MATHEMATICAL FOUNDATIONS FOR COMPUTER SCIENCE WITH MATLAB:

A course about computer programming fundamentals

Jairo Guerrero García

Editorial
Universidad de Nariño

êditorial
Universidad de **Nariño**

Mathematical foundations for Computer Science with MATLAB:

A course about computer programming fundamentals

Mathematical foundations for Computer Science with MATLAB:

A course about computer programming fundamentals

Jairo Guerrero García
Autor

êditorial
Universidad de **Nariño**

Guerrero García, Jairo
Mathematical foundations for computer science with MATLAB : a course about computer programming fundamentals / Jairo Guerrero García.—1ª. Ed. – San Juan de Pasto : Editorial Universidad de Nariño, 2025

116 páginas : ilustraciones, tablas

Incluye referencias bibliográficas p. 108 - 111
ISBN: 978-628-7771-82-6

1. Matemáticas aplicadas—Informática 2. MATLAB—Aplicaciones 3. Programación 4. Teoría de números 5. Matemáticas—Ciencia de la computación 6. Funciones—MATLAB 7. Algebra lineal 8. Matemáticas discretas

519.7 G934m – SCDD-Ed. 22

Universidad de Nariño
FUNDADA EN 1904

Universidad de Nariño
ACREDITADA EN ALTA CALIDAD
RESOLUCIÓN MEN 000022 - ENERO 11 DE 2023

SECCIÓN DE BIBLIOTECA

© Editorial Universidad de Nariño
© Jairo Guerrero García

ISBN: 978-628-7771-82-6

Style review: Daniel Anthony Day
Cover design and layout: Angie Gabriela Ordoñez
Edition: 1st edition.

Publication date: October 2025
San Juan de Pasto - Nariño - Colombia

Reproduction in whole or in part, by any means or for any purpose, without written permission from the author or Editorial Universidad de Nariño is prohibited

Table of Contents

[CHAPTER 1] — Preface 11
[CHAPTER 2] — The Dawn of Computing 13
[CHAPTER 3] — Mathematics for Computer Science 17
3.1 Discrete mathematics 19
3.2 Calculus 23
3.3 Linear Algebra 26
3.4 Probability Theory and Statistics 28
3.5 Number Theory 30
[CHAPTER 4] — Knowing MATLAB 37
[CHAPTER 5] — Comparisons 41
5.1 Workshop #1 – Comparisons in MATLAB 46
[CHAPTER 6] — Loops 59
6.1 Workshop #2 – Loops in MATLAB 67
[CHAPTER 7] — Functions 77
7.1 Workshop #3 – Functions ins MATLAB 83
[CHAPTER 8] — Arrays 95
8.1 Workshop #4 — Arrays in MATLAB 100
[CHAPTER 9] — Epilogue 115
[CHAPTER 10] — References 117

[CHAPTER 1] — PREFACE

This book was created as a guide to learning the mathematical foundations essential for computer programming. Programming, a core aspect of Computer Science, is deeply rooted in mathematics; in fact, Computer Science itself emerged from mathematical principles. Understanding these foundations is crucial for developing efficient and robust software solutions. For this reason, I believe MATLAB is one of the most powerful languages for scientific computing, making it an ideal tool for illustrating the mathematical concepts presented in this book.

The primary objective of this book is to support academic learning. It is specifically designed for students and educators in Computer Science who seek to strengthen their mathematical skills in the context of programming. Additionally, this book has been intentionally written in English to encourage students to engage with academic literature in the language commonly used in the field.

Throughout history, major milestones in computing have been driven by the application of mathematics to solve specific problems. This legacy remains highly relevant in Computer Science education today, which is why this book was developed. It reflects much of my experience as a professor at the University of Nariño, where I have worked extensively on the intersection of mathematics and programming.

By reading this book, students will gain a fundamental understanding of key mathematical concepts applied in programming, with a focus on problem-solving techniques using MATLAB. The book covers topics such as numerical methods, matrix operations, algorithmic thinking, and mathematical modeling, all essential for constructing software. Upon completion, readers will have a strong foundation to approach more advanced topics in scientific computing and software development, equipping them with the skills necessary to analyze and solve computational problems effectively.

Encouraging learners to type out source code manually, rather than copying and pasting, offers significant educational advantages. This practice fosters active engagement with the material, requiring learners to focus on each line and understand its function within the program. Such deliberate involvement enhances comprehension and aids in transferring knowledge from short-term to long-term memory, thereby improving retention (Brack, 2016). Moreover, typing code allows learners to encounter and resolve errors firsthand, promoting the development of debugging skills and a deeper understanding of the code's functionality. This process cultivates critical thinking and problem-solving abilities essential for proficient programming. Additionally, typing encourages mindful learning, as learners must pay attention to syntax and structure, leading to a more reflective and thorough understanding of programming concepts (Cook, 2012). This mindfulness helps grasp the code's purpose and architecture, reducing the likelihood of superficial learning that often accompanies copy-paste practices. By typing code manually, learners develop autonomy and confidence in their coding abilities, which is crucial for their growth as programmers. Incorporating these practices into programming education can lead to a more robust and meaningful learning experience, equipping learners with the skills and understanding necessary for success in the field.

I use MATLAB throughout this textbook. MATLAB is a trademark of The MathWorks.

—The Author,

Pasto, February 19, 2023

Although computing was practically coined in the 20th century, there are traces of computing that existed long ago. Various forms of computing existed before the 20th century, although they were very different from modern digital computers. Some examples include:

- Abacus: The abacus is a simple device for performing arithmetic operations. It consists of beads or stones on rods, and users move the beads to perform calculations *(Maričić & Lazić, 2020*).
- Slide Rule: The slide rule is a mechanical device for multiplication and division. It consists of two logarithmic scales that slide past each other (*Ulmann, 2022*).
- Jacquard Loom: The Jacquard loom, invented in the early 19th century, used punched cards to control the weaving of complex patterns in textiles. It is one of the earliest examples of a machine that used programming (*Albaugh et al., 2021*).
- Charles Babbage's Analytical Engine: In the mid-19th century, Charles Babbage, a British mathematician and inventor, designed the Analytical Engine. Although it was never built, it was a mechanical computer that could perform basic arithmetic operations, store data, and execute instructions (*Haecker, 2022*).

These early forms of computing were used primarily for performing arithmetic calculations and automating simple mechanical tasks. It was not until the mid-20th century when the first digital computers were invented, that computing became a field of study and a major technological force worldwide.

Historically, Computer Science was born from mathematics (*O'Regan, 2018*). Unfortunately, human history is associated with wars. The 20th century was a special case, with two world wars. Sometimes, history plays with some irony in the plot of human lives; the more suffering, the more technological developments are produced.

Technological developments are key to the evolution of civilization. Scientists and scholars are duty-bound to produce innovative ideas to promote such developments, and computing is no exception. Calculations are crucial to winning a war because computing relies on being able to count, which is a form of mathematics. Developers need to understand and work with numbers to create algorithms and software. Many computing concepts are also based on mathematical principles, such as logic and set theory.

While the war was still raging, Alan Turing and his team at Bletchley Park in England worked on cracking the German ENIGMA code. This was a significant turning point in the war, allowing the Allies to intercept and decode German communications, giving them a considerable advantage. All these technological advances were achieved thanks to mathematics (*Bowen, 2019*).

After the war, Alan Turing continued his work in computing and became the father of computer science. He also made significant contributions to artificial intelligence. Claude Shannon, an American mathematician, electrical engineer, and cryptographer, is known as the "father of information theory." Shannon is credited with inventing the binary code used in modern computer systems and developing digital circuit design theory. All of his contributions are based on mathematics (*Magoun, 2019).*

The traditional computing paradigm is based on the Von Neumann architecture, which John Von Neumann proposed in the late 1940s. This architecture is used in most conventional digital computers. It is based on a sequential processing model that involves four main components: the input/output devices, the memory, the arithmetic/logic unit (*ALU),* and the control unit *(O'Regan, 2013).*

In this model, data and instructions are stored in memory. The control unit fetches instructions from memory, decodes them, and executes them by sending signals to the ALU. The ALU performs arithmetic and logic operations on data as directed by the control unit, and the results are then stored back in memory.

The Von Neumann architecture is a "stored program" model, which means that a computer program's instructions are stored in the same memory as the data. This allows for great flexibility in programming, as programs can be modified and rewritten as needed (*Macrae, 2019*).

