The __init__ method is the constructor method, and the value method is used to obtain the new position of the instance once the equation is evaluated.

The following code shows an initial implementation of the class Uniform Object in Python:

```
class UniformObject:
   def __init__(self,s0,v0,a):
      self.position = s0
      self.velocity = v0
      self.acceleration = a
   def value(self,t):
      newposition = self.position+self.velocity*t+self.
acceleration*t**2/2
      newvelocity = self.velocity*t+self.acceleration*t
      return newposition, newvelocity
   def formula(self):
    print('s0 + v0*t + a*t**2/2')
```

As already mentioned, the __init__ function is runs when the user/developer tries to create an instance of the UniformObject class. It attributes the data to the class members accordingly. The value () method is used to calculate the evolution of space and velocity of the object under the designated law of motion. This function takes as argument the time t of simulation in order to obtain the new position and velocity of the object. At least the formula () method just prints the formula used in the class. Changing the formula in this last method will not change how the motion is calculated, and it is just used as an informative piece of data.

The following code exemplifies the application of the UniformObject class to calculate the positions and velocities of an object under the law of motion with different times of simulation.

```
s0 = 0
v0 = 0
a = 2
uobject = UniformObject(0,0,2)
print("object started at position ", uobject.position, "m, velocity ",
   uobject.velocity,"m/s and acceleration ", uobject.acceleration, "m2/s.")
t = 10
s,v = uobject.value(t)
print("at time ", t, "s, object position is", s,"m, velocity ",
     v,"m/s and acceleration ", uobject.acceleration, "m2/s.")
uobject.formula()
t = 20
s,v = uobject.value(t)
print("at time ", t, "s, object position is", s,"m, velocity ",
     v,"m/s and acceleration ", uobject.acceleration, "m2/s.")
uobject.formula()
```

The code mentioned above generates the following output:

```
unformobject.py
object started at position  0 m, velocity  0 m/s and acceleration  2 m2/s.
```

```
at time  10 s, object position is 100.0 m, velocity  20 m/s and acceleration 2 m2/s.
s0 + v0*t + a*t**2/2
at time  20 s, object position is 400.0 m, velocity  40 m/s and acceleration 2 m2/s.
s0 + v0*t + a*t**2/2
```

The correctness of the algorithm can be tested by manual calculation or using a spreadsheet.

Now we would like to have a more generalized object, which does not only perform uniformly accelerated movement, but actually any type of motion (accelerated, deaccelerated, constant velocity). To do this, it is necessary to remember the following concepts of motion:

$$a=\frac{d^2s}{dt^2}=\frac{dv}{dt}$$

$$v=\frac{ds}{dt}$$

Where a is the acceleration (m²/s) , s is the velocity (m/s) and s is the position (m). We may apply Backward Euler time integration to solve this equation and find the velocity, the acceleration and the position at each instant of time. The Backward Euler algorithm reads:

$$\frac{dy}{dt}=f\left(y,t\right)$$

$$\frac{y^{t+\Delta t}-y^t}{\Delta t}=f\left(y^{t+\Delta t},t+\Delta t\right)$$

$$y^{t+\Delta t}=\ y^t+\Delta t*f(y^{t+1},t+\Delta t)$$

Replacing the Backward Euler algorithm in the laws of motion, one obtains:

$$a^{t+\Delta t}=\frac{v^{t+\Delta t}-v^t}{\Delta t}$$

$$v^{t+\Delta t}=\ v^t+\Delta t*a^{t+\Delta t}$$

$$v^{t+\Delta t} = \frac{s^{t+\Delta t} - s^{t}}{\Delta t}$$

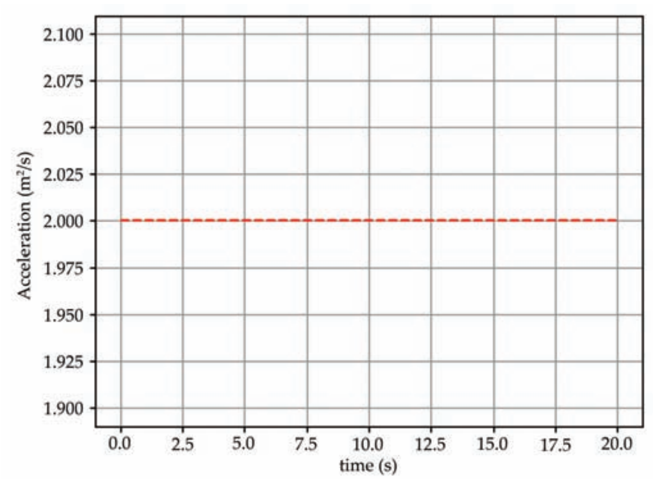
$$s^{t+\Delta t} = s^{t} + \Delta t * v^{t+\Delta t}$$

So at each time step of simulation the new value of position and velocity can be obtained with the instaneous acceleration is defined. The accuracy of the algorithm depends on the time step (Δt) used. The object-oriented approach will help to evaluate the accuracy of different time steps. The problem is defined by assuming a relation of acceleration with time and assuming the initial conditions:

$$s(t=0)=0$$

$$\frac{ds}{dt}(t=0)=v(t=0)=0$$

The following figure illustrates the relation acceleration x time:



Assuming a constant acceleration different from zero, the motion is said to be uniformly accelerated. In this case the analytical solution is well known and was already shown above. A class was developed incorporating the derivative equations of motion. The code is shown below.

```
import numpy as np
class MovingObject:
    def __init__(self, s0, v0, t, a):
        self.position = [s0]
        self.velocity = [v0]
        self.velocitytime = []
        self.accelerationtime = t
        self.acceleration = a

    def dds(self, t, x):
        return np.interp(t, self.accelerationtime, self.acceleration)

    def ds(self, t, x):
        return np.interp(t, self.velocitytime, self.velocity)

    def get_initvelocity(self):
        return self.velocity[0]

    def get_initposition(self):
        return self.position[0]
```

First, it is necessary to import a package very useful for numerical computing, the numpy. This package contains a variety of useful methods to be used in scientific computing. In the present case, it is used the interpolation method from numpy.

The class Moving Object has four class members: the position vector, the velocity vector, a vector of the time that the velocity data is recorded, a vector of the time that the acceleration data is recorded and the acceleration vector.

The class defines four methods. The first method (dds) as the name suggests, calculates the second derivative of the position, which is the acceleration value. This value is obtained by interpolating the table of time x acceleration stored in the instance of the class. Here the package numpy is used to perform that calculation.

The second method (ds) is used to calculate the first derivative of the position, which is the value of the velocity at each time step. An important feature to be clarified is that the first derivative in this case can only be calculated once the second derivative is considered. Again, the value of the velocity is obtained by interpolating in the table of time x velocity stored in the instance of the class.

The third method is used to retrieve the initial condition on the velocity. Although this method is optional, it keeps the program more clear and elegant. The same principle applies to the last method, which is used to get the initial position of the instance of the class.

To integrate the derivative equations, as already mentioned, we implement Backward Euler algorithm. In an object-oriented approach, a class can be implemented to perform this calculation. The following code refers to the developed class.

```
class BackwardEuler:
    def __init__(self, f, t0, N, dt):
        self.f = f
        self.t0 = t0
        self.N = N
        self.dt = dt
        self.x = []
        self.t = []
    def integrate(self, x0):
        x = x0
        t = t0
        self.record(x0, t0)
        while t < N*dt:
            t = t + dt
            dx = self.f(t, x)
            x = x + dt * dx
            self.record(x, t)
    def record(self, x, t):
        self.x.append(x)
        self.t.append(t)
```

The class Backward Euler has six class members. The first member (f) refers to the derivative function to be integrated. Namely:

$$\frac{dy}{dt} = f\left(t, y\right)$$

The second member stored the initial time used to perform the integration. This time must be the same as the time that the initial conditions of velocity and position were obtained. The third member (N) is the number of steps of integration. The fourth member defines the time step of integration, or the difference in time between two subsequent steps.

The fifth and sixth members store the calculate state and time after the solver calculates it. This data can be retrieved to be analyzed, plotted and saved.

Two methods, besides the constructor (__init__) are implemented in the Backward Euler class. The first method, integrate is the core of the solver and it performs the time integration according the algorithm. At each iteration of the while loop, the new state is calculated and stored in the instance of the class by calling the record () method.

In order to be able to test the accuracy of the method and of different time steps, a simple function is implemented to calculate the analytical values of position and velocity at each time. The code of the function is the following:

```
def func_s(s0, v0, t0, a, N, dt):
    s = [s0]
    v = [s0]
    t = [t0]
    while t[-1] < N * dt:
        t.append(t[-1] + dt)
        s.append(s0 + v0 * t[-1] + a * t[-1] ** 2 / 2)
        v.append(v0 + a * t[-1])
    return s, v, t
```

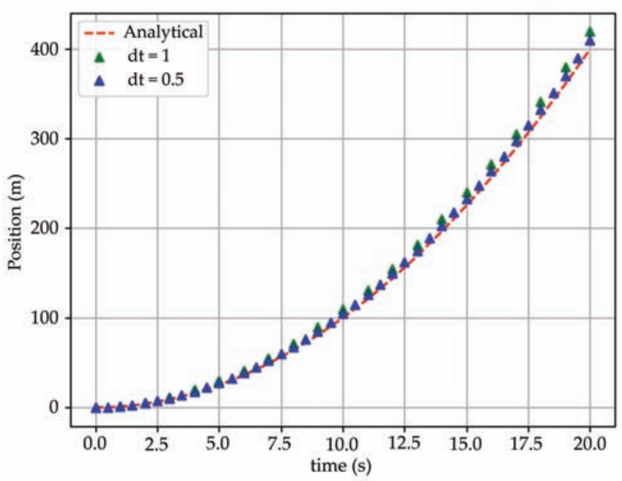
After the classes and the previous function are defined, the following program can be written to calculate the position of an object using the integration algorithm and comparing two different time steps: 1 second and 0.5 seconds.

```
import matplotlib.pyplot as plt

s0 = 0.0
v0 = 0.0
a = [2.0, 2.0]
t = [0.0, 20.0]
uobject = MovingObject(s0, v0, t, a)

t0 = 0.0
N = 20.0
dt = 1.0
analy_s, analy_v, analy_t = func_s(s0, v0, t0, a[0], N, dt)

solver1 = BackwardEuler(uobject.dds, t0, N, dt)
solver1.integrate( uobject.get_initvelocity() )
uobject.velocity = solver1.x
uobject.velocitytime = solver1.t

solver2 = BackwardEuler(uobject.ds, t0, N, dt)
solver2.integrate( uobject.get_initposition() )

s0 = 0.0
v0 = 0.0
a = [2.0, 2.0]
t = [0.0, 20.0]
uobject = MovingObject(s0, v0, t, a)
```

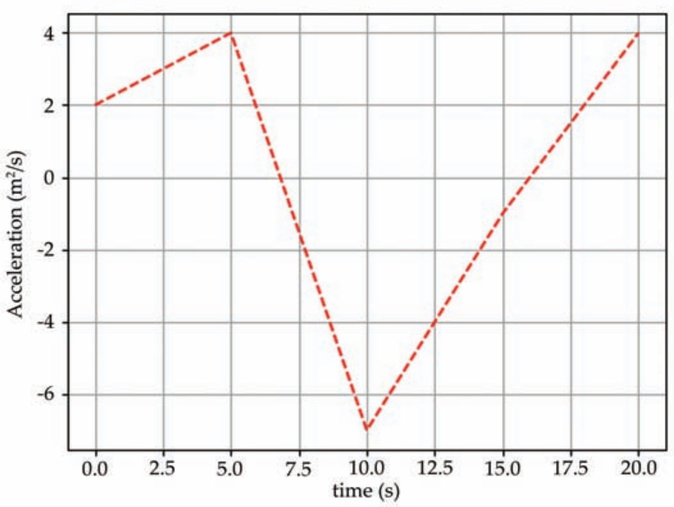
```
t0 = 0.0
N = 40.0
dt = 0.5
solver3 = BackwardEuler(uobject.dds, t0, N, dt)
solver3.integrate( uobject.get_initvelocity() )
uobject.velocity = solver1.x
uobject.velocitytime = solver1.t

solver4 = BackwardEuler(uobject.ds, t0, N, dt)
solver4.integrate( uobject.get_initposition() )

handles = plt.plot(t, a, ‘r--’)

plt.xlabel( ‘time (s)’ )
plt.ylabel( ‘Acceleration (m²/s)’ )
plt.grid( True )
plt.savefig( “Acceleration.png” )
plt.show( )

handles = plt.plot(analy_t, analy_s, ‘r--’, solver2.t, solver2.x , ‘g^’,
solver4.t, solver4.x , ‘b^’)

plt.xlabel( ‘time (s)’ )
plt.ylabel( ‘Position (m)’ )
plt.legend( handles, [‘Analytical’,’dt = 1’,’dt = 0.5’])
plt.grid( True )
plt.savefig( “PositionBEdt1.png” )
plt.show()
```

In the first line of the code, the package matplotlib is imported to be used to generate graphical output of the results. Documentation of this package can be found in the web address: https://matplotlib.org/.

The following figure illustrates the graphical output of the program above, where it is compared the results of the position using the analytical solution, a backward euler algorithm with time step of 1 second, and the backward euler algorithm with 0.5 second.



As expected, the use of a smaller time step generates better results (with less error in comparison with the analytical solution) than the bigger time steps. To show the capability of generalization of the algorithm, it can be used to calculate the position of the moving object using any function of the acceleration with time. Let's assume the following acceleration x time profile.