While the traditional computing paradigm has been highly successful and is still widely used today, it has some limitations. These include its reliance on sequential processing and the need to move data between the memory and the processor, which can slow down performance. Newer computing paradigms, such as parallel computing and quantum computing, have emerged to address these limitations and enable new computing applications and capabilities (*National Academies of Sciences, Engineering, and Medicine, 2019*).

[CHAPTER 3] — MATHEMATICS FOR COMPUTER SCIENCE

Computer Science is the study of computers and computational systems. It involves both theoretical and practical aspects of computing, including algorithms, programming languages, data structures, software engineering, computer architecture, artificial intelligence, databases, human-computer interaction, and computer networking (*ACM & IEEE-CS, 2020*). Computer science is a broad field that encompasses many subfields and specializations, such as cybersecurity, computer graphics, computer vision, machine learning, robotics, and natural language processing. It is also a rapidly evolving field, with innovative technologies and applications constantly emerging.

Computer science has become increasingly important today, as computers have become ubiquitous and are used in almost every aspect of our lives. It is a vital discipline for anyone interested in developing software, designing computer systems, or working with technology in any capacity. The origin of Computer Science can be traced back to the mid-20th century when the first digital computers were being developed. During this time, researchers were developing algorithms, programming languages, and other foundational concepts that would define the field of Computer Science (*Woodford, 2021*).

One of the earliest pioneers of Computer Science was Alan Turing, a British mathematician and computer scientist who played a key role in cracking the German Enigma code during World War II. After the war, Turing continued to work on developing computer systems and artificial intelligence, and his ideas helped shape the direction of the field. Another important figure in the history of Computer Science was John von Neumann, a Hungarian-American mathematician and physicist who worked on developing early electronic computers and is credited with designing the von Neumann architecture, a key model for modern computer systems.

Other significant early contributors to Computer Science include Grace Hopper, who developed the first compiler for a computer programming language, and Ada Lovelace, who is often considered the world's first computer programmer for her work on Charles Babbage's Analytical Engine. Since its early days, Computer Science has continued to evolve and expand, with modern technologies and applications constantly emerging. Today, it is a vital discipline that underpins many areas of contemporary society, from business and industry to healthcare, science, and beyond (*Pucik, 2022*).

Undoubtedly, mathematics is the theoretical basis on which Computer Science is based. Considering that mathematics is broad, there are concepts on which the pioneers were based to create Computer Science. Computer Science relies heavily on various mathematical disciplines to build and analyze computational systems. Some of the key areas of mathematics that are used in Computer Science include:

- **Discrete Mathematics** is a branch of mathematics that deals with discrete structures, such as sets, graphs, and integers. It is used in Computer Science to analyze algorithms, data structures, and computational models *(Aigner, 2023).*
- **Calculus:** Calculus is a branch of mathematics that deals with continuous change and motion. It is used in Computer Science to analyze algorithms' performance and model physical systems, such as computer networks *(Kidron, 2020).*
- **Linear Algebra:** Linear algebra is a branch of mathematics that deals with vectors and matrices. It is used in Computer Science for data analysis, machine learning, and computer graphics *(Trefethen & Bau, 2022*).
- **Probability Theory and Statistics**: Probability theory and statistics are used in computer science to model uncertainty and analyze data. They are used in machine learning, data mining, and artificial intelligence *(Potters & Bouchaud, 2020).*

- **Number Theory:** Number theory is a branch of mathematics that deals with the properties of integers. It is used in Computer Science for cryptography, data compression, and error-correcting codes *(Baker, 2022)*.

These are just a few examples of the many mathematical disciplines used in Computer Science. Computer scientists can develop new technologies and solutions that continue transforming the world by combining mathematical theory with practical applications.

3.1 DISCRETE MATHEMATICS

Discrete mathematics is a branch of mathematics that deals with discrete or countable objects, such as integers, graphs, and sets. Unlike continuous mathematics, which deals with constant objects, such as real numbers and calculus, discrete mathematics focuses on objects that a finite set of values can represent.

Discrete mathematics is used extensively in Computer Science and other fields that involve discrete structures and computational algorithms. Some of the key topics in discrete mathematics include:

- **Combinatorics** is the study of discrete structures and their properties, such as graphs, sets, and permutations. It is used in Computer Science for data analysis, optimization, and algorithm design *(West, 2020)*.

  Here's an example of using combinatorics in MATLAB to generate all possible combinations of elements in a set:

  Suppose we have a set of numbers {1, 2, 3, 4} and want to generate all possible combinations of two elements from this set.

  In MATLAB, we can use the "combnk" function to generate all possible combinations of k elements from a set of n elements. Here's how we can use it for our example:

```
% Define the set
set = [1,2,3,4];

% Generate all possible combinations of two elements
combinations = nchoosek(set, 2)
```

After running this code, the "combinations" variable will contain a 6 x 2 matrix, where each row represents a different combination of two elements from the set:

```
combinations =
     1     2
     1     3
     1     4
     2     3
     2     4
     3     4
```

In this way, we can use combinatorics functions in MATLAB to generate all possible combinations of elements in a set, which is helpful for many applications in Computer Science, mathematics, and other fields.

- **Graph Theory:** This is the study of graphs, which are mathematical structures that represent relationships between objects. Graph theory is used in Computer Science for network analysis, data mining, and algorithm design *(Sporns, 2022).*

  Here's an example of using graph theory in MATLAB to analyze a graph:

  Suppose we have a graph with five nodes and six edges, represented by the adjacency matrix A:

```
A = [0 1 1 0 0;   1 0 0 1 0;   1 0 0 1 0;  0 1 1 0 1;  0 0 0 1 0];
```

We can use the built-in MATLAB function "graph" to create a graph object from the adjacency matrix A:

```
G = graph(A);
```

Once we have the graph object, we can use various built-in functions to analyze its properties. For example, we can use the "degree" function to calculate the degree of each node in the graph:

```
deg = degree(G);
```

The "deg" variable will contain a vector with the degree of each node:

```
deg =
     2
     2
     2
     3
     1
```

We can also use the "shortestpath" function to find the shortest path between two nodes in the graph. For example, to find the shortest path between nodes 1 and 5:

```
path = shortestpath(G, 1, 5);
```

The "path" variable will contain a vector with the nodes along the shortest path:

```
path =
     1     2     4     5
```

In this way, we can use graph theory functions in MATLAB to analyze the properties of a graph, which is helpful for many applications in Computer Science, mathematics, and other fields.

- **Set Theory**: This studies sets and their properties, such as cardinality and intersection. Set theory is used in Computer Science for database design, algorithm analysis, and data modeling *(Kumar et al., 2019).*

  Here's an example of using set theory in MATLAB to perform set operations:

  Suppose we have two sets of numbers, A and B, and we want to perform set operations using MATLAB.

  We can represent the sets using MATLAB arrays. For example, let:

```
A = [1, 2, 3, 4, 5];
B = [4, 5, 6, 7, 8];
```

  To find the union of A and B, we can use the "union" function:

```
C = union(A, B);
```

  The "C" variable will contain the union of A and B:

```
C =
     1     2     3     4     5     6     7     8
```

  To find the intersection of A and B, we can use the "intersect" function:

```
D = intersect(A, B);
```

  The "D" variable will contain the intersection of A and B:

```
D =
     4     5
```

  To find the elements that are in A but not in B, we can use the "setdiff" function:

```
D = intersect(A, B);
```

The "E" variable will contain the elements that are in A but not in B:

```
E =
     1     2     3
```

In this way, we can use set theory functions in MATLAB to perform set operations on sets of numbers or other data types, which is helpful for many applications in Computer Science, mathematics, and other fields.

Discrete mathematics is a fundamental part of Computer Science and provides the theoretical foundation for many important field study areas. Computer scientists can develop algorithms, data structures, and other computational tools critical to modern computing by understanding the principles and concepts of discrete mathematics.

3.2 CALCULUS

Calculus was independently developed by two mathematicians, Sir Isaac Newton and Gottfried Wilhelm Leibniz, in the 17th century.

Newton is credited with developing the foundations of differential calculus in the 1660s while working on problems related to motion and forces. Leibniz, a German mathematician and philosopher, created his version of calculus in the late 1670s, called "infinitesimal calculus." Both men published their work on calculus in the 1680s, and there was initially some controversy over who deserved credit for inventing the subject *(Kossovsky, 2020).*

Today, it is generally recognized that Newton and Leibniz played a significant role in developing calculus and that their work was complementary rather than competitive. The two approaches to calculus, Newtonian and Leibnizian, are still in use today and have led to a rich body of mathematical theory and practical applications. Calculus is a branch of mathematics that deals with rates of change and how things change over time. It is divided into two main branches: differential calculus and integral calculus.