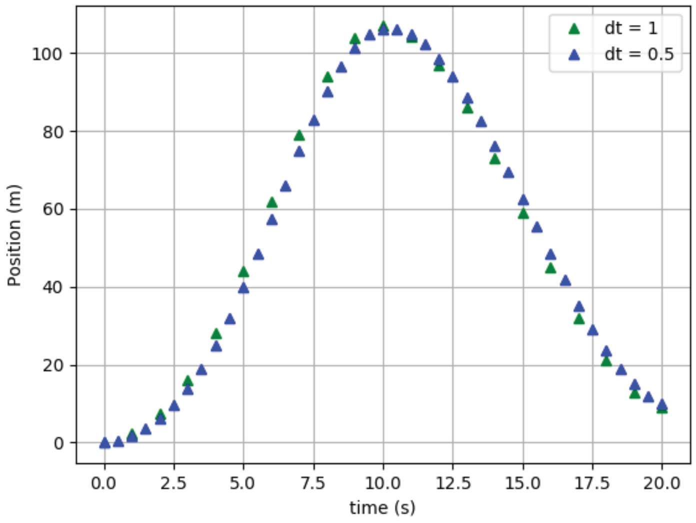


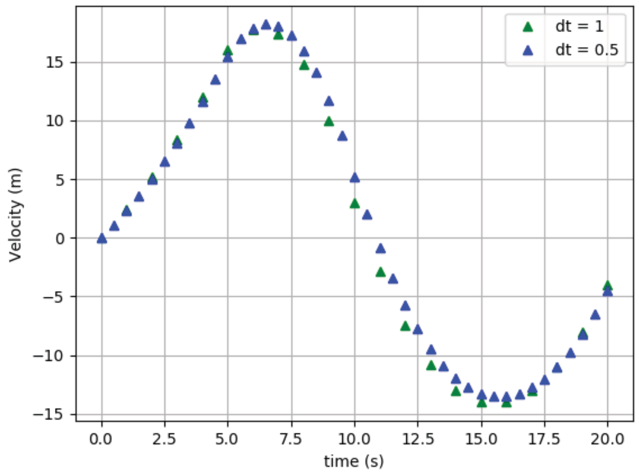
In the acceleration profile above, it can be seen that the object has an increasing acceleration in the time interval of 0 to 5 seconds. Then it starts to deaccelerate until it reaches an acceleration of -7 m²/s, which means that the object reduces significative its velocity, possibly even reaching negative velocity (the object goes backward). From 10 seconds up to 20 seconds it accelerates again, until it reaches the acceleration of 4m²/s. In Python, the acceleration and time lists were defined as:

a = [2.0, 4.0, -7.0, -1.0, 4.0]

t = [0.0, 5.0, 10.0, 15.0, 20.0]

By using the same program that was developed before (just changing the acceleration and time lists as defined above), the obtained result for the position and for the velocity are shown below:





As it was already expected by looking in the acceleration profile, the object does not only move forward, but approximately at 10 seconds, an inflection in the position and the negative values of the velocity profile shows that the object moves backward, and it continues to do so until the end of its movement.

The above example shows a simple but elegant and useful implementation of object-oriented programming in Python to solve scientific computing problems. In the following section we go a bit further in this subject.

Class Hierarcy – Extending a class

On this section, it is show how Python can be used to generate classes derived from more generic ones, and how this functionality can spare time and make complex programs simpler to be written

To start with an example, consider the following class for first order polynomials, or straight lines, according the following equation template:

$$y = c_0 + c_1 x$$

Where c_1 and c_1 are parameters of the equation, i.e class data members, and x is an independent variable. Naturally, y is the dependent variable. The class developed in Python for this type of polynomial could be written as follows:

```
import numpy as np
class Line:
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1

    def __call__(self, x):
        y = self.c0 + self.c1*x
        return y
```

The class defines two methods. The first one (__init__) is the constructor method and initializes the values of the parameters of the line. The second method (__call__) calculates the value of dependent variable given a dependent variable value.

One important thing to mention at this point is the special method __call__. This method is used to refer to the object in a special way. By implementing it, an object instance can perform a task using the following syntax:

```
Myobject( )
```

As an example, the usage of the line class can be done as follows:

```
c0 = 1
c1 = 2
myline = Line(c0, c1)

x = 1
y = myline(x)
print("x = ",x," y = ", y)
```

Suppose now that it is desired to extend the functionality of the Line class, by making it able to calculate a parabola function (a second order polynomial). In this case there are basically three possibilities:

- To write a whole new function from scratch: For a simple case such as the present one, this solution may not be the worst and it is relatively easy. Nonetheless, the repetition of the same code that was already written in the Line class is not recommended and should be avoided.

- To rewrite the Line class: This is also a simple and easy task to implement. However, the code that may have already been written using the Line class may have to be rewritten in order to work with the new version of the class, and this can be a very hard task. So, this procedure should also be avoided.
- To write a new class that incorporates the features already implemented in the Line class and brings the necessary new features to perform the desired calculations. This type of programming is referred to as inheritance, and is the best procedure to be followed in problems such as these.

In Python, the nomenclature used to derive a child class from a parent class follows the specification below:

class ChildClass (ParentClass)

So, in this specific case of the child Parabola class, the definition of it will be written as follows:

class Parabola (Line)

Which means that the new class, Parabola, is a child class from Line and it inherits its class members and functions, invisibly. Naturally, the class Parabola will not be an exact copy of the Line class, but it should extend the constructor by incorporating extra class members, and the __ call__ method is also changed by using the parabola equation as follows:

$$y = c_0 + c_1 x + c_2 x^2$$

In the equation, it is clear that the class members c_0 and c_1 are inherited from the Line class, while the parameter c_2 will be added to this new class. In order to avoid repeating the same code of the parent class, the following syntax should be used to call the methods of the parent class:

ParentClass.methodname(self, arg1, arg2, …)

The following code shows how the Parabola class is defined, and the repetition of already implemented code in the Parent class is avoided by using the syntax shown above.

```
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        Line.__init__(self, c0, c1)
        self.c2 = c2

    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2
```

In the constructor method, the constructor of the parent class is called so as to handle the two class data members that are equal on the Line class and the Parabola class. In this way, rewritten the same code is avoided. The same principle is applied in the __call__ method, where the same function is called from the parent class and it is added the additional value so as to obtain the results of a parabola equation, and not a line.

Another approach to solve the same problem would be, instead of extending the functionality of a Line, to restrict the Parabola. If we think of a Line in the point of view of a Parabola equation, the line equation can be written as:

$$y = c_0 + c_1 x + 0x^2$$

So, it can be said that a line is a parabola with the parameter c_2 set to 0 (zero). In this sense, the Parabola class would be written as:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0 = c0
        self.c1 = c1
        self.c2 = c2

    def __call__(self, x):
        y = self.c0 + self.c1*x + self.c2*x**2
        return y
```

And the Line class would be a child class from the Parabola, according the following code:

```
class Line(Parabola):
```

```
def __init__(self, c0, c1):
    Parabola.__init__(self, c0, c1, 0)
```

Here, only the construction need to be overridden, since it is explicitly defined that the last parameter of the Parabola must be set to 0 (zero). The __call__ is just the same, so there is no need to rewrite it.

The concept of the approach to be used (from Line to Parabola or from Parabola to Line) can be extended to any problem in general, using a perspective of inheriting class from a simpler to a more complex one, or vice versa. In general, it is natural to think that new classes will inherit from simpler, more generic class. However, not necessarily the generic classes will be simpler, and the way that a problem will be solved depends on how the problem is seen by the developer. There is no right or wrong.

Application in Scientific Computing

Bahn et al. (2002) developed an object-oriented scripting interface to a mature density functional theory code. The advantage of using an object-oriented approach is highlighted by the authors, in the sense that there was no need to rewrite the underlying number-crunching code. The paper shows in detail the advantages and disadvantages of the homogeneous interface.

Adams et al. (2002) developed a software package called PHENIX, meaning Python -based Hierarchical ENvironment for Integrated Xtallography. The software is used for crystallographic macromolecular structure determination. According the authors, the developed software will be able to provide algorithms to proceed from reduced intensity data to a refined molecular model and making easier to define structure solution for both the novice and expert crystallographer.

In 2007, Pierce created the PsychoPy, a platform-independent experimental control system written in the Python interpreted language using entirely free libraries. The author mentions that the motivation to develop such software is the fact that computer display technology is a major contributor to the studies in visual processing. The software package provides tools that allows a variety of different exercises, from stimulus presentation and response collection from a big range of devices, to simple data analysis such as psychometric function fitting.

In the field of Bioinformatics, Sukumaran & Holder (2010) contributed by creating a Python library for phylogenetic computing, called DendroPy. The software provides object-oriented reading, writing, simulation and manipulation of phylogenetic data, with an emphasis on phylogenetic tree operations. Special features are used in order to perform efficient calculation of tree distances, similarities and shape under various metrics. The framework supports a variety of phylogenetic data formats (NEXUS, Newick, PHYLIP, FASTA, NeXML, etc.).

One work already mentioned in the UML chapter, Perez et al. (2012) developed pyOpt, a Python framework focused on non-linear constrained optimization. A distinction is maintained between the problem and the solver, which provides high flexibility to the framework. Different optimization algorithms are implemented in pyOpt and are accessible in the common interface. The authors demonstrate the applicability of the developed framework by solving a variety of problems with different levels of complexity.

The object-oriented programming has been shown useful for macromolecular simulation and design through the implementation of Rosetta3 by Leaver-Fay et al. (2014). Rosetta3 is a molecular modeling program, freely available for academic use. Its architecture enables the rapid prototyping of novel protocols by providing easy to use interfaces to powerful tools for molecular modeling.

MODELICA

Modelica is a programming language focused on the development and simulation of mathematical models of complex nature or man-made systems. It is an object-oriented and equation based programming language. According Fritszon (2003), the four main characteristics of Modelica are:

The flow of data is acausal, since the language is equation-based instead of statement based. This feature enables the reuse of classes and more adaptability of one model for different contexts.

Its features englobes physical objects from a diversity of domains: electrical, mechanical, chemical, biological and mathematical application are only some of the domains which can be studied using Modelica.

The language possesses a generic unified class component

It is strongly based on component modelling, with constructs for creating and connecting components, making the development of complex systems easier than other programming languages.

The history of its development started in 1996 with the PhD thesis of Hilding Elmqvist. He and a group of programmers started to work together in the area of object-oriented modelling technology and applications. The initial goal was to write a paper on the existing technologies on object-oriented programming, including an investigation on the possibility of unifying existing modeling languages, as part of the ESPRIT project Simulation in Europe Basic Research Working Group (SiE – WG).

In a short period of time after the work has started, the focus of those involved in the project shifted from simple providing a revision on the state-of-the-art, to start to develop a novel unified modeling language based on the whole experience of tool designers, application experts and computer scientists. The design started from scratch, and a new name was given to the language: Modelica.

The group founded the Technical Committee 1 inside EuroSim. Not much later, in February 2000, the Modelica Association was estabilished as a non-profit organization with international projection and focused on promoting and maintain the development and propagation of the Modelica Language and Modelica Standard Libraries.

Obtaining a Modelica IDE

In order to use Modelica features, one needs to obtain a software for developing and simulating, which is able to read the Modelica code. The following list is a compilation of some commercial and free Modelica IDEs (Integrated Development Environment) available:

Table 2 – List of comercial IDEs for Modelica.

Software	Provider	Type	Description
Simplorer	ANSYS	Commercial	multi-disciplinary system modeling and simulation solution.

Dymola	Dassault Systèmes	Commercial	Modelica translator which is able to perform all necessary symbolic transformations for large systems.
SimulationX	ESI ITI GmbH	Commercial	Software with graphically-interactive features for modeling, simulation and analysis of multi-domain systems from 1D to 3D.
MapleSim	Maplesoft	Commercial	high-performance multi-domain modeling and simulation tool.
Wolfram System Modeler	Wolfram	Commercial	high-fidelity modeling environment that uses versatile symbolic components and computation to drive design efficiency and innovation

Table 3 – List of free IDEs for Modelica.

Software	Provider	Type	Description
JModelica.org	---	Free	extensible Modelica-based open source platform for optimization, simulation and analysis of complex dynamic systems.
Modeliac	----	Free	compiler for a subset of the Modelica language including parts of the "equation" subset that can express relations between Real variables.
OpenModelica	----	Free	complete Modelica modeling, compilation and simulation environment based on free software.

Basic concepts of Modelica

Every system developed in Modelica is based on classes, also referred to as models. Once a class is defined, it is possible to any number of instances of the class, the objects. The classes define the blueprints of the objects, which are to be "produced" through Modelica compiler and run-time system.

The Class is divided into components. The most relevant of them are the variable declarations and the equations sections. To illustrate, the following model is used to generate our first system, which behaves according the Linear ODE:

$$\frac{dx}{dt} = ax + b,\ x(0) = 1.0$$

The variable x is also called a state of the system. The time derivative ($\frac{dx}{dt}$) is represented in Modelica through the method der( ). The above system can be written in Modelica according the following code:

```
class LTISystem
 Real x(start = 1);
 parameter Real a = 3;
 parameter Real b = 1;
equation
der(x) = a*x+b;
end LTISystem;
```

In OpenModelica, this model can be generated by going to File > New Modelica class and typing the above code. After this, one can go to Simulation > Simulation Setup and configure the start time and stop time, the solver and other options of the simulation. To exemplify, the above model was simulated from 0 to 2 using euler solver. The result is shown below:

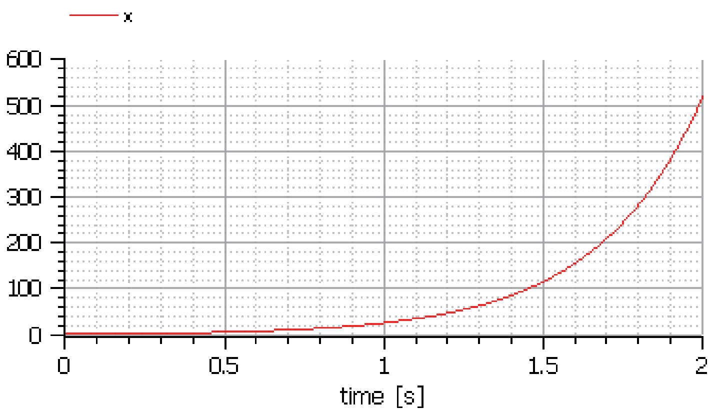


Figure 46: Result for the simulation of the simple LTISystem model.

The simulation can also be performed by opening the Command Prompt Compiler and writing the following directive:

simulate(LTISystem,stopTime = 2)

Looking again the definitions of the model LTISystem, it can be subdivided into two main parts: the first part declares the variables and parameters of the system. The keyword Parameter defines which variables are parameters, and omitting this word creates variables which are not view as parameters by the model. For instance, the variable x is not set as a Parameter. A initial value can be given to variables though it is optional. Not setting the initial value sets the variable to 0 at the beginning of the simulation.

The second block in the LTISystem model consists of the equations or the definitions on how the system behaves. As already mentioned, the declaration of derivatives of variables is done using the der( ) expression.

Comments can be given in the model to clarify the meaning of each block or line of code. They also make the job easier of a third party upgrading or working in the model, since descriptive text gives explanation of what means the components of the program.

A comment can be written inside double quotes (“ a comment “). Usually this type of comment is used in the same line as the model definition and variable declaration, documenting the program. They are reffered to as definition comments, for they are not completely ignored by

Modelica. In fact, these comments are used by Modelica programming environment to appear in menus or help texts for the user.

Another way of inserting comments is by using double slashes (/ /), and every text appearing after this symbol in a line is completely ignored by the compiler and serves only as documentation for the programmer. A third mode, which allows comments to span over many lines is by starting it with /* and ending it with */.

The last way of commenting is using the block Annotation. This block helps to create structured documentation for the model. It also provides graphical features for the model, i.e, Modelica is able to generate a representation of the model according definitions using the annotation. For example, the following code:

```
annotation(
   Diagram(graphics =
   {Polygon(origin = {-2, 4}, points = {{-44, 36}, {44, 36}, {44, -36}, {-44, -36},
{-44, 36}}),
      Text(origin = {-10, 31}, extent = {{20, -50}, {-2, 3}}, textString =
"LTI System", fontSize = 50)}, coordinateSystem(initialScale = 0.1)));
```

Generates a graphical output of a rectangle representing the system with the text LTI System as the figure below.

Figure 47: Graphical representation of the LTI System using annotation.

Modelica object-oriented approach

In common object-oriented softwares, such as Java, Phyton, C++ and many others, object-oriented programming supports operations on stored data, which can be variables or objects. On the other side, Modelica emphasizes structured mathematical modeling. In this sense, a class is a collection of mathematical descriptions of the model, simplifying further analysis. This is called declarative programming.

In essence, the way that object-oriented programming is seen in Modelica can be summarized as (Fritszon, 2003):

- The concepts of the model are structured using object-orientation, which emphasizes the declarative structure of the mathematical equations. The three main concepts that gives foundation to this structure is the development of hierarchies, component-connections and inheritance.
- The dynamical properties of the model are expressed in a declarative way using equations
- The object then consists into an instance containing a set of shared data.

The structure provided by Modelica to develop declarative statements avoid the necessity of the user to be rewriting the way information or data flows in the model and how to simulate it. Al these steps are taken care by the Modelica compiler.

Acausal Physical Modeling

Acausal physical modeling is a very unique and special feature of Modelica. To better explain it, consider a linear equation of two variables, x and y, as stated below:

$$y = 2 * x + 3$$

One can easily deduce that, from the way that the equation is written, values for y can be obtained by inputting values of x in the equation. However, how this equation can be used if the problem is to obtain values of x once values of y are available? Normally two procedures can be taken in most of the programming languages. The first one, which may not be used for some non-linear systems, is to rearrange the equation, isolating the x on one side and shifting everything to the other, as follows:

$$x = \frac{y-3}{2}$$

A second option, called implicit approach, is to insert a dummy variable (F) and shift all the terms of the equation to one side, letting the dummy variable isolated as follows:

$$F = -y + 2 * x + 3$$

And F must be zero. To solve this, there are different iterative algorithms that one can provide an initial guess, and the algorithm converges to find the root of the equation, i.e the values for x and y which satisfies the constraint F = 0.

However, in Modelica an equation is declared in acausal model, which means that it does not matter which variables are input or output on it, the declaration is the same. That means for the example mentioned above, the equation with the y on the left side can be used to find values of x given some values of y. The casuality of the equation is unspecified before solving the system. It just becomes causal once it is set the system of equation to be solved.

According Fritszon (2003), the main advantage if this type of modelling is that the solution direction is dependent on the direction of the flow of data, defined by the inputs and outputs of the system. The data flow context is defined by stating which variables are input and which ones are output.

In this sense, a system can be solved in any direction. For example, suppose a physical system of tank being filled and discharging at the same time. The normal procedure is to find the tank discharge once the inlet is defined as well as the geometries of the tank. In Modelica, the problem can be totally inversed without rewriting the model, i.e to find the inlet of the system once the outlet is defined.

Components, Connections and Connectors

There are three main characteristics that forms a Modelica complete model:

- Components
- Connection mechanism
- Component framework

Components are connected through connection mechanism. These network of connected components with each other forms the connection diagrams. The component framework works as a driver, ensuring that communication works and constraints are satisfied along the connection network.

The component is a single Modelica class, with well-defined interfaces, also called ports or connectors, used to communicate, send and receive data from the outside world. The component should be defined outside the world, so specific features of the simulation world are separated from the definitions of the system itself, for reusability. A component can also be composed by other components in a hierarchical structure.

Connections diagrams in Modelica are used to represent graphically the interaction between components in a system. These connections represent real physical dimensions, for instance electrical wires, pipes with fluids, heat exchange between the components, etc. The components are represented by, for instance, rectangles and connectors are represented by small square dots on the extremes of it, denoting input/output ports.

The connectors are instances of the connector class of Modelica. This class defines the variables that are transferred from linked components. A simple example is given below:

```
connector Pipe
Pressure P;
flow Discharge Q;
end Pipe;
```

The Pipe connector contains two variables, Pressure and Discharge, which is specially designated as a flow variable. The flow keyword defines the type of coupling: which can be:

- For non-flow variables, equality coupling, according to Kirchhoff's first law;
- For flow variables, sum-to-zero coupling, according to Kirchhoff's current law;

For the example above, it means that connecting two components will define that the Pressure is the same between the two ports, and the discharge that leaves one components enters to the other component (negative and positive discharge summing to zero).

SECTION 4 GENERAL APPLICATIONS IN SCIENTIFIC PROBLEMS

AN OBJECT-ORIENTED APPROACH FOR FUNCTION DIFFERENTIATION IN PYTHON

The following application is a modification of the example "Class Hierarchy for Numerical Differentiation" developed by Langtangen (2016).

The purpose of this chapter is to develop a simple program able to differentiate any function using numerical techniques, and when available, compare the numerical output with the analytical one. The desired program interface enables the user to type as arguments of the main program:

- The expression to be evaluated;
- The method of differentiating that should be used
- A value for the independent variable(s)
- Optionally, the analytical difference value for comparison purpose

In summary, the user will provide in a command-line the following directives for instance:

numdiff.py ‘x**2’ Forward1 3 6

The program than provides the numerical approximation of the difference. In the case above, it is known that the analytical difference is given by:

$$\frac{d\left(x^2\right)}{dx} = 2x \rightarrow x = 3 \rightarrow 2*3 = 6$$

There are different ways of numerically obtaining this derivative. The following are some of these methods:

1st – order forward difference

$$\frac{dy}{dx} = \frac{y(x+h) - y(x)}{h} + \sigma(h)$$

2nd – order forward difference

$$\frac{dy}{dx} = \frac{y(x) - y(x-h)}{h} + \sigma(h)$$

2nd order central difference

$$\frac{dy}{dx} = \frac{y(x+h) - y(x-h)}{2h} + \sigma\left(h^2\right)$$

4th order central difference

$$\frac{dy}{dx} = \frac{4}{3}\frac{y(x+h) - y(x-h)}{2h} - \frac{1}{3}\frac{y(x+2h) - y(x-2h)}{4h} + \sigma\left(h^4\right)$$

Where $\sigma\left(h^n\right)$ holds the error of truncation. Because all of the methods possess properties in common, it is natural to develop a superclass which represents a generic differentiation method. Subclasses derived from it will perform the differentiation according the desired algorithm. The following code represents the generic parent class:

```
class Diff:
    def __init__(self, f, h=1E-5, dfdx_exact = None):
        self.f = f
        self.h = float(h)
        self.exact = dfdx_exact
    def trunerror(self, x):
        if self.exact is not None:
            return self.exact(x) - self(x)
```

The superclass Diff has three class members, the function to be differentiated (f), the finite step size (h), and a value of the analytical difference that may be provided by the user, so one may be able to compare the accuracy of the chosen method or of the chosen step size.

From this generic superclass, subclasses can be derived, according each of the methods implemented in the program. The following is the code for the implemented differentiation methods, each one defined as a class.

```
class Forward1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h

class Backward1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x) - f(x-h))/h

class Central2(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/(2*h)

class Central4(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (4./3)*(f(x+h) - f(x-h)) /(2*h) - \
        (1./3)*(f(x+2*h) - f(x-2*h))/(4*h)
```

The code above is saved in file Diff.py. In order, not to mix the differentiation classes which can be used in different programs with the simple command line program here presented, the statements of the program are defined in a different file. Python can import these classes in the command line program using the statement:

```
from Diff import *
```

However, Python can only import the file if it finds it on the Python Path. To include the folder that the file Diff.py was created in Python Path, use the following directive:

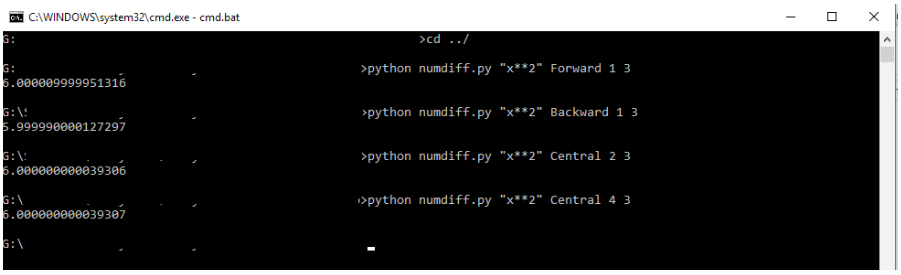
```
import sys
sys.path.append("directory/where/file/is/saved")
```

In this way, Python adds to its path the directory and it is able to find the module that should be imported.

The complete code for the command line program to calculate the numerical difference is as follows:

```
import sys
from Diff import *
from math import *
from Equation import Expression
formula = sys.argv[1]
f = Expression(formula,["x"])
difftype = sys.argv[2]
difforder = sys.argv[3]
classname = difftype + difforder
df = eval(classname + '(f)')
x = float(sys.argv[4])
print(df(x))
```

When the user runs this program, it prints the value of the numerical difference according the desired numerical method that the user required. The following screenshot shows the testing of this program.



```
C:\WINDOWS\system32\cmd.exe - cmd.bat
G:[redacted]>cd ../

G:[redacted]>python numdiff.py "x**2" Forward 1 3
6.000009999951316

G:\[redacted]>python numdiff.py "x**2" Backward 1 3
5.999990000127297

G:\[redacted]>python numdiff.py "x**2" Central 2 3
6.000000000039306

G:\[redacted]>python numdiff.py "x**2" Central 4 3
6.000000000039307

G:\[redacted]
```

Figure 1: Testing the numdiff program

DEVELOPMENT OF A SIMPLE OBJECT ORIENTED SIMULATOR OF DYNAMICAL SYTEMS

In this chapter, it explained and implemented step by step a simple object-oriented simulator. A simulation software is a program capable of modelling some phenomenon described by one or more mathematical functions. In essence, the simulator allows the user to observe a process without performing it.

There are simulation software's in different areas, such as analysis of power systems, behavior, weather conditions, electronic circuits, chemical reactions and processes, biological and environmental processes, feedback control systems, among a variety of other areas.

Besides simulating well-based mathematical functions, simulation is also used to test new theories, and these results can be validated with observed data of the event under analysis.

Simulations falls into two main branches: continuous simulation and discrete simulation. Discrete simulations are used to describe events that can be represented statistical events such as the arrival of clients in a bank queue. Continuous simulations are used in a variety of physical processes since these phenomena are non-discrete.

To start with a simple example, with develop a simulator algorithm capable of modelling the following discrete equation:

$$x^t = ax^{t-1}$$

From the model equation, one can see that the variable x is a function of time, and it is discrete since values from it are inferred at a fixed sample rate of 1 (one). The variable a is a constant, and is referred to as a parameter of the model. The parameter $\frac{x^t}{x^{t-1}}=a$ can also be seen as a ratio between the subsequent values of $\frac{x^t}{x^{t-1}}=a$:

$$\frac{x^t}{x^{t-1}} = a$$

Before making any calculations, one can infer some possibilities regarding the dynamics of the model, according the value of a:

If $a < 0$, the value of $-\infty$ decreases indefinitely towards $-\infty$

If $a = -1$, the value of x oscillates between $+x$ and $-x$

If $-1 < a < 0$, the value of x converges to 0 (zero)

If $a = 0$, the value of next x after the beginning of the simulation will be always 0 (zero).

If $0 < a < 1$, the value of x decreases and converges to 0 (zero)

If $a = 1$, the value of $a > 1$ is always constant, does not increase or decrease

If $a > 1$, the value of $+\infty$ is increases indefinitely towards $+\infty$

So even a simple equation such as the one mentioned above can have a variety of behavior. So much more are complex systems which depends on a series of parameters, conditions and different equations depending on a series of factors.

Before simulating the model, it is important to know that a simulation has three important events, or methods:

- Initialization: The initial values for all states of the system need to be configures at this phase.
- Observe: There are different ways of monitoring the states of a system. One way is by collecting it in a matrix, by printing them in the screen or performing any visualization of the system.
- Update: In this moment, the states of the system are updated as new time step is given. This part is defined as a function and it runs repeatedly.

Before starting to perform any calculation and any implementation of a specific code, we can develop a template of the main simulator class, which will perform each of the three steps of the simulation process

```
class Simulator:
    def __init__(self, f, x, t):
        self.f = f
        self.x = [x]
        self.t = [t]
    def observe(self, x, t):
        self.x.append ( x )
        self.t.append ( t )

    def update(self, N):
        N = 1
        # code goes here
```

The Simulator class is a generic simulator which we will extend to be able to simulate the discrete system described above. To do so, we rewrite the update function, which will at each time step, calculate the new state value and observe it, until it reaches the number of steps required by the user. The definition of this class is shown below:

```
class DiscreteSimulator(Simulator):
    def update(self, N):
        x = self.x[-1]
        t = self.t[-1]
        for i in range(N):
            x = self.f(x,t)
            t = t + 1
            self.observe(x,t)
```

The update of the states of the system is done by calculating the provided function that defines the dynamics of the function under study. In the present case, we can define this function as growth function as follows:

```
def growth(x,t):
    return 0.2*x
```

Here we use the parameter a equal to 0.2, which means that the value of x will converge to zero given enough simulation time. The initial value of the state x, naturally, must be different from 0, otherwise nothing will happen. The simulation can be performed and graphical visualization can be generated with a few lines of code, as follows:

```
import matplotlib.pyplot as plt

x0 = 10.0

t0 = 0.0

discrsim = DiscreteSimulator(growth,x0,t0)

discrsim.update(30)
plt.plot(discrsim.t,discrsim.x)

plt.show()

plt.savefig('example.png')
```

Here we define the initial state with a value of 10 and the initial simulation time equal 0 (zero). In the following line, an instance of the Discrete Simulator class is created with the function and the initial states. The function update is used to evaluate the function in the given time steps (30 time steps). Graphical visualization is easily generated with the matplolib library, and the result is shown in the figure below.

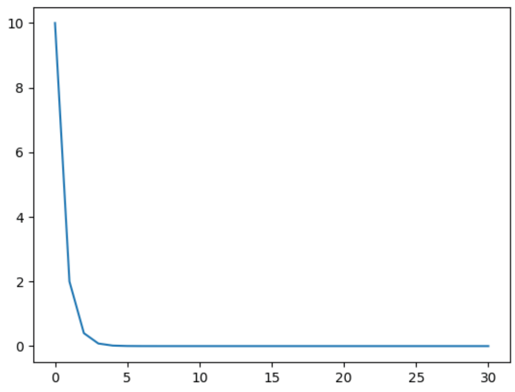


Figure 2: Result for the simulation of na discrete model.

To allow code extensibility, it is useful in Python to subdivide the code in Modules, at each according its functionality. In the present case, we may divide the code in the following components:

- One file with the definitions of the Simulator class and its subclasses
- One file with the definition of the model which is being tested
- One file with the specific simulation conditions (initial states, number of steps)
- One file to generate instances of the classes involved in the problem and to perform the simulation itself.

This type of code division allows modularity, and allows the user as well as the developer to focus on different parts of the problem when looking different files, as well as not to do modifications on files in the final version.

Now it is desired to extend the simulator, to be able to deal with models with multiple variables. To do so, we assume the following discrete model:

$$x_t = 0.5x_{t-1} + y_{t-1}$$

$$y_t = -0.5x_{t-1} + y_{t-1}$$

$$x_0 = 1, y_0 = 1$$

To be able to implement additional states, we make use of numpy library to use arrays to store the states. The observation will now be stored in a new class which allows the Simulator class to be responsible only for carrying out the simulation, and at each simulation time calling the observer which will perform the recording of the simulation data. The classes are stored in a file by the name Simulator.py

```
import numpy as np
class Simulator:
  def __init__(self,f,x,t):
    self.f = f
    self.x = x
    self.t = t
    def update(self,N):
        N = 1
        # code goes here

class Observer:
    def __init__(self,x,t,N):
        self.x = np.zeros((len(x),N+1))
        self.t = np.zeros(N+1)

    def __call__(self,x,t,i):
        self.x[:,i] = x
        self.t[i] = t

class DiscreteSimulator(Simulator):
    def update(self,N):
        self.simobs = Observer(self.x,self.t,N)
        x = self.x
        t = self.t
        self.simobs(x,t,0)
        for i in range(1,N+1):
            x = self.f(x,t)
            t = t + 1
            self.simobs(x,t,i)
```

By using numpy arrays, the Observer class is able to store any number of states. The model definition is written in a file ModelConfig.py, as follows.