Differential calculus deals with studying the rate of change of a function at a point and calculating derivatives. Derivatives are used to find the slope of a

function at a given point and its maximum and minimum points. They are used in various fields, including physics, engineering, economics, and others *(Sprunger & Jacobs, 2019)*.

Integral calculus deals with the study of quantity accumulation and the calculation of integrals. Integrals are used to find the total quantity that has accumulated over time, and they have various applications, such as finding the area under a curve or the volume of a three-dimensional shape.

Differential and integral calculus provide a robust set of tools for analyzing and understanding how things change over time and are used extensively in many fields of science, engineering, and mathematics. There is also a close relationship with the concept of Infinitesimal calculus aside from differential and integral calculus.

Infinitesimal calculus is an older term for the branch of mathematics that deals with limits, derivatives, and integrals. Differential and integral calculus are two subfields of infinitesimal calculus that study functions and their derivatives and integrals *(Martínez et al., 2020)*.

Differential calculus involves studying the rate of change of a function at a point and calculating derivatives. Derivatives are used to find the slope of a function at a given point and to determine its maximum and minimum points.

Integral calculus, conversely, studies the accumulation of quantities and calculates integrals. Integrals are used to find the total amount that has accumulated over time, and they have various applications, such as finding the area under a curve or the volume of a three-dimensional shape.

In summary, differential and integral calculus are subfields of infinitesimal calculus that study functions and their derivatives and integrals. Differential calculus deals with the rate of change of a function, while integral calculus deals with the accumulation of a quantity over time. Together, these two fields provide a robust set of tools for analyzing and understanding how things change over time and are used extensively in many fields of science, engineering, and mathematics *(Kirchner, Benzmüller & Zalta, 2019)*.

Here's an example of how to use MATLAB to perform some basic operations of differential calculus:

Suppose we want to find the derivative of a simple function, such as f(x) = x^2, at a specific point, say x = 2. We can use MATLAB's symbolic toolbox to define the function, find the derivative, and evaluate it at x = 2. Here's how to do it:

Define the function using the "syms" command to create a symbolic variable x:

```
syms x
f = x^2;
```

Find the derivative of the function using the "diff" command:

```
df = diff(f)
```

This will output the derivative of the function, which is 2x.

Evaluate the derivative at a specific point, such as x = 2, using the "subs" command:

```
df_value = subs(df, x,2)
```

This will output the value of the function's derivative at x = 2, which is 4.

So, in this example, we used MATLAB's symbolic toolbox to define a function, find its derivative, and evaluate it at a specific point, all using the principles of differential calculus.

Now, here's another example of how to use MATLAB to perform some basic operations of integral calculus:

Suppose we want to find the definite integral of a simple function, such as f(x) = x^2, over a specific interval, say from x = 0 to x = 1. We can use MATLAB's symbolic toolbox to define the function, find the integral, and evaluate it over the given interval. Here's how to do it:

Define the function using the "syms" command to create a symbolic variable x:

```
syms x
f = x^2;
```

Find the indefinite integral of the function using the "int" command:

```
F = int(f)
```

This will output the indefinite integral of the function, which is (1/3)x^3 + C, where C is a constant of integration.

Evaluate the definite integral of the function over a specific interval, such as from x = 0 to x = 1, using the "subs" command:

```
F_value = subs(F, x, 1) - subs(F, x, 0)
```

This will output the value of the function's definite integral over the given interval, which is 1/3.

So, in this example, we used MATLAB's symbolic toolbox to define a function, find its indefinite integral, and evaluate its definite integral over a specific interval, all using the principles of integral calculus.

3.3 LINEAR ALGEBRA

Linear algebra is a branch of mathematics that studies linear equations, vectors, matrices, and linear transformations. It is a fundamental tool used in many fields, including physics, engineering, Computer Science, economics, and statistics.

Linear algebra involves the study of the properties of linear transformations and their representation by matrices. It also deals with solving systems of linear equations, which arise in many applications, and with the study of vector spaces and their subspaces. Some of the key concepts in linear algebra include linear independence, basis, dimension, inner product, determinants, eigenvectors, and eigenvalues *(Farin & Hansford, 2021).*

Linear algebra has a wide range of applications in different areas of science and engineering. For example, it is used in computer graphics to represent and manipulate 3D images, in data analysis to perform principal component analysis, in control theory to model and analyze dynamic systems, in cryptography to encrypt and decrypt messages, and in machine learning to perform linear regression and other types of statistical analysis.

Here's an example of how to use MATLAB to perform some basic operations of linear algebra:

Suppose we want to solve a system of linear equations, such as the following:

$$2x + 3y = 7$$

$$4x + 5y = 11$$

We can use MATLAB to define the system of equations as a matrix equation, and then use the backslash operator to solve it. Here's how to do it:

Define the system of equations as a matrix equation:

```
A = [2 3; 4 5];
b = [7; 11];
```

Here, we have defined the coefficient matrix A and the right-hand side vector b.

Solve the system of equations using the backslash operator:

```
x = A\b
```

This will output the solution vector x, which satisfies the system of equations.

In this example, the solution vector x is [-1; 3], which means that the solution to the system of equations is x = -1 and y = 3.

So, in this example, we used MATLAB to define a system of linear equations as a matrix equation, and then solved it using the backslash operator, all using the principles of linear algebra.

3.4 PROBABILITY THEORY AND STATISTICS

Probability theory and statistics are two branches of mathematics that are closely related and often studied together.

- **Probability theory** deals with the study of random events and the likelihood of their occurrence. It provides a framework for quantifying uncertainty and making predictions in situations where the outcome is not known in advance. Some of the key concepts in probability theory include probability distributions, expected values, variance, and random variables (Matloff, 2019).

  Here's an example of how to use MATLAB to simulate a random variable with a certain probability distribution:

  Suppose we want to simulate a random variable X that follows a normal (Gaussian) distribution with mean 0 and standard deviation 1. We can use the "randn" function in MATLAB to generate random numbers from a standard normal distribution, and then transform them to the desired distribution using the mean and standard deviation.

  Here's how to do it:

  Generate a vector of random numbers from a standard normal distribution:

```
 n = 1000; % number or samples
 x = randn(n, 1); % generate n samples from a standard normal
distribution
```

  Transform the vector to a normal distribution with mean 0 and standard deviation 1:

```
 mu = 0 % mean
 sigma = 1; % standard deviation
 y = mu + sigma*x; % transform to normal distribution
```

Plot the histogram of the simulated random variable:

```
histogram(y, 'Normalization', 'pdf'); % plot the histogram
xlabel('x'); ylabel('f(x)'); % set the axis labels
```

This will plot the histogram of the simulated random variable X, which should look like a normal distribution with mean 0 and standard deviation 1.

In this example, we used MATLAB to simulate a random variable with a normal distribution, which is a fundamental concept in probability theory.

- **Statistics**, on the other hand, is the study of collecting, analyzing, and interpreting data. It involves the application of probability theory to real-world situations and the use of statistical models to make predictions and draw conclusions from data. Some of the key concepts in statistics include hypothesis testing, confidence intervals, regression analysis, and experimental design *(Avella-Medina, 2020).*

  Here's an example of how to use MATLAB to perform a t-test on two samples within the world of statistics:

  Suppose we have two samples of data, and we want to test whether their means are significantly different from each other. We can use a t-test to determine whether the difference between the sample means is statistically significant.

  Here's how to do it:

  Define the two samples of data:

```
x = [1, 2, 3, 4, 5];
y = [4, 5, 6, 7, 8];
```

Calculate the means and standard deviations of the two samples:

```
mean_x = mean(x);
std_x = std(x);
mean_y = mean(y);
std_y = std(y);
```

Perform a two-sample t-test to compare the means of the two samples:

```
[h, p, ci, stats] = ttest2(x, y);
```

This will perform a two-sample t-test and output the test results, including the test statistic (in the "stats" variable), the p-value (in the "p" variable), and the confidence interval for the difference in means (in the "ci" variable).

In this example, the t-test may reveal whether the difference in means between the two samples is statistically significant or not, and helps us make conclusions about the underlying populations from which the samples were drawn. The t-test is a fundamental tool in statistical inference, and is widely used in many fields including medicine, social sciences, and engineering.

Probability theory and statistics are used in many fields, such as physics, engineering, finance, medicine, and social sciences. They are essential tools in data science and machine learning, as they provide a foundation for understanding and analyzing complex data sets.

In summary, probability theory and statistics are two branches of mathematics that are fundamental to understanding uncertainty and making predictions in a wide range of applications.

3.5 NUMBER THEORY

Number theory is a branch of mathematics that deals with the properties of numbers, especially integers. It is one of the oldest and most fundamental areas of mathematics, and has applications in many other fields such as cryptography, Computer Science, and physics *(Miller & Takloo-Bighash, 2021).*

Some of the central topics in number theory include:

- **Prime numbers:** the study of the properties and distribution of prime numbers, which are integers that can only be divided by 1 and themselves.

  Here's an example of how to use MATLAB to generate prime numbers:

  MATLAB provides the isprime function, which tests whether a given number is prime or not. We can use this function to generate a list of prime numbers up to a certain limit.

  For example, to generate all prime numbers less than 100:

```
primes = [];
for i = 2:99
     if isprime(i)
           primes = [primes, i];
     end
end
```

  This code initializes an empty array called "primes", and then loops overall numbers from 2 to 99. For each number, it checks whether it is prime using the isprime function, and if so, adds it to the "primes" array.

  After running this code, the "primes" array will contain all prime numbers less than 100, we have an excerpt below:

```
primes =

  Columns 1 through 11

     2     3     5     7    11    13    17    19    23    29    31

  Columns 12 through 22

    37    41    43    47    53    59    61    67    71    73    79

  Columns 23 through 25

    83    89    97
```

This is just one example of how to generate prime numbers in MATLAB using the isprime function. There are many other ways to generate prime numbers using MATLAB as well, depending on the specific requirements of your problem.

- **Diophantine equations:** the study of equations in which we seek integer solutions, such as the famous Pythagorean equation a^2 + b^2 = c^2.

  Here's an example of how to use MATLAB to solve a Diophantine equation:

  Let's say we want to find all integer solutions to the equation:

  $$5x + 12y = 37$$

  This is a linear Diophantine equation, which means that we are looking for integer solutions to a linear equation. To solve this equation in MATLAB, we can use the gcd function, which computes the greatest common divisor of two numbers.

  We can start by finding the gcd of 5 and 12:

```
gcd(5, 12)
```

  This gives us a result of 1, which means that 5 and 12 are coprime (i.e., they have no common factors other than 1). This is good news, because it means that there are integer solutions to the equation.

  To find the solutions, we can use the extended Euclidean algorithm to find the coefficients of 5 and 12 that add up to 1:

```
[u, v, gcd] = gcd(5, 12)
```

  This gives us u = 5, v = -2, and gcd = 1. These coefficients tell us that:

  $$5 * (5) + 12 * (-2) = 1$$

Now we can use these coefficients to find all integer solutions to the equation 5x + 12y = 37. We start by finding one particular solution by setting y = 0:

```
x = 5*37 + 12*k
y = k
```

where k is any integer. For example, if we set k = 0, we get x = 185 and y = 0.

To find all integer solutions, we can add a multiple of the coefficients u and v to this particular solution. For example, if we add u = 5 to x and v = -2 to y, we get:

```
x = 5*37 + 12*k + 5
y = k - 2
```

This gives us another solution to the equation. We can continue adding multiples of u and v to find all integer solutions. For example, if we add 2u = 10 and 2v = -4, we get:

```
x = 5*37 + 12*k + 15
y = k - 4
```

And so on. In this way, we can use MATLAB to find all integer solutions to the Diophantine equation 5x + 12y = 37. The specific implementation will depend on the equation you are trying to solve.

- **Modular arithmetic**: the study of arithmetic operations performed "modulo" a certain number, which is a way of understanding periodic or repeating patterns in arithmetic.

  Here's a short example of how to use modular arithmetic in MATLAB:

  Suppose we want to calculate the remainder when 100 is divided by 7. We can use the modulo operator (mod) to perform modular arithmetic:

```
remainder = mod(100, 7)
```

The result will be 2, which is the remainder when 100 is divided by 7.

Another example is computing the inverse of a number modulo a given modulus. Suppose we want to find the inverse of 3 modulo 7, which is a number x such that $3x \equiv 1 \pmod 7$. We can use the modinv function from the Symbolic Math Toolbox to compute this inverse:

```
x = modinv(sym(3), sym(7))
```

The sym function is used to convert the integers 3 and 7 to symbolic objects, which are required by the modinv function. The result will be a symbolic fraction 5/21, which is the inverse of 3 modulo 7. If we want to convert this to a regular fraction, we can use the double function:

```
double(x)
```

This will output 0.2381, which is approximately equal to 5/21.

- **Cryptography:** the study of encoding and decoding messages using mathematical algorithms, often based on number theory.

  Here's a short example of how to use cryptography in MATLAB:

  Suppose we want to encrypt a message using the simple Caesar cipher, which involves shifting each letter of the message by a fixed number of positions in the alphabet. We can use the double and char functions in MATLAB to convert between characters and their ASCII codes.

  Here's an example of how to encrypt the message "HELLO WORLD" using a shift of 3:

```
message = 'HELLO WORD'; % the message to be encrypted
shift = 3; % the number of positions to shift the letters

% convert the message to ASCII codes and apply the shift
encrypted = mod(double(message)- 65 + shift, 26) + 65;

% convert the encrypted ASCII codes back to characters
result = char(encrypted);
```

The result will be the encrypted message "KHOOR ZRUOG".

To decrypt the message, we simply apply the reverse shift:

```
shift = -3; % the reserve shift

% apply the reverse shift and convert back to characters
decrypted = char(mod(double(result) - 65 + shift, 26) + 65);
```

The result will be the original message "HELLO WORLD".

Number theory has a rich history, with contributions from many famous mathematicians such as Euclid, Euler, Gauss, and Riemann. It is an active area of research today, with many open problems and connections to other areas of mathematics and science.

MATLAB (short for "matrix laboratory") is a programming language and numerical computing environment that was created by Cleve Moler in the late 1970s. Moler was a professor of mathematics at the University of New Mexico, and he created MATLAB to provide a way for his students to do numerical calculations on a computer (Fortuna, Frasca & Buscarino, 2021).

The first version of MATLAB was written in Fortran and ran on the minicomputers that were commonly used in academic and research environments at the time. In 1984, Moler and his colleagues formed a company called The MathWorks to commercialize MATLAB, and they rewrote the software in C to improve its performance and portability.

Over the years, MATLAB has evolved to become a powerful tool for numerical computation, data analysis, visualization, and programming. The software includes a large library of mathematical functions and toolboxes for specific applications, such as signal processing, control systems, and image processing. MATLAB is also widely used in academic research and in industries such as engineering, finance, and science.

Today, MATLAB is distributed by The MathWorks and is used by millions of engineers, scientists, and students around the world. Its popularity is due to its ease of use, powerful mathematical and visualization capabilities, and the large community of users who contribute to its development and support.

According to The MathWorks—the producer—MATLAB is a programming and numeric computing platform used by millions of engineers and scientists to analyze data, develop algorithms, and create models (*2023*).

Engineers and scientists need a programming language that allows them to directly express mathematical matrices and vector arrays. Linear algebra in MATLAB is learned and looks like an academic textbook. The same is true for data analysis, signal and image processing, control design, and other applications.

Everything about MATLAB is designed specifically for engineers and scientists, because: function names are familiar and easy to remember, the desktop environment is tuned for scientific and engineering workflows. The documentation is written for engineers and scientists, not computer scientists.

MATLAB toolboxes offer professionally developed, rigorously tested, and fully documented functionality for a wide range of scientific and engineering applications. Toolboxes are designed to work together, integrating with parallel computing environments, GPUs, and C code generation.

MATLAB Apps are interactive applications that combine direct access to large collections of algorithms with immediate visual feedback. You can instantly visualize how the different algorithms work with your data. Iterate until you get the expected results, then automatically generate MATLAB code to reproduce or automate your work.

Major engineering and scientific challenges require extensive coordination across teams to bring ideas to implementation. Each passing of information along the way adds errors and delays. MATLAB helps automate the entire path from research to production by allowing you to:

Connecting: MATLAB allows you to connect with more than 1,000 hardware devices.

Analyzing: Integrating MATLAB into Production Environments

Scaling: MATLAB Runs Algorithms Faster and with Big Data by Scaling to Clusters, the Cloud, and GPUs.

Simulating: Connectivity to Simulink and State flow for model-based design and simulation.

MATLAB does the work of making your code faster. Math operations are distributed throughout your computer's kernels, library calls are highly optimized, and all code is compiled.