```
def model(x,t):
    x1 = 0.5*x[0] + x[1]
    x2  = -0.5*x[0] + x[1]
    return x1,x2
```

The initial conditions and the number of steps in the simulation are now defined also in a separate file, named InitConfig.py- The data in this file is as follows:

```
t = 0.0
x = [1.0,1.0]
N = 30
```

In this file, it can be seen that it was chosen not to use two different variables to solve the problem as the set of equations suggested, but instead we want to keep the same number of arguments in the model definition, so the program will be able to run for one or for multiple states problem.

The last file, used to create the instance of the classes to simulate and to perform the simulation itself is written in the file workflow.py. The code for this file is shown below.

```
import matplotlib.pyplot as plt
from InitConfig import *
from ModelConfig import *
from Simulator import *
discrsim = DiscreteSimulator(model,x,t)
discrsim.update(N)
t = discrsim.simobs.t
x = discrsim.simobs.x[0,:]
y = discrsim.simobs.x[1,:]
plt.plot(t,x,t,y)
plt.savefig(‘example.png’)
plt.show()
```

Now it is necessary to import the other program modules in order to run it properly. This is done using the directive import in the top of the code. Visual output is generated using the Matplotlib, and shown below.

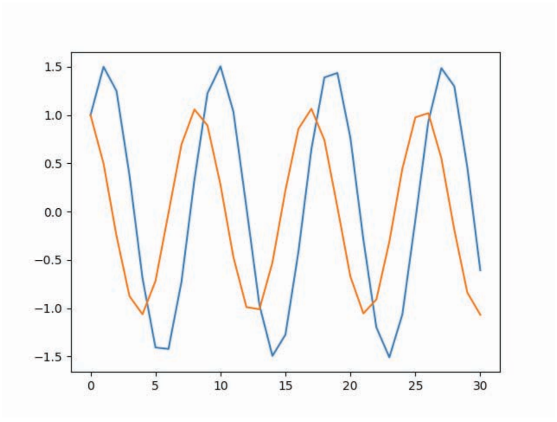
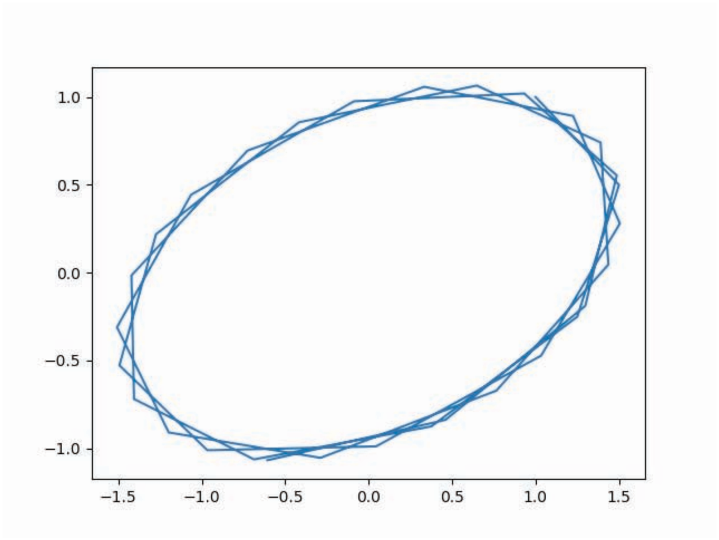


Figure 3: Results for the dicrete model with 2 states.

The program can be evaluated for different parameters of this model, and the user will see that only certain kinds of behaviors are possible in this system. The model may show exponential decay or growth or oscillatory behavior. The linearity of this type of model can be checked by performing a plot of one state against the other, what is called x-y phase space.



From the plot above, it can be seen that the system is in an oval, period system. This type of dynamics is typical of linear systems.

Application of the object-oriented simulator to Predator-Prey model

The predator-prey model is a famous mathematical description of ecological interaction between two species, also known as Lotka – Volterra equations. Basically, the model assumes that the dynamics of the predator-prey relationship can be modelled by the following assumptions:

- Prey grows if there are no predators
- Predators decay if there are no preys

The scenario implemented by the model can be summarized as follows: Suppose a closed ecological system, where no migration occurs out or into the system. This ecosystem is composed by only 2 typed of animals: the predator and the prey. In this simple configuration, the food chain can be represented as below:

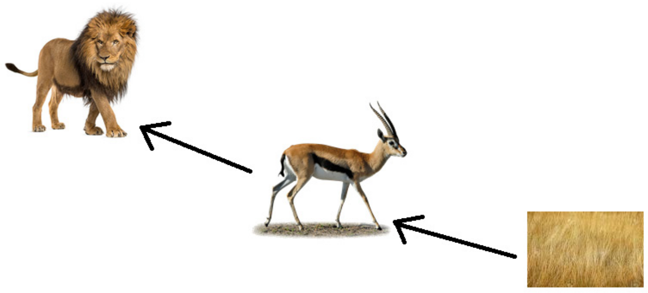


Figure 4: Ecological dynamics of the predator prey model.

The prey has an infinite amount of available food to eat. The interactions between predator and prey can be defined according the following events:

- The prey's death rate increases as the predator population increases.
- The predators' growth rate increases as the prey population increases

With this assumption, the size of the predator and the prey populations can be described by a system of 2 nonlinear differential equations.

$$x' = Ax - Bxy$$

$$y' = -Cy + Dxy$$

Where x' is the rate of change in the prey population size, x is the prey population size.. y is the predator rate of change in the population size, and y is the predator population size.. The parameters A, B, C and D describes the interactions between the two species.

A short history on the Lotka-Volterra model

In 1910, Alfred J. Lotka proposed the predator-prey model as a theory for autocatalytic chemical reactions. In 1920, the model was extended by Andrey Kolmogorov to represent the interactions among an herbivorous animal and a plant specie. The set of equations was published in 1926 by Vito Volterra, inspired by the marine biologist Umberto D'Ancona who became his son-in-law.

The model was later extended in order to include prey-growth, and it has been commonly applied to evaluate the dynamics of some species on different parts of the globe.

Implementation of the model in the object-oriented Simulator

To be able to implement the model, it first needs to be transformed from continuous to discrete mode. The discretization can be done using Forward Euler time integration, in which case the set of equations become:

$$x_{t+1} = x_t + Ax_t - Bx_t y_t$$

$$y_{t+1} = y_t - Cy_t + Dx_t y_t$$

Assuming A = 0.01, B = 0.001, C = 0.01 D = 0.005. Before performing any calculations, as the project grows, it requires better organization so every data and code can be easily found. By structuring the program using tree directory, different models and tests can be performed, without messing with the main code and avoid repeating the same lines. The following organization may be followed:

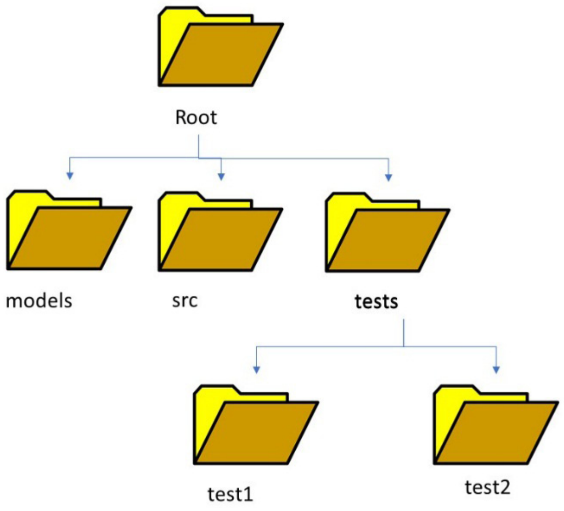


Figure 5: Directory tree for the simulator Project – Version 0.1

The program is now subdivided into three main folders:

- Models: This folder holds the modules with definitions on each model. One model can be composed of a single function, a set of functions, one single class or a set of classes and functions. It is desirable to keep the level of complexity open, so the simulator is a flexible program.
- Src: The abbreviation stands for Source Code, and is the place where the main code to run the simulator stays. For the moment, the folder holds a single module, Simulator.py, with the definition of the Simulators classes as developed before.
- Tests: Any result that one may desire to store can be placed under the tests folder, in a subfolder with an appropriate name holding all the files that the user may have generated as output, such as graphs, spreadsheets and so on.

To do the first implementation of the Predator-Prey model, navigate to the folder and create a module "PredatorPreyv1.py". The "v1" at the end of the file name indicates that this is the first version, or the first attempt to solve this problem. The code inside this file is written below:

```
def model(x,t):
    A = 0.5
```

```
B = 0.2
C = 0.5
D = 1
x1 = x[0] + A * x[0] - B * x[0] * x[1]
x2 = x[1] - C * x[1] + D * x[0] * x[1]
return x1,x2
```

Changes must also be made to the file “workflow.py”, to properly import all the modules and run the simulation. The import statements are rewritten according the following code:

```
from src.Simulators import *
from models.PredatorPreyv1 import *
from tests.preypredatorv1.InitConfig import *
```

A first attempt to run the simulation can be done, by assuming the initial population of gazelles (preys) to 100, the lion population also to 10, and 30 time steps. By doing so, the following output is obtained.

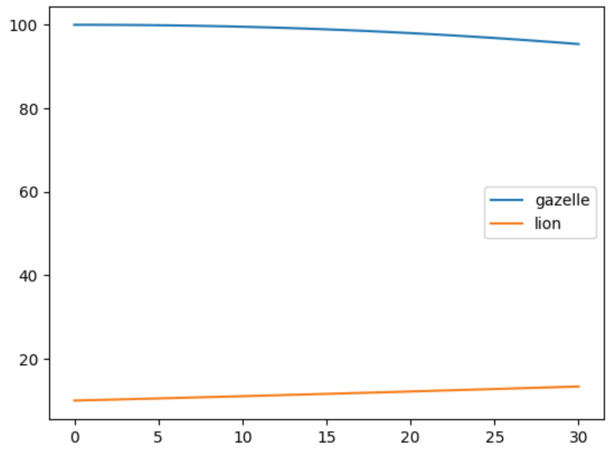


Figure 6: Simulation of predator-prey model – First attemp.

From the figure above, one can see that, as expected, because the population of gazelle is relatively high in comparison to the lion, there is an initial decrease in the gazelle’s population, with a parallel increase in the lion's population. This means that the predator is consuming the prey, and consequently reproducing, while the prey's reproduction is not enough to keep its population constant, so it decreases.

One may also extrapolate ideas from the simulation, inferring that, because the lion's population is steadily growing, the gazelle's population will at some point in time reach zero, and so the lion population will also die because of the lack of prey. However, mental extrapolations of dynamical systems are very dangerous and can arise wrong conclusions. To do a proper inference regarding the population's size, we need to run the model of a larger period of time.

The following figure shows the results for this simulation using N (time steps) = 3000.

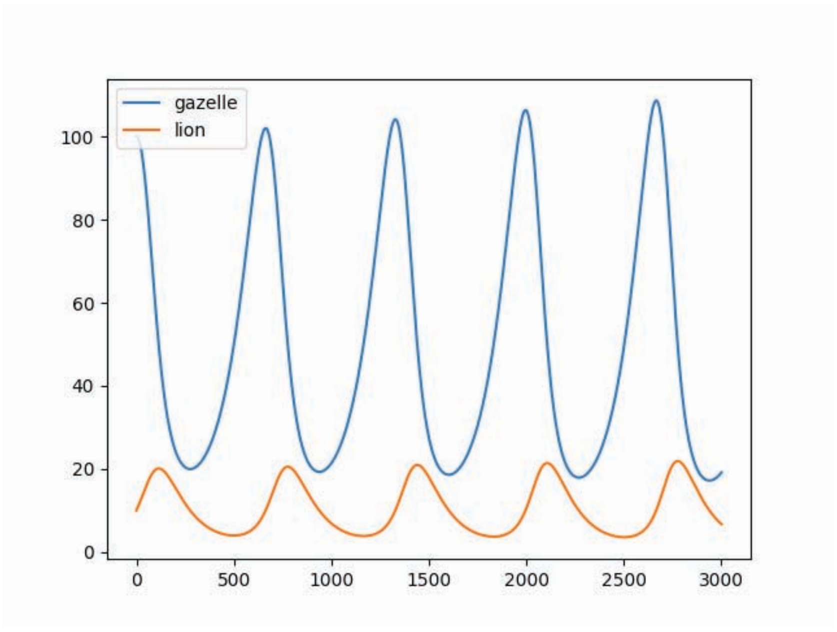


Figure 7: Simulation of predator-prey model. Second attemp

The mental extrapolation that the gazelle population would be exterminate showed itself wrong according the results obtained for a larger simulation time, as shown above. And also a quasi-steady-state can be seen from the behavior of the two species, with an increase of the gazelle population, followed by an increase of the lions population with consequent death of gazelles, and the reduced availability of gazelles also diminishes the lion population, what makes the gazelle population grow again. This pattern is repeated over and over again.

Again, we can perform an analysis on the x-y state phase diagram to retrieve information regarding the (non) linear dynamics of the system. The following figure illustrates this diagram:

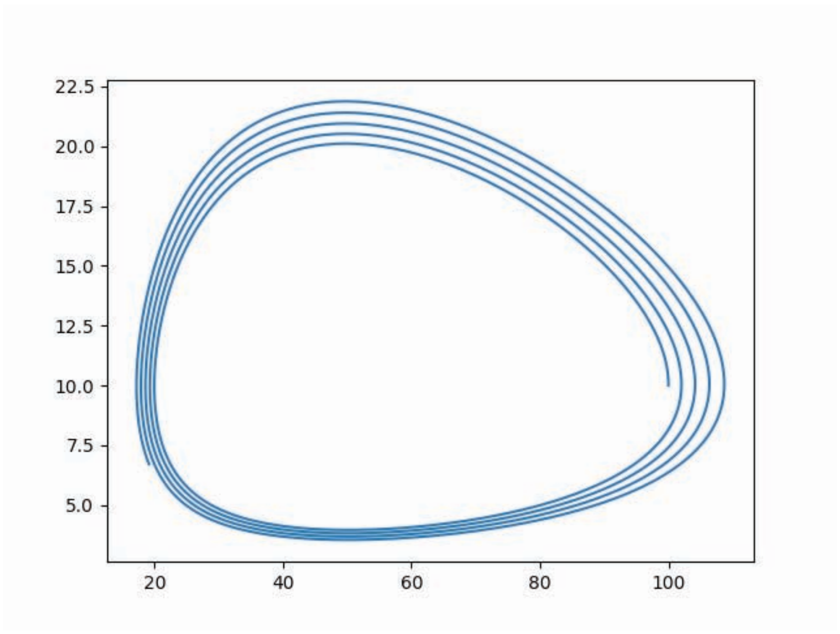


Figure 8: x-y state phase diagram for predator prey model.

From the figure above, it can be seen that the oscillations are not constant, but are monotonically changing in amplitude, either for increasing or for decreasing. In this case, if we simulate the model for enough time, one can see that the oscillations are continuously increasing.

Incorporating object-oriented programming in the model – Predator Prey example

Until this step of development, the simulator previously written is object-oriented. However, the model is implemented using functional programming. The concepts of the prey and the predator using this feature as very abstract. So, the next step is to incorporate object-oriented concepts in the model itself, in such a way that it can easily be extended. In the present case, it is desirable to have a predator-prey model which can incorporates different species and expand from a dual interaction to multiple interactions in a complex ecosystem.

If we think in the prey as an object, and predator as another object, then these concepts can be incorporated into one (or more) classes and the interaction between them, defined inside the methods of the class.

As an initial approach, in order to not raise the complexity of the system too much, one may think that both the predator and the prey are Animals, so in this sense, both belongs to the same class. The class Animal that is here developed refers not only to a single animal but to the population of animals which share the same characteristics, i.e all the prey animals with the same parameters dynamics will make one instance of the class Animal. In a single model, one may have different prey that interacts with the predator in different forms. In this case there will be different instances of Animals representing preys.

In the previous formulation it was shown that the model representing the dual interaction between species has four parameters (A, B, C, D). However, B and D are parameters multiplied by two interacting species. On the other hand, the parameters A and C are multiplied only by the species population that the equation represents.

Two represent this dynamic, we develop one class with three class members:

- Population: refers to the number of animals in the group
- birthrate: a number that represents the ratio of increase of the animal population due to the actual number of individuals.
- Interactionrate: a number that represents the ratio of change in a population to the interaction between the actual specie with another (dual interaction).

Each term of the prey-predator equation prey represents an important dynamic process in the ecosystem. In the class Animal, these processes are represented by two methods:

- Reproduce: This method calculates the balance between the amount of individuals that were born minus those who died of natural death, minus the immigration.
- Interact: This method calculates the change in population due to the interact with other populations. In this case, the change

in the population hunted diminishes while the change in the hunter population increases.

The code developed for this class is saved in the module “PreyPredatorv2”, so we develop a refined version (version 2) of the Prey-Predator model, using object-oriented features. The following is the code for the Animal class:

```
class Animal:
    def __init__(self,population,A,B):
        self.population = population
        self.birthrate = A
        self.interactionrate = B

    def reproduce(self):
        A = self.birthrate
        return A*self.population

    def interact(self,other):
        B = self.interactionrate
        return B*self.population*other.population
```

A few changes have also to be made on other parts of the code. The rest of “PredatorPreyv2.py” module is written as follows:

```
A = 0.01
B = -0.001
C = -0.01
D = 0.0002
gazelle = Animal(100.0,A,B)
lion = Animal(10.0,C,D)

def discrmodel(t):
# x[0] - gazelle population
# x[1] - lion population
```

```
    x1 = gazelle.population + gazelle.reproduce() + gazelle.interact(lion)
    x2 = lion.population + lion.reproduce()  + lion.interact(gazelle)

    if x1<0:
        x1=0
    if x2<0:
        x2=0

    return x1,x2

def set_states(x):
# x[0] - gazelle population
# x[1] - lion population

    gazelle.population = x[0]
    lion.population = x[1]

def get_states():
# x[0] - gazelle population
# x[1] - lion population

    x1 = gazelle.population
    x2 = lion.population
    return x1,x2
```

The same parameters used to test the version 1 of the predator-prey model are again used, to check the correctness of the model. The main model function was renamed to discrmodel, so it explicitly express that this is a discrmodel. This was done because the next step is to develop a continuous model.

Two new functions are incorporated into the program, the get_states ( ) and the set_states ( ). These methods are used during the simulation process to update the value of the states in each object instance, and to retrieve when necessary the actual value of the states inside the objects.

Shifting the focus to the main source code, minor changes had to be made to the "Simulators.py" module. The simulator must be able to access the get_states ( ) and set_states ( ) methods. So, these methods are new class members of the Simulator class. Additionally, because the discrmodel now takes only one argument (the simulation time), this is also reflected in the new code. And the set_states ( ) method is called at each iteration of the simulation to update the states in the objects. The modified constructor method for the Simulator parent class then reads:

```
class Simulator:
    def __init__(self,f,x,t,statesetter,stategetter):
        self.f = f
        self.x = x
        self.t = t
        self.statesetter = statesetter
        self.stategetter = stategetter
```

And the modified update ( ) function of the DiscreteSimulator class is now according the following code:

```
    def update(self,N):
        self.simobs = Observer(self.x,self.t,N)
        x = self.x
        t = self.t
        self.simobs(x,t,0)
        for i in range(1,N+1):
            x = self.f(t)
            t = t + 1
            self.statesetter(x)
            self.simobs(x,t,i)
```

Taking these modifications into account and running the simulation should produce the exactly same result as the example using version 1 of the predator prey model.

In order to show the elegancy of the object-oriented approach and the clear representation of each element in the model using this technique, we extend the system under test by adding an additional population and comparing how the whole ecosystem reacts to the insertion of this third element. Let's say the baboons are now present, with an initial population of 100. The baboons are assumed to be able of reproducing with a ratio two times bigger than the gazelles, however they are also two times more hunted by the lions. For simplicity, the ratio of interaction from the lions to the gazelles and from the lions to the baboons to the be the same.

The rewritten code to represent this new ecosystem is shown below:

```
A = 0.01
B = -0.001

C = -0.01
D = 0.0002

E = 0.02
F = -0.002
gazelle = Animal(100.0,A,B)

lion = Animal(10.0,C,D)

baboon = Animal(100.0,E,F)

def discrmodel(t):
# x[0] - gazelle population
# x[1] - lion population
# x[2] - baboon population

   x1 = gazelle.population + gazelle.reproduce() + gazelle.interact(lion)
   x2 = lion.population + lion.reproduce()  + lion.interact(gazelle) + lion.
interact(baboon)
```

```
x3 = baboon.population + baboon.reproduce() + baboon.interact(lion)

if x1<0:
   x1=0
if x2<0:
   x2=0
if x3<0:
   x3=0

return x1,x2,x3
```

The functions get_states ( ) and set_states must also incorporate the new state, namely the baboon population, but as this step is straight forward there is no need to show it here. The result for simulating 3000 time steps is shown in the figure below.

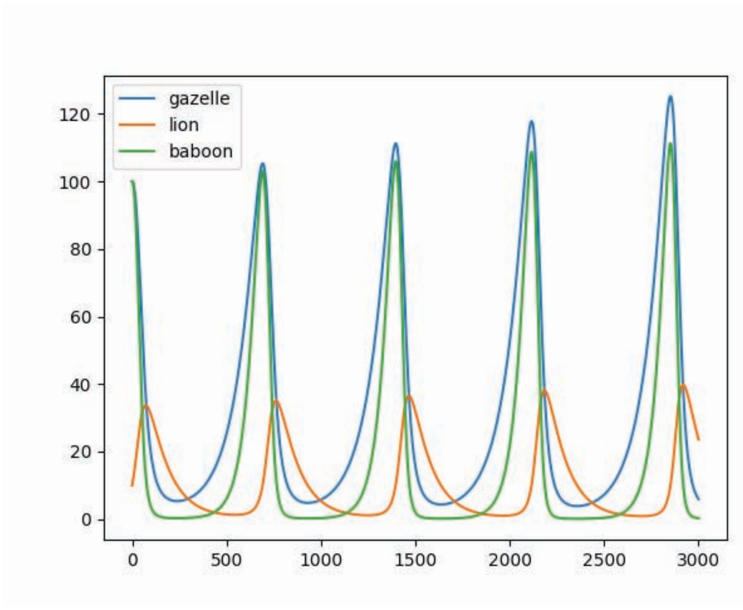


Figure 9: Results for the prey-predator model with three populations.

The reader may try to see how this system behaves with different parameters of birth rate, interaction between population, and different populations. The object-oriented approach in this case helps to see in a non-mathematical form the elements that forms the equation. Without using this resource, the populations are only represented by a value of a variable, "x" for instance, and the meaning of what x stands for is a bit more hidden.

Translating Predator Prey discrete model to continuous model

The next natural step in the development of the Predator – Prey model is to use a continuous model, since the original model is described in terms of the derivatives of the population according:

$$dx = Ax - Bxy$$

$$dy = -Cy + Dxy$$

In the way that is written above, the model can not be solved numerically. In order to simulate, it is necessary to discretize it in space. One can apply the Forward Euler time integration as already mentioned above:

$$x(t + \Delta t) = x(t) + \Delta t * (Ax(t) - Bx(t)y(t))$$

$$y(t + \Delta t) = y(t) + \Delta t * (-Cy(t) + Dx(t)y(t)$$

The main difference from this discrete model to the one developed before, is that the one can keep track of the time step given, increasing or decreasing it in order to have a stable continuous model. The above discretization is just one of the methods that can be used. Another very common method is the Backward Euler, or the 4th order Runge Kutta and other algorithms similar to these.

There are no big changes in the main code, expect that implementation of the time integration algorithms to solve the model. The following code reflects the definition of the function used to calculate the Forward Euler integration and the Runge Kutta in a simple form:

```
def ForwardEuler(f,t,dt,x):
    dx = f(t,x)


    x = x + dt * dx
    return x
def RungeKutta4(f,t,dt,x):
```

```
k1 = f(t,x)
k2 = f(t+dt/2,x+dt/2*k1)
k3 = f(t+dt/2,x+dt/2*k2)
k4 = f(t+dt,x+dt*k3)

x = x + dt/6*(k1+2*k2+2*k3+k4)
return x
```

The class ContinuousSimulator replaces the DiscreteSimulator class to perform the necessary calculations in this case. An additional class member, solver, stores which time integration algorithm is to be used. Regardless of that, minor changes in the update method are implemented, and the code then becomes:

```
def update(self,N,dt):
    self.simobs = Observer(self.x,self.t,N)
    x = self.x
    t = self.t
    self.simobs(x,t,0)
    for i in range(1,N+1):
        x = self.solver(self.f,t,dt,x)
        t = t + dt
        self.statesetter(x)
        self.simobs(x,t,i)
```

USING THE OPENMDAO FOR MODEL DEVELOPMENT AND SIMULATION

OpenMDAO is a high-performance computing platform for systems analysis and optimization that enables you to decompose your models, making them easier to build and maintain, while still solving them in a tightly-coupled manner with efficient parallel numerical methods (OPENMDAO, __).

Along with the software, a library of sparse solvers and optimizers are provided which are able to work with the MPI based, distributed memory data passing scheme. Nevertheless, the user can also choose not to obtain MPI, in which case OpenMDAO runs efficiently in serial using numpy data passing implementation.

The software claims unique capability regarding automatic analytic multidisciplinary derivatives. As long as the user provides the analytical derivatives of the components, OpenMDAO is capable of solving the chain rule across the model, computing system level derivatives for Newton solvers and/or gradient based optimizers. With this feature, the solution of large non-linear problems is possible, even for models with over 25 thousand variables using adjoin derivatives.

If analytical derivatives of the components are not available, OpenMDAO is callable of translating them numerically to finite difference components and computes semi-analytic multidisciplinary derivatives. The ability of the software to make use of semi-analytic derivatives increases computational efficiency greatly. For instance, the computation of an aero-structural wind turbine optimization could be reduced 5x when compared with traditional approaches.

Installation of the software

One can install OpenMDAO easily using a single line in the command prompt:

```
>>  pip install openmdao
```

A second option is to obtain the most recent version of OpenMDAO from the Github repository using the following line in a command prompt:

```
>>  pip install git+http://github.com/OpenMDAO/OpenMDAO.git@
master
```

OpenMDAO is available for Windows, MacOS or Linux systems. The supported platforms of MacOS are:

- Mavericks (10.9.5)
- Yosemite (10.10.5)
- El Capitan (10.11.x)

For Windows the following platforms are supported:

- Windows 7
- Windows 8

- Windows 10 (not officialy)

The platforms for Linux are:

- Trusty Tahr
- Vivid Verdet
- Xenial Xerus

The prerequisites for OpenMDAO are Python, including the basic packages for scientific computing Numpy and Scipy. The supported versions of Python are 2.7.9 and 3.4.3, although it may also work in more recent versions. The Numpy library should be version 1.9.2 or above. Scipy package supported is 0.15.1 or above.

Basic Object-Oriented modeling with OpenMDAO

This section describes the basic concepts and tools used in OpenMDAO to define correctly a problem and to solve it.

A System is the basic concept in OpenMDAO. Systemas are related to Components and Groups. Component is the computational class in OpenMDAO, where the user develops the model and wrap external analysis code. Group are collections of Components and other sub Groups with data passing and execution sequence. Problem contain the whole model.

System

In 2015, Hwang developed a mathematical architecture called Modular Analysis and Unified Derivatives, representing an unique abstraction to model large system represented by many smaller components. This architecture is the basis of OpenMDAO.

The fundamental concept of the Modular Analysis and Unified Derivatives (MAUD) is that an entire system can be represented as a hierarchical set of systems of (non) linear equations. This architecture is composed by a fundamental building block referred to in OpenMDAO as System block. This class represents a system of equations that needs to be solved together in such a way that a single solution satisfies them all.

A system of equations is composed of input values (parameters) and output values (unknows). Suppose as an example the linear equation below:

$y = 2x + 3$

A system defined by the equation above has 1 parameter, or input value (x) and one output value (y). The above equation is said to be explicit, since the required output variable is isolated from the rest of the equation and can be directly obtained by calculating the right hand side of the equation.

Another way to represent the same equation is to make it implicit. Using this approach, a residual (which should be zero) is equal to the equation shifted to one side of it. The system composed by one linear equation stated above could be rewritten in the implicit form as:

$R(y) = y - 2x - 3 = 0$

In this configuration, the system acquires a new attribute, called resids, which stores the list of residuals for efficient processing.

The equations are defined in one method of the System class called solve_nonlinear( ). This method can directly calculate the value of unknows for explicit equations, or find the correct value for the states that converges the residuals to zero.

Another method of this class, apply_nonlinear( ) computes the residuals values of a state. This function is not used if the system is defined only by explicit equations.

The Component class and the Group class are subclasses of the System.

Component

According the definitions of the OpenMDAO, Component is the child class of Subsystem that composes the lowest level system. The classes that are derived from it are the only ones allowed to create parameter, output, and state variables. The development of new models is done by subclassing the Component class and defining a solve_nonlinear and/or apply_nonlinear methods which defines the specific dynamics of processes under study.

The necessary variables are added to the class in the constructor (__init__) using the methods add_parameter, add_output and add_state functions. In the following code, a simple component is created and parameters, outputs and states are added to it.