Engineers and scientists rely on MATLAB to send a spacecraft to Pluto, match patients needing organ transplants with donors, or simply generate a business report. A team of MathWorks engineers continually verifies software quality by running millions of tests against the MATLAB codebase every day.

Considering the above, the first step is recognizing the main environment of MATLAB to do the first steps into MATLAB programming. Now, MATLAB is available for Microsoft Windows, Mac, and Linux. Regardless the operating system, MATLAB has the same integrated development environment as this:

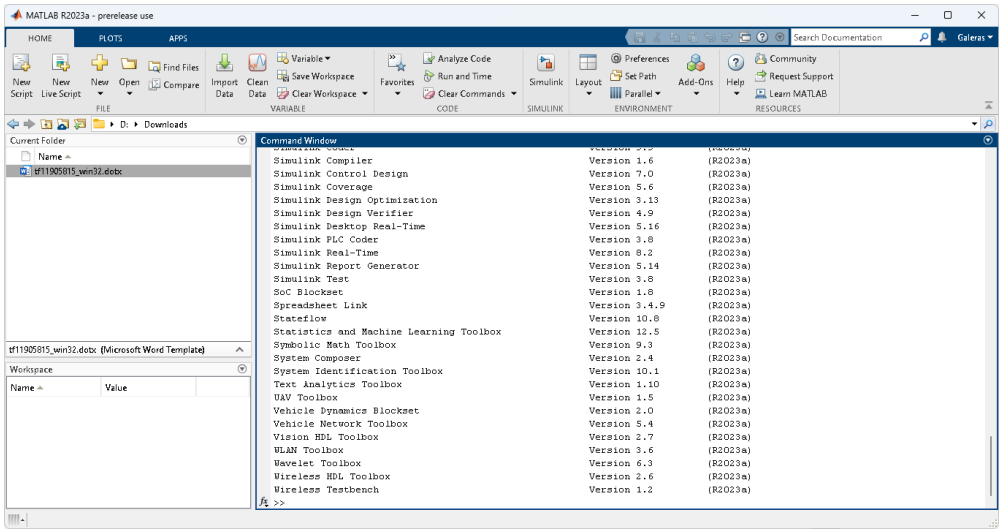


Figure 4-1. MATLAB main screen

Four main areas are depicted in Figure 4-1. The first area is the Ribbon: it is at the top of the screen. The Ribbon is a collection of common functionalities and tools for working with MATLAB. As default, the Ribbon has 3 tab strips: HOME, PLOTS, and APPS. The first one includes the main functionalities related to the

daily work with MATLAB. The second one is oriented to the graphical representation based on plotting. The third one is related to the collection of toolboxes installed in the system.

The second area is in the top-left location, and it is dedicated to the management of the file system. In such an area, users can explore the file system for interacting with files and folders.

The third area is in the bottom-left location of the screen, and it is called Workspace. The Workspace is representation of the entities created in the memory of the computer. Every single entity as variables—scalars or matrices—will be available as a list in the Workspace. The Workspace allows users exploring the values of the variables used in calculations.

Perhaps the most important area in the main screen is the Command Window, which is the greatest one on the screen. It is possible that the configuration of the main screen can vary according to the screen resolution; however, the areas are always present regardless of the selected layout. The Command Window is the entry point of statements written in MATLAB language; with such a window, users can interact with the system to do some calculations and starting programming with the language.

One of the first steps in computing is understanding Boole's algebra. Boole's algebra, also known as Boolean algebra, is a branch of algebra that deals with logical operations and the manipulation of logical values (Rushdi, 2023). It was developed by the English mathematician and philosopher George Boole in the mid-19th century. The basic idea of Boolean algebra is to represent logical values—such as true and false, or 0 and 1—as symbols, and to define logical operations—such as AND, OR, and NOT—that can be performed on these symbols. These operations can then be used to manipulate logical expressions and to simplify them.

For example, in Boolean algebra, the logical AND operation is represented by the symbol "∧", and the logical OR operation is represented by the symbol "∨". Using these symbols, we can represent logical expressions and perform operations on them. Here are some examples:

(A ∧ B) ∨ (C ∧ D) : This is a logical expression that represents the OR operation between two AND operations. It means that either both A and B are true, or both C and D are true (or both).

¬(A ∨ B): This is a logical expression that represents the NOT operation on the OR operation. It means that A OR B is false.

A truth table is a table that shows the possible combinations of truth values—typically true or false, represented by 1 or 0—for a set of propositions, and the resulting truth value of a compound proposition formed from those propositions using logical operators. Truth tables are commonly used in logic, mathematics, and Computer Science to determine the truth value of complex logical expressions. The Table 1. below shows a truth table based on 3 logical operators:

Table 5-1. Truth table based on 3 logical operators

P	Q	NOT P	NOT Q	P AND Q	P OR Q
False	False	True	True	False	False
False	True	True	False	False	True
True	False	False	True	False	True
True	True	False	False	True	True

In a truth table, the columns correspond to the input propositions and the output, and the rows correspond to the possible combinations of input truth values. The output column represents the truth value of the logical expression formed from the input propositions.

In addition to its theoretical importance in the foundations of mathematics and logic, Boolean algebra has practical applications in Computer Science, where it is used in digital circuits and in the design of computer programs. In fact, Boolean algebra is the foundation of digital electronics and is used to design, build, and analyze digital systems.

Is a fundamental concept in computing and digital electronics. It provides a way to reason about logic and binary data in a formal, mathematical way, and allows us to manipulate and analyze digital signals, such as those that represent the states of electronic switches in a computer circuit.

Here are some of the keyways in which Boolean algebra is important in computing:

- **Logic design:** Boolean algebra is used to design digital circuits and systems, such as CPUs, memory, and input/output devices. The principles of Boolean algebra allow us to design circuits that perform logical operations, such as AND, OR, and NOT, that are fundamental to digital electronics (*Roth Jr, Kinney & John, 2020*).

- **Computer programming:** Many programming languages, such as C, C++, Java, and Python, include Boolean data types—usually represented as true/false or 1/0—and logical operators—such as &&, ||, and !—that are based on Boolean algebra. These operators are used to control program flow and make decisions based on conditions (Tissenbaum, Sheldon & Abelson, 2019).

- **Digital signal processing:** Many signal processing algorithms, such as filters, modulators, and demodulators, are based on Boolean algebraic concepts. For example, a digital filter might use Boolean operations to combine input samples and produce an output signal (*Steiglitz, 2020*).

- **Database design:** Boolean algebra can be used to design database queries that involve logical operations such as AND, OR, and NOT. These queries can be used to retrieve specific data from a database based on certain conditions (Mount & Zumel, 2019).

Overall, Boolean algebra is a powerful and essential tool for working with digital signals and logic in computing. Its concepts and principles are used throughout Computer Science and engineering to design and analyze digital systems, and to develop software and algorithms that manipulate and process digital data.

In computer programming, a comparison is an operation based in Boolean Algebra that is used to compare two values or expressions and determine their relationship. The result of a comparison is usually a Boolean value, which is either true or false, depending on the outcome of the comparison.

There are several types of comparisons that can be performed in computer programming, including:

- **Equality comparison:** This type of comparison checks if two values are equal. In many programming languages, the equality operator is written as ==.

- **Inequality comparison:** This type of comparison checks if two values are not equal. In many programming languages, the inequality operator is written as !=.

- **Greater-than comparison:** This type of comparison checks if one value is greater than another. In many programming languages, the greater-than operator is written as >.
- **Less-than comparison:** This type of comparison checks if one value is less than another. In many programming languages, the less-than operator is written as <.
- **Greater-than-or-equal-to comparison:** This type of comparison checks if one value is greater than or equal to another. In many programming languages, the greater-than-or-equal-to operator is written as >=.
- **Less-than-or-equal-to comparison:** This type of comparison checks if one value is less than or equal to another. In many programming languages, the less-than-or-equal-to operator is written as <=.

These comparison operators can be used in various contexts in programming, such as in conditional statements (e.g., if statements) and loops. Comparisons are a fundamental concept in programming and are used extensively in many programming tasks. In MATLAB, you can use comparison operators to compare values and produce logical (true/false) results. Here are some of the most used comparison operators in MATLAB:

>: Greater than

<: Less than

>=: Greater than or equal to

<=: Less than or equal to

==: Equal to

~=: Not equal to

The following MATLAB script prompts the user to input their name and age, and then provides a message based on the age input, it's an example of using comparisons in MATLAB. The code block checks the value of the age variable using conditional statements. If the age is less than 0 or greater than

130, it displays an error message. If the age is 18 or greater, it displays a message indicating that the user is an adult. Otherwise, it displays a message indicating that the user is not an adult.

```
%-----------------------------------------------------
% >>> THE AGE OF A PERSON <<<
% by Jairo Guerrero, University of Nariño
%-----------------------------------------------------
age = 0;
name = '';
%-----------------------------------------------------
disp('>>> THE AGE OF A PERSON <<<');
name = input('What is your name? ', 's');
age = input('How old are you? (years) ');
if age < 0 || age > 130
    disp('Oops! wrong age...');
elseif age >= 18
    disp(['Hi ', name, '. You are an adult']);
else
    disp(['Hi ', name, '. Your are NOT an adult']);
end
%-----------------------------------------------------
```

Here we have another example using comparisons. Let's create a MATLAB script for calculating the area of a triangle based on its semi perimeter.

```
%-----------------------------------------------------
% >>> THE AREA OF A TRIANGLE ACCORDING TO ITS SIDES <<<
% by Jairo Guerrero, University of Nariño
%-----------------------------------------------------
a = 0.0;
b = 0.0;
c = 0.0;
p = 0.0;
s = 0.0;
%-----------------------------------------------------
```

```
disp('>>> THE AREA OF A TRIANGLE ACCORDING TO ITS SIDES <<<')
disp('')
a = input('Type the side a: ');
b = input('Type the side b: ');
c = input('Type the side c: ');
p = (a + b + c) / 2;
if (p * (p - a) *(p - b) * (p - c)) <= 0
    disp('Such a triangle does NOT exist...')
else
    s = sqrt(p * (p - a) *(p - b) * (p - c));
    disp(['the area of your triangle is ', num2str(s)]);
end
%-----------------------------------------------------
```

5.1 WORKSHOP #1 – COMPARISONS IN MATLAB

This workshop will cover the topic of conditionals. Conditionals are considered as an algorithmic structure that allows branching –flow change– in the execution of a program from the evaluation of a logical expression –comparison–. In the theoretical development, regardless of the programming language, a foundation in Boolean algebra is required; expressions depend on the syntax of the programming language. From the conditionals, all the previous concepts are used. Therefore, it requires a greater motivational effort on the part of the teacher; For their part, students in their role as programmers assume situations where a decision must be made through Boolean logic. Thus, students require prior knowledge in propositional logic.

Proposed Exercises:

• We need a script for asking for the number of years according to the age of a person. If such a number is out of the valid range (between 0 and 120), an error message should be displayed; otherwise, calculate and show the number of months lived according to the number of years given by the user.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all %Clear screen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case *"Number of years according to the age of a person."*

```
disp("Number of years according to the age of a person")
```

Step 3: with the **disp** command the message *"How old are you?"* is displayed. Then the variable **x** is created with **input** to store the value entered by the user.

```
disp("How old are you?")
disp(" ")
x = input ("Enter your age: ");
```

Step 4: Once the user enters the age, the **if** conditional validates if the age is between zero and 120, if the number is out of this range the message "*The entered age is not validated*" is displayed.

```
if (x<0)||(x>120)
    disp("The entered age is not valid, re- run the program")
```

Step 5: if the entered age is in the range between 0 and 120, **else** will allow the program to continue executing and calculate the months and the message "*the months lived are:* ". The command **num2srt(m)** displays the result numerically.

```
else
    m = x*12;
    disp(" ")
    d = ['the months lived are: ', num2str(m)]
    disp(d)
end
```

Step 6: Finally, some tests are performed in **Command Window** to check the correct functioning of the program.

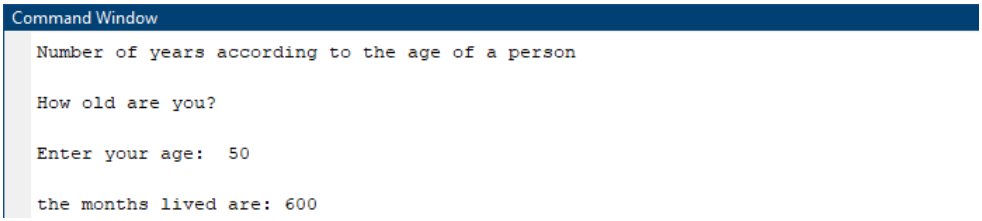


Figure 5-1. Command Window – Exercise 1 Comparisons in MATLAB

• Constructing a script for asking for a number between 1 and 12. Show the name of the month according to the given number by the user; if such a number is not in the valid range, show an error message.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case *"Number between 1 and 12. Show the name of the month according to the given number by the use."*

```
disp("Number betwewn 1 and 12. Show the name of the month
according to the given number by the user")
```

Step 3: A variable **x** is created with the **input** command that will allow the user to enter the desired value.

```
x = input ("Enter a number: ");
```

Step 4: with the conditional **switch** each of the options entered is validated, if any option is fulfilled, the program displays the corresponding month, **otherwise** it shows the message *" the entered value is incorrect re-run the program"* saying that no option was fulfilled.

```
switch x
    case 1
        disp('The mont is: January')
    case 2
        disp('The mont is: February')
    case 3
        disp('The mont is: March')
    case 4
        disp('The mont is: Abril')
    case 5
        disp('The mont is: May')
    case 6
        disp('The mont is: June')
    case 7
        disp('The mont is: July')

    case 8
        disp('The mont is: August')

    case 9
        disp('The mont is: September')
    case 10
        disp('The mont is: Octuber')
    case 11
        disp('The mont is: November')
    case 12
        disp('The mont is: December')
    otherwise
end
```

Step 5: Finally, some tests are performed in **Command Window** to check the correct functioning of the program.

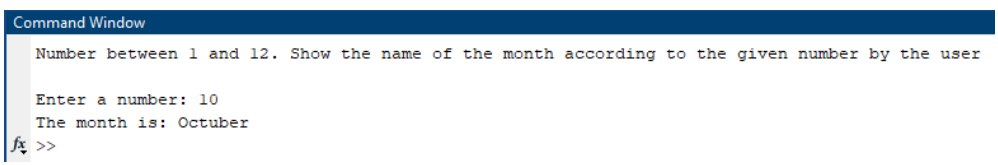


Figure 5-2. Command Window – Exercise 2 Comparisons in MATLAB

• We need a script for calculating both solutions of a quadratic equation according to the values of the 3 coefficients provided by the user.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case *"Number between 1 and 12. "Quadratic aquation solution"*.

```
disp("Quadratic equation solution")
```

Step 3: variables **A**, **B** and **C** are created with the **input** command for the user to enter the value of the coefficients to solve the quadratic equation.

```
A = input("enter the coefficient A: ");

B = input("enter the coefficient B: ");

C = input("enter the coefficient C: ");
```

Step 4: Once the coefficients are entered, the solutions of the quadratic equation are found considering the following equation:

$$x_{1,2} = \left(\frac{-B \pm \sqrt{B^2 - 4AC}}{2A}\right)$$

The equation is entered into the program and the results obtained in **x1** and **x2** are stored and printed.

```
disp("solutions of the quadratic equation are: ")

disp(" ")
```

```
x1 = (-B+sqrt(B^2 - 4*A*C))/(2*A)

x1 = (-B-sqrt(B^2 - 4*A*C))/(2*A)
```

Step 5: Finally, some tests are performed in **Command Window** to check the correct functioning of the program. Also, the variables used can be consulted in the **Workspace.**

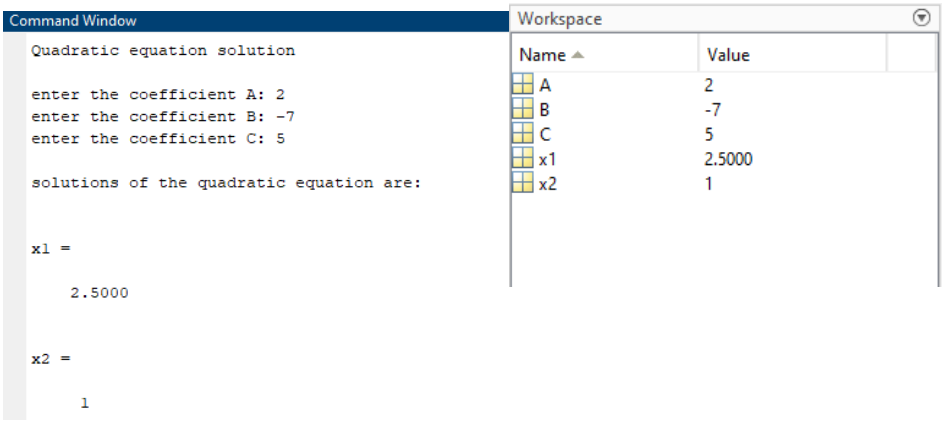


Figure 5-3. Command Window and Workspace

Exercise 3 Comparisons in MATLAB

NOTE: What if A = 0? What if the equation has complex solutions? How can we improve the script in this regard?