```
class MyComponent(Component):
    def __init__(self):
        super(MyComp, self).__init__()
        self.add_param('a', val=0.)
        self.add_output('b', shape=2)
        self.add_state('c', val=[-1., 1.1])
```

It is necessary to specify initial values for the variables of the component, so the program can efficiently allocate the needed space in the vectors for data passing. The default shape for a single value if float, while the default shape for a vector is numpy float array.

The method solve_nonlinear( ) takes as arguments the parameters, the unknown vectors and a residuals vector. These are stored as dictionaries in the Component class, so the reference to them is done as one would do with any dictionary in Python. The following code exemplifies this method:

```
def solve_nonlinear(self, params, unknowns, resids):
    unknowns['y'] = 2 * params['x'] + 3
```

If the solve_nonlinear method defines any implicit equation, then the apply_nonlienar method must be implemented. The function calculates the residuals for the given parameters and states. In OpenMDAO, two options can be used to define how to converge implicit equations:

- OpenMDAO solver
- Make the component converge itself.

The definition of analytical derivatives can be done in a component by overriding the linearize method. This method linearizes the non-linear equations and delivers the partial derivatives to the framework.

```
def linearize(self, params, unknowns, resids):
    J = {}
    J['y','x'] = 2
    J['y','y'] = 1
```

Group

The Group class is used to collect smaller objects in order to organize complex system into sub-units. Groups can be formed by a collection of components or a collection of smaller groups. In essence,, Group is a System object instance composed of the equations from its children that are linked via data connections.

The fact that a Group can hold other groups in OpenMDAO creates a powerful object-oriented interface. This configuration allows complex systems to be seen in a simple way, where in the higher level the main concepts are seen, and the inner levels intrinsic dynamics and details of the system are presented without making the whole system too much confusing.

The creation of Groups in the framework is done by adding one or more Systems or Groups, or even a mixture of them.

```
comp1 = MyComp()
comp2 = MyComp()
comp3 = MyComp()
comp4 = MyComp()
group1 = Group()
group1.add(‘comp1’, comp1)
group1.add(‘comp2’, comp2)
group1.add(‘comp3’, comp3)
group2 = Group()
group 2.add(‘comp4’, c3)
group 2.add(‘sub_group_1’, group1)
```

The dependencies among systems in a group are represented by the connections between the variables in the Group’s subsystems. These connections can be estabilished in two ways: explicitly or implicitly.

The explicit connection estabilished the flow of information from one ouput (or state) of the System to an input (parameter) of another system using the Group connect method. For instance:

Group1.sub_group_1.connect('comp1.y', 'comp2.x')

The implicit connection is established using the promotion mechanism in a Group. Whenever, a System is added to a Group, variables can be specified to be promoted from the subsystem to the group level. In this way, a variable can be referred as it was from the Group instead of the Subsystem which it belonged originally.

Group1.add("comp1", component1, promotes=['y'])

If multiple variables subsystem are promoted with the same name, then those variables will be implicitly connected.

A Group is an element of the architecture of OpenMDAO used to assemble multiple system of equations and solving them together. In this sense, they differ from the Component class which is used to define variables and equations that defines the transformations inside the system. The Group class uses a Solver to solve the collection of Components is it was a single problem. Two solvers are implemented: a linear solver and a non-linear solver. The default linear solver is Scipy GMres and the default non-linear solver is a RunOnce solver. This last one will call the solve_nonlinear method on each system in the Group sequentially. Besides the default ones, there is a collection of other linear and non-linear solvers that can be used in replacement of the defaults.

Problem

The problem is a single instance, top-level element used to couple all the Groups forming the Model itself. The Problem instance can be used to perform analyses, to design experiments or to do optimization-.

The Problem has a single top-level Group called root. This group can be attributed to the Problem instance when creating it, via the constructor, or passed later. The following is an example of creating a problem and passing the root group via the constructor.

prob = Problem(ExampleGroup())

In order to control the solution of the problem, OpenMDAO uses a driver class. The base Driver class is the simplest driver, which works by simply calling the solve_nonlinear method on the root Group. Nonetheless, there are a variety of other drivers available to perform different experiments, such as optimization, case iteration and design of experiment drivers. The driver is the object which determines how the problem will be solved.

The simulation is called by first setting up the problem and then calling the method run, as follows:

```
prob.setup()
prob.run()
```

TANK SYSTEM MODELLING

In the following section, it is implemented an object-oriented approach for modeling a system of fluid tanks using the programming language Python. This is a simple but useful type of process, especially in chemical engineering problems, and it can be further extended with additional process dynamics, such as chemical reactions that occur in the tank, heating jackets and other features. The approach here developed can be also used to other unit operations in chemical and process engineering.

The system of tanks can be represented by the figure below. There is a continuous source of fluid to the system, with a known amount of fluid per time

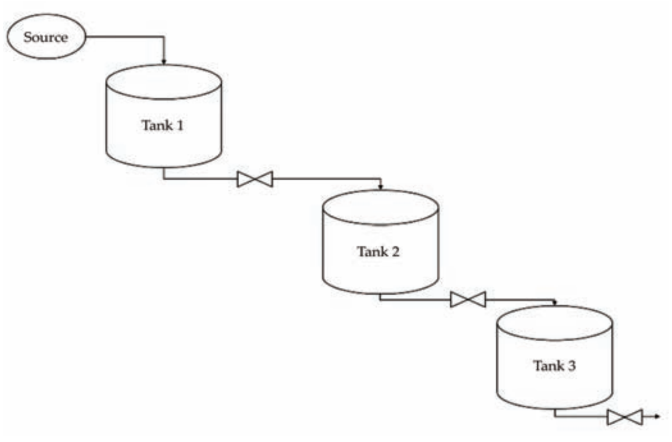


Figure 10: Tank system representation.

Some features of the system here implemented are:

- There is no interaction between the levels of each tank. This means that the level in the second tank does not interfere with the level of the first one.

- Each tank has a constant cross section area. The system can be further extended to also represent tanks with a table relating fluid level and cross section.
- The discharge of teach tank is proportional to the square root of the fluid level, according:

$$q = C_v \sqrt{h}$$

Where q is the discharge, h is a coefficient of discharge and h is the fluid level in the tank.. This equation comes from a simplified representation of the Bernoulli equation shown below:

$$q = \sqrt{2gh} c_0 \pi \left(\frac{d}{100} \right)^2 / 4$$

Where g is the gravitational acceleration, d is an dimensionless discharge coefficient, and d is the diameter.

The objective is to obtain the states and outputs of the system, i.e, the fluid levels and the flows given as boundary condition the flow at the source at some points in time.

The system can be represented using the continuous state-space representation:

$$\dot{x} = f(t, x, \ldots)$$

$$y = g(t, x, \ldots)$$

Where $\dot{x}$ is the derivative vector of the states, t is the time, x is the state vector (fluid level at each tank), $f(t,x,\ldots$ is the output vector (discharge of each tank), and $f(t, x, \ldots)$ and $g(t, x, \ldots)$ are (non) linear functions representing the dynamics of the process.

The first step is to solve the problem is to apply the material balance to each tank, i.e, the amount of fluid entering minus the amount leaving is equal to the rate of accumulation of fluid in the tank. In mathematical representation:

$$A \frac{dh}{dt} = q_i - q_{out}$$

Where q_i is the cross-section of the tank, q_i is the inflow and q_{out} is the outflow, governed by the Bernoulli equation described above. Representing each tank dynamics in state-space representation:

$$\dot{h} = q_i - q_{out} / A$$

$$q_{out} = C_v \sqrt{h}$$

Developing the above representation for each tank in the system, the state space representation is extended to the vector form:

$$\begin{bmatrix} \overset{\text{ÿ}}{h}_1 \\ \overset{\text{ÿ}}{h}_2 \\ \overset{\text{ÿ}}{h}_3 \end{bmatrix} = \begin{bmatrix} q_i / A_1 \\ q_1 / A_2 \\ q_2 / A_3 \end{bmatrix} - \begin{bmatrix} q_1 / A_1 \\ q_2 / A_2 \\ q_3 / A_3 \end{bmatrix}$$

$$\begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} C_{v1} \\ C_{v2} \\ C_{v3} \end{bmatrix} - \begin{bmatrix} \sqrt{h_1} \\ \sqrt{h_2} \\ \sqrt{h_3} \end{bmatrix}$$

To solve this problem, one can apply Forward Euler time integration scheme to the derivatives of the state. With this approach to complete system is update at each time step using the following set of equations:

$$h_1^{t+\Delta t} = h_1^t + \Delta t \left(q_i^t - q_1^t\right)/A_1$$

$$h_2^{t+\Delta t} = h_2^t + \Delta t \left(q_1^t - q_2^t\right)/A_2$$

$$h_3^{t+\Delta t} = h_3^t + \Delta t \left(q_2^t - q_3^t\right)/A_3$$

$$q_1^{t+\Delta t} = C_{v1}\sqrt{h_1^t}$$

$$q_2^{t+\Delta t} = C_{v2}\sqrt{h_2^t}$$

$$q_3^{t+\Delta t} = C_{v3}\sqrt{h_3^t}$$

$$t = t + \Delta t$$

First approach – Functional programming

To show the advantages of the object-oriented programing when solving this type of problem, it is first implemented a direct, functional programming approach to solve the problem of the three tank system. The functional programming, as the name depicts, is a code composed of functions defining the problem and how to solve it. The main advantage of this type of approach is that the rapidly development of simple problems allows one to solve them quickly. However, it is difficult to extended such codes and, once the problems becomes more and more complex, it can become confusing to identify the functionality of each component of the program.

The first step consists of writing a function describing the problem of the three tanks. In Python, the code for this problem can be written as follows:

```
import numpy as np
def system_of_tanks(t,x,t_table,q_table,valve,A,ce):
    h0 = x[0]
    h1 = x[1]
    h2 = x[2]
    dx = np.zeros(3)
    y = np.zeros(3)

    qi = np.interp(t,t_table,q_table)
    y[0] = ce[0]*valve[0]*np.sqrt(h0)
    dx[0] = (qi - y[0])/A[0]

    y[1] = ce[1]*valve[2]*np.sqrt(h1)
    dx[1] = (y[0] - y[1])/A[1]

    y[2] = ce[2]*valve[2]*np.sqrt(h2)
    dx[2] = (y[1] - y[2])/A[2]
    return dx,y
```

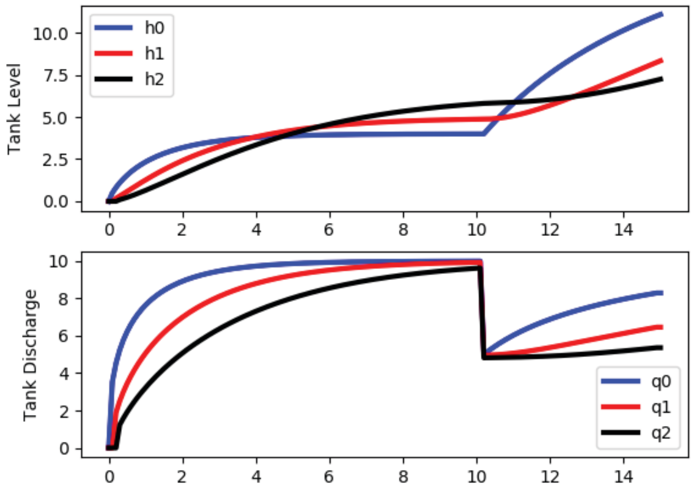
The code is straight forward. As input to the function, it is given the previous states of the system, the current time, the flow source data, the opening of the valves in the outlet of each tank, the cross section of the tanks and the discharge coefficient. The function returns the derivative of the states (the derivative of the fluid level in each tank) and the current ouputt (the discharge of each tank).

This function can be used to solve a variety of problems consisting of different flow sources condition, different tanks geometries and coefficients of discharge of the tanks. Nonetheless, it is strict with regards to a system of three tanks connected in series without interaction from downstream to upstream. If one desires to test the effect of adding one more tank or providing interaction among the tanks, then the function must be rewritten. This step can lead to errors in the code and can be even difficult to trace.

The rest of the code consists in defining the flow source condition, the valves opening, tank cross sections and coefficient of discharges. A second function (spreadsheet) derived from the first one is written applying the definitions on the variables mentioned above, and the problem is simulated using the Forward Euler time integration scheme according the code below.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
tq = [0,100]
   q_table = [10,10]
   valve = [1.0,
          1.0,
          1.0]
   A = [2,
       2,
       2]
   ce = [5,
       4.5,
       4]

   spreadsheet = lambda t,x: system_of_tanks(t,x,tq,q_table,valve,A,ce)

   N = 151
   dt = 0.1
   t = np.zeros(N)

   x = np.zeros([3,N])
   y = np.zeros([3,N])

   for i in range(N-1):
       dx,y[:,i] = spreadsheet(t[i],x[:,i])
       if t[i]>10.0:
           valve = [0.5,0.5,0.5]
       x[:,i+1] = x[:,i] + dt*dx
       t[i+1] = t[i] + dt

   df0 = pd.DataFrame(y.transpose(),t)
   df1 = pd.DataFrame(x.transpose(),t)
   frames = [df0,df1]
   df = pd.concat(frames,axis = 1)
   df.to_csv('example.csv')

   plt.figure()
   plt.subplot(2,1,1)
   plt.plot(t,x[0,:],'b-',linewidth=3,label='h0')
   plt.plot(t,x[1,:],'r-',linewidth=3,label='h1')
   plt.plot(t,x[2,:],'k-',linewidth=3,label='h2')
   plt.ylabel('Tank Level')
   plt.legend(loc='best')
   plt.subplot(2,1,2)
   plt.plot(t,y[0,:],'b-',linewidth=3,label='q0')
   plt.plot(t,y[1,:],'r-',linewidth=3,label='q1')
   plt.plot(t,y[2,:],'k-',linewidth=3,label='q2')
   plt.ylabel('Tank Level')
   plt.legend(loc='best')
   plt.show()
```

The code above produces a .csv file with the simulated outputs ("example.csv") and a graphical output, as shown below.