• Showing the greatest value from 3 given numbers by the user using comparisons

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case *"Compare three numbers"*.

```
disp("Compare three numbers")
```

Step 3: with the **disp** command the message "*Enter three numbers*" is printed for the user to enter the numbers. variables **A**, **B** and **C** are created with **input** to store the values entered by the user via keyboard.

```
disp("Enter the three numbers")
disp(" ")
A = input("enter the number 1: ");
B = input("enter the number 2: ");
C = input("enter the number 3: ");
```

Step 4: With the conditional **if** the values entered by the user are compared, first if determines if A is greater than B and C, if the condition is not fulfilled compares if B is greater than A and C, and finally compares if C is greater than A and B. when any of the conditional is fulfilled, the program prints the message "*the greatest number is*" and with the **num2str** command the numerical value of the variable is observed.

```
if (A>B)&&(A>C)
    da = ['the greatest number is: ' , num2str(A)];
    disp(da)
elseif (B>A)&&(B>C)
    db = ['the greatest number is: ' , num2str(B)];
    disp(db)
elseif (C>A)&&(C>B)
    dc = ['the greatest number is: ' , num2str(C)];
    disp(dc)
end
```

Step 5: Finally, some tests are performed in **Command Window** to check the correct functioning of the program. Also, the variables used can be consulted in the **Workspace.**

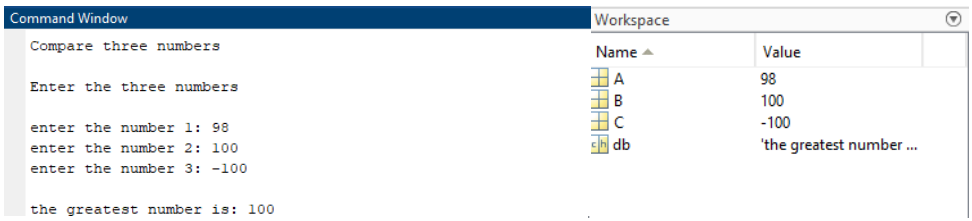


Figure 5-4. Command Window and Workspace

Exercise 4 Comparisons in MATLAB

• Allowing the user for typing a number between 1 and 10. If such a number is out of the valid range, show an error message; otherwise, show the number in roman notation.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case *"Number between 1 and 12. "Number in Roman notation"*.

```
disp("Number in roman notation")
```

Step 3: Variable **x** is created with **input** so that the user can enter a value.

```
x = input("enter the number: ");
```

Step 4: with the conditional **if**, the first thing that is validated is if the number is in the range of 1 to 10, if it is not between those values, the program prints the message "*the entered value is incorrect re-run the program*". On the contrary if the value is in that range, the program with the **elseif** goes on to validate that value meets the condition and prints the corresponding Roman numeral.

```
if (x<=0)||(x>10)
    disp("the entered value is incorrect re-run the program")
elseif (x==1)
    disp('the number is in roman notation: I')
elseif (x==2)
    disp('the number is in roman notation: II')
elseif (x==3)
    disp('the number is in roman notation: III')
elseif (x==4)
    disp('the number is in roman notation: IV')
elseif (x==5)
    disp('the number is in roman notation: V')
elseif (x==6)
    disp('the number is in roman notation: VI')
elseif (x==7)
    disp('the number is in roman notation: VII')
elseif (x==8)
    disp('the number is in roman notation: VIII')
elseif (x==9)
    disp('the number is in roman notation: IX')
elseif (x==10)
    disp('the number is in roman notation: X')
end
```

Step 5: Finally, some tests are performed in **Command Window** to check the correct functioning of the program.

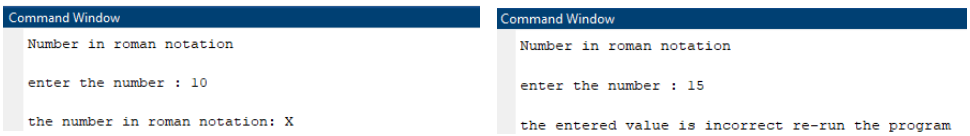


Figure 5-5 Command Window –Exercise 5 Comparisons in MATLAB

• Allowing the user for typing a letter of the English alphabet. Show its equivalent in morse notation.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case *"Number between 1 and 12. "Morse notation"*.

```
disp("Morse notation")
```

Step 3: A variable **x** is created with **input** so that the user can enter the letter. ***'s'*** is added so that the user does not add quotation marks when entering the letter by keyboard, because as it is a character, MATLAB must perform this process so that the value entered is stored in the variable correctly.

```
x = input("enter the letter : ", 's');
```

Step 4: When the user enters the letter, the program validates the entered information by means of the conditional **if** and **elseif**, when the entered letter meets the condition, the corresponding Morse notation is printed.

```
if (x=='A')
    disp("Morse notation is: ·-")
elseif (x=='B')
    disp("Morse notation is: -···")
elseif (x=='C')
    disp("Morse notation is: -·-·")
elseif (x=='D')
    disp("Morse notation is: -··")

elseif (x=='E')
    disp("Morse notation is: ·")
elseif (x=='F')
    disp("Morse notation is: ··-·")
elseif (x=='G')
    disp("Morse notation is: --·")
```

```
elseif (x=='H')
    disp("Morse notation is: ····")
elseif (x=='I')
    disp("Morse notation is: ··")
elseif (x=='J')
    disp("Morse notation is: ·---")
elseif (x=='K')
    disp("Morse notation is: -·-")
elseif (x=='L')
    disp("Morse notation is: ·-··")
elseif (x=='M')
    disp("Morse notation is: --")
elseif (x=='N')
    disp("Morse notation is: -·")
elseif (x=='O')
    disp("Morse notation is: ---")
elseif (x=='P')
    disp("Morse notation is: ·--·")
elseif (x=='Q')
    disp("Morse notation is: --·-")
elseif (x=='R')
    disp("Morse notation is: ·-·")
elseif (x=='S')
    disp("Morse notation is: ···")
elseif (x=='T')
    disp("Morse notation is: -")
elseif (x=='U')
    disp("Morse notation is: ··-")
elseif (x=='V')
    disp("Morse notation is: ···-")
elseif (x=='W')
    disp("Morse notation is: ·--")
elseif (x=='X')
    disp("Morse notation is: -··-")
elseif (x=='Y')
    disp("Morse notation is: -·--")
```

```
elseif (x=='Z')
    disp("Morse notation is: --··")
end
```

Step 5: Finally, some tests are performed in **Command Window** to check the correct functioning of the program.

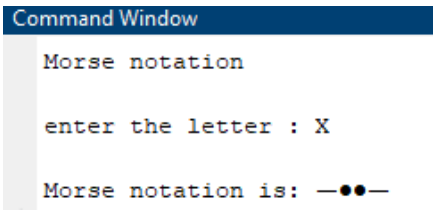


Figure 5-6. Command Window – Exercise 6 Comparisons in MATLAB

A loop in computer programming is a programming structure that allows a set of instructions to be executed repeatedly until a certain condition is met. A loop enables a program to perform a repetitive task, such as iterating over a list of items or repeatedly performing a calculation, without having to manually code each iteration (*Hosseini, Ouaknine & Worrell, 2019*).

There are different types of loops in programming, including:

- **for loop:** This loop is used to iterate over a sequence of values, such as a list or range of numbers, for a specific number of times.
- **while loop:** This loop repeatedly executes a set of instructions while a certain condition is true. The loop will continue until the condition becomes false.
- **do-while loop:** This loop is like a while loop, but it executes the instructions at least once before checking the condition.

Loops are a fundamental programming concept, and they are used in many different applications to automate repetitive tasks, perform calculations, and process data.

MATLAB provides several types of loops, including for loops, while loops, and do-while loops. Here's an overview of each of these types of loops and how to use them in MATLAB.