In order to better visualize the dynamics of the system, the valves opening were halved at t = 10 s so as to disturbe the steaty state already reached by the 1st tank around t = 4 s. In the following approaches, the result of the program is exactly the same regarding the values of the states and of the outputs. However, it is shows that the advantage of using an object-oriented approach is the clear functionality of different components of the code and easy extensibility for more complex systems (interacting tanks, more tanks in series, controlling the tanks level, etc).

Second approach – Improved Functional programming

In this approach, the first tentative in representing the tank system is extended by decomposing the single function representing the dynamics of the process into subfunctions. One function represents the dynamics of each tanks. The second function incorporates this function, coupling them and generating the tanks system.

With this approach, the functional programming is more easily extensible than the first one. As the dynamics of each tank does not change, more tanks can, in a clearer way, be added to the problem in a way that is more next to the object-oriented approach.

The function representing the dynamics of each tank is written according the following code in Python programming language:

```
def tank(t,x,qi,valve,A,ce):
    y = ce*valve*np.sqrt(x)
    dx = (qi - y)/A
    return dx,y
```

This function can be incorporated into another to represent a single tank system or even a series of many tanks. The function that incorporates the one above and represent the three tank system is the one written below. In this function, notice how the subfunction “tank” is called and how clearer is this code regarding the readability and understanding of what type of structures is the system composed of.

```
def system_of_tanks(t,x,t_table,q_table,valve,A,ce):
    h0 = x[0]
    h1 = x[1]
    h2 = x[2]
    dx = np.zeros(3)
    y = np.zeros(3)

    qi = np.interp(t,t_table,q_table)
    dx[0],y[0] = tank(t,x[0],qi,valve[0],A[0],ce[0])

    dx[1],y[1] = tank(t,x[1],y[0],valve[1],A[1],ce[1])
    dx[2],y[2] = tank(t,x[1],y[1],valve[2],A[2],ce[2])

    return dx,y
```

The rest of the code is exactly the same as depicted for the first functional programming approach shown. However, this type of programming has clear advantages in comparison with the last one, regarding extensibility and readability.

Third approach – Object-Oriented programming

The next approach consists into a big modification of the whole procedure of calculating, in order to develop a full object -oriented approach of the system under study. It will be clear by the end of this section that this procedure is much more flexible and easily extensible than the former approaches.

It is intuitive that the tanks in the system are objects, pertaining to a class that defines a generic tank of any cross section and what are the dynamics of any tank. Additionally, the inflow source is also an object of a generic Source class, that defines any table of data against time given as input to a system. The output of the tanks is regulated by valves, which are also seen as objects of a generic Valve class.

Any of these classes mentioned above can be seen as a generic Unit, with connections which are referred to as Inlet and Inlet. Moreover, the generic Unit class pre-defines some methods that are common to the subclasses. These methods are the equation method, which defines the dynamics of all the units pertaining to a subclass, and an apply state function which is used at the end of each simulation step to update the states of the unit. The code for the Unit class is written in Python as follows:

```
class Unit:
    def __init__(self):
        self.Inlet = []
        self.Outlet = []
    def equation(self,t,x):
        return np.array(0),np.array(0)

    def apply_states(self,x):
        y = 0
```

The classes defining other components of the system are derived from this parent class. The first one to be defined is a generic Signal Source class, which defines the component which provides a pre-defined signal at each time step according a table of time x signal. The code of this class is shown below:

```
from scipy import interpolate
class Source(Unit):
    def __init__(self,t_table,q_table,kind):
        Unit.__init__(self)
        self.q_table = interpolate.interp1d(t_table,q_table,kind)
        self.signal = 0.0

    def equation(self,t,x):
        y = self.q_table(t)
        return np.array(0),np.array(y)

    def setInlet(self,objup):
        self.Inlet = objup

    def setOutlet(self,objdown):
        y = self.q_table(0)
        self.setSignal(y)
        self.Outlet = objdown

    def setSignal(self,value):
        self.signal = value

    def apply_states(self,x,y):
        self.setSignal(y)
```

This class defines three additional methods, setInlet( ), setOutlet( ) and setSignal( ) methods. These type of functions are called setter methods, for they are used to set an attribute without directly accessing it. In Python is not possible to define private methods, so the user should be aware which methods are to be directly accessed and which are to be accessed using setter methods. In the case of the Inlet and Outlet properties, it can be seen that some other procedures are performed when the program attempts to update this property. If the property is directly updated, such as in the following case:

```
Signal.Outlet = obj
```

May not generate directly any errors, but the program may not perform as expected, since the procedures calculated using the setter methods were bypassed. So, in the program flow it is important to, either know which methods should be updated using a setter method, or do not use setter methods at all (when possible), and a third option and the most robust one is to use a setter method for all properties in a class, so no confusion will exist if a property requires to be updated using a setter method.

From this class, a FlowSource class is derived and the main difference between the parent class and this subclass is an extra property, flow which is equal to the signal generated. The code is the following:

```
class FlowSource(Source):
    def __init__(self,t_table,q_table,kind):
        Source.__init__(self,t_table,q_table,kind)
        self.flow = 0.0

    def setSignal(self,value):
        self.flow = value
```

This class, a child from the Source class, overrides the constructor by calling the Source class constructor and adding a property, flow. The setter method setSignal( ) is overriding to redefine the attribution of values to the flow property, instead of the signal property, which is not used in this case.

The third class is the Valve, which defines the behavior and characteristics of the Valve objects present in the system. These valves behaves according Bernoulli equation mentioned at the beginning of the section. The opening of the valves is defined using a signal Source obj. This is done so as to have more flexibility regarding the control and manipulation of the opening of the valves. The Valve class is derived from the generic Unit superclass, and defined in Python according the following code:

```
class Valve(Unit):
    def __init__(self,ce,SignalSourceobj):
        Unit.__init__(self)
        self.ce = ce
        self.Pos = SignalSourceobj
        self.flow = 0

    def equation(self,t,x):
        ce = self.ce
        Pos = self.Pos.signal
        h = self.Inlet.Height
        y = Pos*ce*np.sqrt(h)
        return np.array(0),np.array(y)

    def setInlet(self,objup):
        ce = self.ce
        Pos = self.Pos.signal
        h = objup.Height
        y = Pos*ce*np.sqrt(h)
        self.Inlet = objup
        self.flow = y
    def setOutlet(self,objdown):
        self.Outlet = objdown

    def apply_states(self,x,y):
        self.flow = y
```

The last main physical component of the system to be defined is the Tank class, which defines the behavior and properties of the tank instances that are present in the process. This class is also a subclass of the Unit. The code for this type of component is shown below:

```
class Tank(Unit):
    def __init__(self,A,Height):
        Unit.__init__(self)
        self.A = A
        self.Height = Height

    def equation(self,t,x):
        A = self.A
        qi = self.Inlet.flow
        qout = self.Outlet.flow

        dx = (qi - qout)/A

        return np.array(dx),0

    def setInlet(self,objup):
        self.Inlet = objup

    def setOutlet(self,objdown):
        self.Outlet = objdown
    def apply_states(self,x,y):
        self.Height = x
```

This is the only class in this system which returns a derivative, instead of an algebraic output of an equation or an interpolation. Therefore, the time integration scheme has to be implemented so simulate this system, according a defined timestep.

The last class, System, is used to join all the instances into a single structure, connect them and to simulate the system using a predefined time integration scheme. It is used as an initial approach the Forward Euler time integration procedure, which is one of the simplest, however it lacks stability, therefore being necessary to use small time steps of simulation. This integration scheme is specially difficult to use for stiff problems.

The class System defines three methods, besides the constructor:

- Add( ). This method is used to collect the instances created into a single structure (a Python dictionary in the present case).
- Connect( ): This function is used to connect the instances, i.e to set the Inlets and Outlets of the instances in the proper way (if the outlet of obj1 is obj2, then the inlet of obj2 is obj1)
- Simulate( ): This method is used to predict the system behavior in a defined time horizon, using a time step for the integration procedure. The user gives the time step and the amount of steps of simulation, and the results are stored as class members.

The code written for this class is the following:

```
class System:
    def __init__(self):
        self.unit = dict()
        self.t = []
        self.x = []
        self.y = []

    def add(self,ID,obj):
        self.unit[ID] = obj
```

```
def connect(self,ID1,ID2):
    self.unit[ID1].setOutlet(self.unit[ID2])
    self.unit[ID2].setInlet(self.unit[ID1])

def simulate(self,dt,N):
    unit = self.unit

    t = np.linspace(0.0,N*dt,N)

    numcolumns = len(unit)
    x = np.zeros([numcolumns,N])
    y = np.zeros([numcolumns,N])
    dx = np.zeros([numcolumns])
    yp = np.zeros([numcolumns])

    for i in range(N-1):
        k = 0
        for j in unit:
            dx[k],yp[k] = unit[j].equation(t[i],x[k,i])
            k = k + 1

        x[:,i+1] = x[:,i]+ dt*dx
        y[:,i+1] = yp

        k = 0
        for j in unit:
            unit[j].apply_states(x[k,i+1],y[k,i+1])
            k = k + 1
        self.t = t
        self.x = x
        self.y = y
```

The flow source table, valves signal table, as well as the geometries of the tanks and the valves are defined according the following code in Python:

```
tq = [0,100]
q_table = [10,10]

tp = [0,10,15]
pos0_table = [1,0.5,0.5]
pos1_table = [1,0.5,0.5]
pos2_table = [1,0.5,0.5]



A = [2,
    2,
    2]
ce = [5,
     4.5,
     4]
```

Where ‘t_q’ and ‘q_table’ are the table data of the flow source. The ‘tp’ is the time vector for the definitions of the valves signal sources ‘pos0_table’, ‘pos1_table’ and ‘pos2_table’. The ‘A’ vector defines the cross-section area of the tanks. The vector ‘ce’ stores the values of the discharge coefficient of the valves. To define the objects composing the system, first it is necessary to create the signal sources, both flow and valves signal.

```
valve0source = Source(tp, pos0_table, ‚zero‘)
valve1source = Source(tp, pos1_table, ‚zero‘)
```

```
valve2source = Source(tp, pos2_table, ,zero‘)
flowsource = FlowSource(tq, q_table, ,linear‘)
```

The argument ‘zero’ is used to define that this type of signal should be interpolated using zero-order hold, i.e the last input value is given until a new one is presented, when the signal is updated. This type of interpolation opposes with, for example, the linear one where the input at the current time is a linear interpolation of the two values next to it.

Next, it is defined the tanks and the valves using the specified values according the variables mentioned above. The code for these definitions is shown below:

```
tank0 = Tank(A[0],0.0)
valve0 = Valve(ce[0],valve0source)

tank1 = Tank(A[1],0.0)
valve1 = Valve(ce[1],valve1source)

tank2 = Tank(A[2],0.0)
valve2 = Valve(ce[2],valve2source)
```

Finally, the connection of the whole system, including the connection of the elements and running the simulation is done through the creation of an instance of the System class. To create the System class, the constructor is called without arguments, according the following line:

```
spreadsheet = System()
```

The following lines of code are used to insert all the created objects into a single structured list (a Python dictionary).

```
spreadsheet.add(‘flowsource, flowsource)
spreadsheet.add(‘tank0, tank0)
spreadsheet.add(‘valve0source, valve0source)
spreadsheet.add(‘valve0, valve0)
spreadsheet.add(‘tank1’, tank1)
spreadsheet.add(‘valve1source’, valve1source)
spreadsheet.add('valve1', valve1)
spreadsheet.add('tank2', tank2)
spreadsheet.add('valve2source', valve2source)
spreadsheet.add('valve2', valve2)
```

Next the outlet and inlet of the element have to be connected according the process structure using the connect( ) method of the spreadsheet instance object. The following lines of code are used to connect the instances.

```
spreadsheet.connect('flowsource', 'tank0')
spreadsheet.connect('tank0', 'valve0')
spreadsheet.connect('valve0', 'tank1')
spreadsheet.connect('tank1', 'valve1')
spreadsheet.connect('valve1', 'tank2')
spreadsheet.connect('tank2', 'valve2')
```

The simulation can be performed by calling the simulate( ) method of the spreadsheet instance object, with the timestep and the number of steps as input arguments to the function, as shown in the following code:

```
N = 150
dt = 0.1
spreadsheet.simulate(dt,N)


x = spreadsheet.x
y = spreadsheet.y
t = spreadsheet.t
```

One major advantage of the object-oriented programming is shown in the following case, where the whole system is rearranged in a new configuration.

Advantages of the object-oriented programming – rearranging the Tank system

In this section, the same program used before is used and expanded in order to model a new system of Tanks, according the following figure:

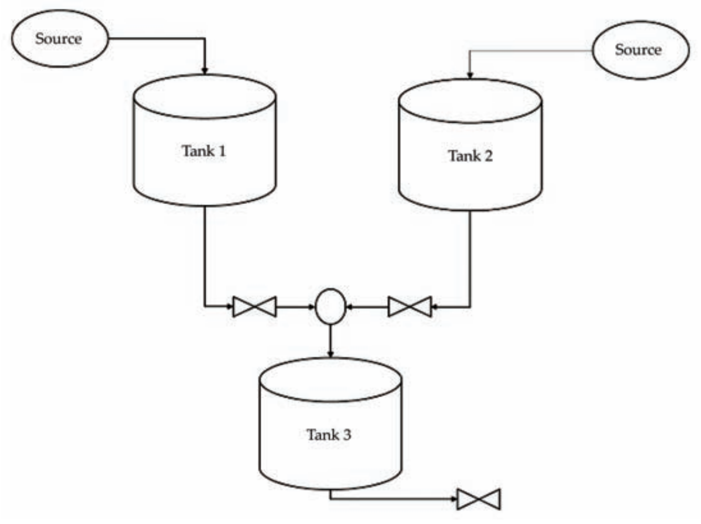


Figure 11: Rearranged Tank system.

In this new configuration, a new source is added, and the sum of the contributions of the first two tanks flow to the third tank. The sum of the contributions can be done by adding a new class to the System, which is called Joint. The code of the new class is shown below:

```
class Joint(Unit):
    def __init__(self):
        Unit.__init__(self)
        self.flow = 0.0

    def equation(self,t,x):
        y = 0
        for i in self.Inlet:
            y = y + i.flow
        self.flow = y
        return np.array(0),np.array(y)

    def setInlet(self,objup):
        self.Inlet.append(objup)
        y = 0
        for i in self.Inlet:
            y = y + i.flow
        self.flow = y

    def setOutlet(self,objdown):
        self.Outlet = objdown

    def apply_states(self,x,y):
        self.flow = y
```

The advantage of the object-oriented approach is that no modifications on the previously developed objects are to be performed. An instance of the Joint class has to be created and connections are changed in such a way that it reflects the new system under interest. The creation of this instance is done according the following code:

```
joint = Joint()
```

Which defines the object 'joint'. Additionally, there is an extra flow source which feeds the Tank 2. The definition of this object is done according the following code:

```
flowsource1 = FlowSource(tq,q_table,'linear')
```

The spreadsheet object is redefined according the code below:

```
spreadsheet = System()
spreadsheet.add('flowsource0',flowsource0)
spreadsheet.add('flowsource1',flowsource1)
spreadsheet.add('tank0',tank0)
spreadsheet.add('valve0source',valve0source)
spreadsheet.add('joint',joint)
spreadsheet.add('valve0',valve0)
spreadsheet.add('tank1',tank1)
spreadsheet.add('valve1source',valve1source)
spreadsheet.add('valve1',valve1)
spreadsheet.add('tank2',tank2)
spreadsheet.add('valve2source',valve2source)
spreadsheet.add('valve2',valve2)

spreadsheet.connect('flowsource0','tank0')
spreadsheet.connect('flowsource1','tank1')
spreadsheet.connect('tank0','valve0')
spreadsheet.connect('valve0','joint')
spreadsheet.connect('tank1','valve1')
spreadsheet.connect('valve1','joint')
spreadsheet.connect('joint','tank2')
spreadsheet.connect('tank2','valve2')
```

The rest of the code remains the same. New configurations of the system can be tested without deep modification of the code. The object-oriented approach allows the definition of different process flowsheets by reusing the same classes, rearranged according the necessity. This same procedure would not be possible using the functional programming, where the main function needs to be rewritten in order to reflect the new conditions under study.

FURTHER READING

In this section, we present some useful literature for those interested in extending their knowledge in Scientific Computing, Object-Oriented programming and the application of Object-Oriented programming in Scientific Computing.

Gladwell, I; Nagy, J. G; Ferguson Jr., W. E. (2007) Introduction to Scientific Computing. Available in: http://www.mathcs.emory.edu/~ale/NAbook_Aug_2008.pdf

This book provides principles on scientific computing, with many examples on basic mathematical algorithms on solving linear systems, differentiation, integration and other methods. Application of such methods using Matlab is done all over the book, providing the reader clear insight without requiring high expertise with Numerical Methods.

Bindel, D; Goodman, J. (2009) Principles of Scientific Computing. Available in: http://www.cs.nyu.edu/courses/spring09/G22.2112-001/book/book.pdf

This book covers numerical methods used in scientific computing in a broad sense, from analysing sources of errors in computation, linear algebra methods and algorithms up to nonlinear equations and optimization, dynamics and differential equations finishing with Monte Carlo method applications. Each chapter provides exercises so the reader can take deep insight into every concept presented.

Heath, M. T. (2002) Scientific Computing: an introductory survey. 2nd ed. McGraw Hill.

Besides covering numerical methods used in scientific computing, softwares that can be used are presented by the author, with historical description and further reading on each topic along the book. Three chapters are dedicated to numerical methods used in solving differential problems: the first one describes initial value problems, the second one boundary value problems and the last one the use of numerical methods for solving partial differential equations.

Johansson, R. (2016) Introduction to Scientific Computing in Python. Available online.

The author presents, in a very simple and useful manner the way Python as a programming language can be used to solve problems in numerical computation. The first chapter introduces Python from the very beginning, showing what is it, its features and how to obtain. Further chapters present different libraries in Python useful for Scientific Computing such as Scipy, Numpy and Sympy. The author also explain how Python can be integrated with other programming languages (C and Fortran) , tools for high-performance computing and how to control software versions.

Pitt-Francis, J; Whiteley, J. (2012) Guide to Scientific Computing in C++. Springer-Editor London. ISBN: 978-1-4471-2736-9. DOI: 10.1007/978-1-4471-2736-9

The authors present essential principles on using object-oriented C++ programming for scientific computing. Many examples are given to support the theory described and to help the reader to familiarize himself with the concepts presented. Special features of the language are also described, such as parallel computing using MPI. A brief introduction

to the language is first presented, and later more advanced features are examined, such as templates and exceptions.

Yang, D. (2001) C++ and Object-Oriented Numeric Computing for Scientists and Engineers. Springer New York. DOI: 10.1007/978-1-4613-0189-9. ISBN: 978-1-4613-0189-9

Basic concepts of C++ programming language, and object oriented numeric computation for students and professionals are described in this easy to read and complete book. Examples are shown independent of the operating system. At the end, special features not present in other languages used for scientific computing are presented, such as the preconditioned conjugate gradient (CG) method and generalized minimum residual (GMRES) method.

Henderson M. E; Anderson C. R; Lyons S. L. (1999) Object Oriented Methods for Interoperable Scientific and Engineering Computing (Proceedings in Applied Mathermatics, 99). Society for Industrial & Applied Mathematics,U.S. (29. September 1999)

The book is a compilation of the papers presented at the October 1998 SIAM Workshop on Object Oriented Methods for Interoperable Scientific. It covers different topics and problems related with designing and implementing computational tools for science and engineering.

Langtangen, H. P. (2016) A Primer of Scientific Computing with Python. Springer Berlin Heidelberg. ISBN: 978-3-662-49887-3

The book covers basic principles of Python programming language, and advanced features with many applications in scientific computing and numerical methods. Some examples are shown from the perspective of first, functional approach up to a full object-oriented programming approach. The language is concise, easy to understand and provides the necessary information to develop good knowledge with Python language.

Kiusalaas, J. (2013) Numerical methods in engineering with Python 3. Cambridge University Press. New York.

The main focus of the book is to teach numerical methods. Nonetheless, applications in Python shows how to use existent tools to solve most of the common problems present in engineering applications.

Fritzson, P. (2004) Principles of Object Oriented Modeling and Simulation with Modelica 2.1 Wiley-IEEE Press, 2004.

Modelica is a high-level language developed specifically to solve mathematical problems represented through models. The book above describes from the first principles up to advanced features how Modelica and its object-oriented features can be used to solve problems in biology, physics, mathematics and engineering using simple but precise tools present in this special language.

REFERENCES

1. Gall J. 1986. Systemantics: How systems really work and how they fail. Second Edition. Ann Arbor, MI: The General Systemantics Press.
2. Booch, G. Maksimchuk, R. A. Engle, M. W. Young, B. J. Conallen, J. Houston, K. A. 2007. Object-Oriented Analysis and Design with Applications Third Edition. Addison-Wesley.
3. Firesmith, D. G. (1993). Object-oriented requirements analysis and logical design: a software engineering approach. New York: Wiley.
4. Yourdon, E. (1994). Object-oriented systems design: an integrated approach. Englewood Cliffs, NJ: Yourdon Press.
5. Booch, G. (1996). Managing object-oriented software development. Annals of Software Engineering, 2(1), 237-258.
6. Thapa, R. S. Project Development & Management: The Object Oriented Approach.
7. Bindel, D; Goodman, J. (2009) Principles of Scientific Computing.
8. Heath, M. T. (2013) Scientific Computing – An Introductory Survey. 2nd edition. McGraw Hill.

9. Gladwell, I; Nagy, J. G; Ferguson Jr., W. E. F. (2007) Introduction to Scientific Computing. Available in: http://www.mathcs.emory.edu/~ale/NAbook_Aug_2008.pdf.
10. Laurie Williams (2004) An Introduction to the Unified Modeling Language. Available in: http://agile.csc.ncsu.edu/SEMaterials/UMLOverview.pdf
11. Lucid Chart. UML Tutorial. Available in: https://www.lucidchart.com/pages/what-is-UML-unified-modeling-language .
12. Ostermann, S., Prodan, R., & Fahringer, T. (2009, October). Extending grids with cloud resource management for scientific computing. In Grid Computing, 2009 10th IEEE/ACM International Conference on (pp. 42-49). IEEE.
13. Perez, R. E., Jansen, P. W., & Martins, J. R. (2012). pyOpt: a Python-based object-oriented framework for nonlinear constrained optimization. Structural and Multidisciplinary Optimization, 45(1), 101-118.
14. Qin, J. Fahringer, T. UML-Based Scientific Workflow Modeling. IN: Scientific Workflows - Programming, Optimization, and Synthesis with ASKALON and AWDL. pp 75-89.
15. Selic, B. (2007, May). A systematic approach to domain-specific language design using UML. In Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on (pp. 2-9). IEEE.
16. Tutorials Point. Learn UML – Unified Modelling Language. Available in: https://www.tutorialspoint.com/uml/index.htm .
17. Claudio Ferrari, Maurizio Bonafede, Maria Elina Belardinelli, LibHalfSpace: A C++ object-oriented library to study deformation and stress in elastic half-spaces, Computers & Geosciences, Volume 96, 2016, Pages 136-146, ISSN 0098-3004, http://dx.doi.org/10.1016/j.cageo.2016.08.011.
18. (http://www.sciencedirect.com/science/article/pii/S0098300416302618)
19. Dubois-Pelerin, Yves, and Thomas Zimmermann. "Object-oriented finite element programming: III. An efficient implementation in C++." Computer methods in applied mechanics and engineering

108.1 (1993): 165-183.

20. Francis, J. P. Whiteley, J. (2012)Guide to scientific computing with C++. Undergraduate Topics in Computer science. Springer

21. Jasak, Hrvoje, Henry G. Weller, and Niklas Nordin. In-cylinder CFD simulation using a C++ object-oriented toolkit. No. 2004-01-0110. SAE Technical Paper, 2004.

22. Kale, Laxmikant V., and Sanjeev Krishnan. "CHARM++: a portable concurrent object oriented system based on C++." ACM Sigplan Notices. Vol. 28. No. 10. ACM, 1993.

23. Mangani, L., et al. "Development and validation of a C++ object oriented CFD code for heat transfer analysis." ASME Summer Heat Transfer (2007).

24. Vukics, A., and H. Ritsch. "C++ QED: an object-oriented framework for wave-function simulations of cavity QED systems." European Physical Journal D--Atoms, Molecules, Clusters & Optical Physics 44.3 (2007).

25. MathWorks (2017). Object-Oriented Programming. Natick.

26. Boisvert, R. F., Moreira, J., Philippsen, M., & Pozo, R. (2001). Java and numerical computing. Computing in Science & Engineering, 3(2), 18-24.

27. Robert Sedgewick and Kevin Wayne. 2007. Introduction to Programming in Java: An Interdisciplinary Approach (1st ed.). Addison-Wesley Publishing Company, , USA.

28. IDR Solutions. The top 11 Free IDE for Java Coding, Development & Programming. Available in: https://blog.idrsolutions.com/2015/03/the-top-11-free-ide-for-java-coding-development-programming/.

29. Tutorials Point. Learn Java Programming – Java basics. Available in: https://www.tutorialspoint.com/java .

30. Adams, P. D., Grosse-Kunstleve, R. W., Hung, L. W., Ioerger, T. R., McCoy, A. J., Moriarty, N. W., & Terwilliger, T. C. (2002). PHENIX: building new software for automated crystallographic structure determination. Acta Crystallographica Section D: Biological Crystallography, 58(11), 1948-1954.

31. The Python Guru. Python Tutorial. Available in: http://thepythonguru.com/wp-content/uploads/2016/03/thepythonguru.pdf

32. Langtangen, H. P. (2016). A primer on scientific programming with Python. Heidelberg: Springer.

33. Leaver-Fay, A., Tyka, M., Lewis, S. M., Lange, O. F., Thompson, J., Jacak, R., ... & Davis, I. W. (2011). ROSETTA3: an object-oriented software suite for the simulation and design of macromolecules. Methods in enzymology, 487, 545.

34. Matplotlib documentation. Available in: https://matplotlib.org/

35. Peirce, J. W. (2007). PsychoPy—psychophysics software in Python. Journal of neuroscience methods, 162(1), 8-13.

36. Sayama, H. (2015). Introduction to the Modeling and Analysis of Complex Systems. SUNY Binghamton. ISBN 978-1-942341-09-3

37. Sukumaran, J., & Holder, M. T. (2010). DendroPy: a Python library for phylogenetic computing. Bioinformatics, 26(12), 1569-1571.

38. Bahn, S. R., & Jacobsen, K. W. (2002). An object-oriented scripting interface to a legacy electronic structure code. Computing in Science & Engineering, 4(3), 56-66.

INDEX

Object-oriented modelling for Scientific Computing

The main audience of this book are mathematics, biology, physics and engineering students interested in acquiring more knowledge in scientific computing by using object-oriented programming (OOP). OOP is present in many different programming languages. However, not all of them can be easily used in scientific computing. Therefore, in this book we show the application of OOP technique using C++, Java, Python and Matlab. These languages stand among the most popular ones to solve scientific problems, especially the numerical ones. The whole book is divided into 4 main sections. Section 1 gives a brief description of the basic concepts of OOP, terminology and the history of its development. The second section introduces scientific computing and some simple algorithms in numerical methods are presented, which lies among the most common types of problems that the scientist may face. Section 3 encompasses the major part of the book and contains the practical techniques for the development of object-oriented software solutions. The first tool presented is the Unified Modelling Language (UML), which is not a programming language, but a tool to develop the concepts in software, documenting it and making easier the implementation of necessary algorithms and methods, to satisfy the main objectives of the software. The second tool presented is the C++ programming language. In the chapter a introduction to the basic features of the language is given, and an introduction to the object-oriented features of it. The same guideline is followed for Matlab, Java, Python and Modelica. The Section 4 shows practical applications of scientific computing using object-oriented approach to solve problems. Specifically, it is presented the application of this technique to model the famous Predator-Prey model, or Lotka-Volterra model, which represents the relationship between a prey and its predator in a system under certain constraints. The second application is the description of the OpenMDAO tool for modelling and analysis of mathematical problems. The last application consists into modelling a system of tanks, starting from a functional programming and upgrading it until it reaches an object-oriented approach. Finally, it provides some suggestions of further reading. After, the references used along the book are presented.

Elaine Ferreira Avelino was born in Rio de Janeiro, Brazil. She obtained her Bsc in Forestry Engineering at the Rural University of Rio de Janeiro (UFRRJ) in 2007 and Msc. in 2012 at UFRRJ, in the area of Forestry and Environmental Sciences, with specializazion in Wood Technology. She started her career in the Secretary for Environment of Rio de Janeiro, with Urban and Environmental Planning. Elaine was a professor of Zoology, Enthomology, Forestry Parasithology and Introduction to Research at the Pitagoras University. Since 2013 she works as an international consultant for forest management and environmental licensing.

Euan Russano was born in Minas Gerais, Brazil. He is a Chemical Engineer since 2012 by the Rural University of Rio de Janeiro (UFRRJ). He obtained his Msc. in 2014 at UFRRJ, in the area of Chemical Engineering, with specializazion in Process Control. In 2014, Russano began to develop his PhD at the University Duisburg-Essen (UDE) in the field of Water Science. His carrer was initiated in a Petrobras project in the Polymers Laboratory, at UFRRJ. From 2012 to 2014 he worked in the Fluids Flow Laboratory (Petrobras/ UFRRJ) with oil well pressure control. Since 2014 he works as an research assistant at the University of Duisburg-Essen, with water systems identification and control.

ISBN 978-1-77361-281-2

ARCLER PRESS