For Loops: A for loop is used when you want to execute a set of statements a specific number of times. The syntax of a for loop in MATLAB is:

```
for index = values
    % staments
end
```

The index variable is the loop counter, and the values variable is a vector or matrix that contains the values that the loop counter will take. For example, to print the numbers from 1 to 5, you could use the following code:

```
for i = 1:5
    disp(i)
end
```

While Loops: A while loop is used when you want to execute a set of statements if a certain condition is true. The syntax of a while loop in MATLAB is:

```
while condition
    % staments
end
```

The condition variable is the condition that the loop will check before executing each iteration of the loop. For example, to print the numbers from 1 to 5 using a while loop, you could use the following code:

```
i = 1;
while i<=5
    disp(i)
    i = i + 1;
end
```

Do-While Loops: MATLAB doesn't have a built-in do-while loop, but you can achieve the same effect using a while loop that executes at least once. For example, to print the numbers from 1 to 5 using a do-while loop, you could use the following code:

```
i = 1;
do
    disp(i)
    i = 1 + 1;
while i<= 5
```

In this code, the do block executes at least once before the while condition is checked.

The following MATLAB script calculates the factorial of a positive integer entered by the user.

Here is a line-by-line explanation of what the code is doing:

- Line 5: This line initializes a variable i to 1. i will be used as a counter in the while loop that calculates the factorial.
- Line 6: This line initializes a variable f to 1. f will be used to accumulate the factorial as the loop runs.
- Line 7: This line initializes a variable n to 0. n is not actually used for anything in this line, but it's good practice to initialize all variables.
- Line 9: This line displays a message on the console, indicating that the script is about to calculate the factorial of a number.
- Line 10: This line displays a blank line, for formatting purposes.
- Line 11: This line prompts the user to input a positive number, which is stored in the variable n.
- Line 12: This line checks if n is less than 0. If it is, a message is displayed indicating that only positive values should be typed. If n is greater than or equal to 0, the loop that calculates the factorial is executed.
- Line 15: This line starts a while loop that runs as long as i is less than or equal to n.
- Line 16: This line updates f to be equal to its current value times i. The effect of this is to calculate the factorial of n.
- Line 17: This line increases i by 1, so that the loop will eventually terminate.
- Line 19: This line displays the final result, which is the factorial of n. The num2str function is used to convert the number values to strings so they can be concatenated with the rest of the message in the disp function.

Overall, this script is a simple but effective example of how to use MATLAB to calculate a mathematical function.

```
%-------------------------------------------------------
% THE FACTORIAL OF A NUMBER
% By Jairo Guerrero, University of Nariño
%-------------------------------------------------------
i = 1;
f = 1;
n = 0;
%-------------------------------------------------------
disp('>>> THE FACTORIAL OF A NUMBER <<<')
disp('')
n = input('Type a positive number: ');
if n < 0
    disp('Oops! type positive values only.');
else
    while i <= n
        f = f * i;
        i = i + 1;
    end
    disp(['The factorial of ', num2str(n), ' is ', num2str(f)]);
end
%-------------------------------------------------------
```

Figure 6-1. MATLAB script calculates the factorial of a positive integer entered by the user

The following MATLAB script calculates the divisors of a given number entered by the user.

Here is a line-by-line explanation of what the code is doing:

- Line 5: This line initializes a variable n to 0. n will store the number entered by the user later in the code.
- Line 6: This line initializes a variable i to 1. i is a counter variable that will be used in the while loop later to test for divisors.
- Line 8: This line displays a message on the console, indicating that the script is about to calculate the divisors of a number.
- Line 9: This line displays a blank line, for formatting purposes.
- Line 10: This line prompts the user to input a number, which is stored in the variable n.

- Line 11: This line starts a while loop that runs as long as i is less than or equal to n.
- Line 12: This line checks if i is a divisor of n, using the mod function. If the remainder of the division of n by i is 0, then i is a divisor of n, so the code inside the if statement is executed.
- Line 13: This line displays the value of i along with a message indicating that it is a divisor of n. The num2str function is used to convert the number values to strings so they can be concatenated with the rest of the message in the disp function.
- Line 15: This line increases i by 1, so that the loop will eventually terminate.
- Line 16: This is the end of the while loop.

Overall, this script is a simple but effective example of how to use MATLAB to calculate the divisors of a number. It demonstrates the use of the mod function to check for divisibility and the while loop for repeating a task a certain number of times.

```
1  %----------------------------------------
2  % THE DIVISORS OF A NUMBER
3  % By Jairo Guerrero, University of Nariño
4  %----------------------------------------
5  n = 0;
6  i = 1;
7  %----------------------------------------
8  disp('>>> THE DIVISORS OF A NUMBER <<<')
9  disp('')
10 n = input('Type a number: ');
11 while i <= n
12     if mod(n, i) == 0
13         disp([num2str(i), ' is divisor of ', num2str(n)]);
14     end
15     i = i + 1;
16 end
17 %----------------------------------------
```

Figure *6-2*. MATLAB script calculates the divisors of a given number entered by the user

The following MATLAB script generates a random integer between 1 and 100, and then prompts the user to guess the number. The program provides feedback on each guess, indicating whether the user should guess a higher or lower number, until the user correctly guesses the "magic" number.

Here is a line-by-line explanation of what the code is doing:

- Line 5: This line generates a random integer between 1 and 100 using the randi function, and stores it in the variable magic.
- Line 6: This line initializes the variable n to 0. n will store the user's guess later in the code.
- Line 7: This line initializes the variable tries to 0. tries will store the number of guesses made by the user.
- Line 9: This line displays a message on the console, indicating that the script is about to prompt the user to guess a magic number.
- Line 10: This line starts a while loop that will continue until the user correctly guesses the magic number.
- Line 11: This line prompts the user to input a number between 1 and 100, which is stored in the variable n.
- Line 12: This line increasess the variable tries by 1, indicating that the user has made another guess.
- Line 13-19: These lines use if-elseif-else statements to provide feedback to the user based on their guess. If the guess is too low, the code in line 14 is executed. If the guess is too high, the code in line 16 is executed. If the guess is correct, the code in line 18 is executed.
- Line 20: This is the end of the while loop.

Overall, this script is a fun example of how to use MATLAB to create a simple guessing game. It demonstrates the use of conditional statements (if-elseif-else) for providing feedback to the user and the use of a while loop for repeating a task until a certain condition is met.

```
%----------------------------------------
% THE MAGIC NUMBER
% by Jairo Guerrero, University of Nariño
%----------------------------------------
magic = randi([1,100]);
n = 0;
tries = 0;
%----------------------------------------
disp('>>> THE MAGIC NUMBER <<<')
while n ~= magic
    n = input('Type a number between 1 and 100: ');
    tries = tries + 1;
    if magic > n
        disp('The magic number is greater');
    elseif magic < n
        disp('The magic number is lower');
    else
        disp(['CONGRATS! You did it in ', num2str(tries), ' tries']);
    end
end
%----------------------------------------
```

Figure 6-3. MATLAB script generates a random integer between 1 and 100

The following MATLAB script is a program for listing prime numbers. Here's a breakdown of the code:

- pn, i, n, j, and dc are all variables that are initialized to zero. pn represents the number of prime numbers to be listed, i is a counter for the while loop, n represents the current number being evaluated, j is a counter for the for loop, and dc is a counter for the number of divisors of n.
- The fprintf function is used to print the string "LISTING PRIME NUMBERS" to the command window.
- The input function is used to prompt the user for the number of prime numbers they want to list. The input is stored in the pn variable.
- The while loop runs until i is less than pn.
- Inside the while loop, dc is set to 0. This variable is used to count the number of divisors of the current number n.
- A for loop is used to iterate through all the numbers from 1 to n. Inside the for loop, the rem function is used to check if n is divisible by j. If it is, dc is increased by 1.

- After the for loop, if dc is equal to 2, then the current number n is a prime number, and it is printed to the command window using the fprintf function. i is also increased by 1 to keep track of the number of prime numbers listed.
- Finally, n is increased by 1 and the loop continues.

This program uses a brute force method to check whether each number is prime, by checking whether it has only two divisors. There are more efficient algorithms for determining whether a number is prime, but this method is simple and easy to understand.

```
%-------------------------------------------------------
% >>> LISTING PRIME NUMBERS <<<
% by Jairo Guerrero, University of Nariño
%-------------------------------------------------------
pn = 0;
i = 0;
n = 2;
j = 0;
dc = 0;
%-------------------------------------------------------
fprintf("LISTING PRIME NUMBERS\n");
pn = input("How many prime numbers do you want? ");
while i < pn
    dc = 0;
    for j = 1:n
        if rem(n,j) == 0
            dc = dc + 1;
        end
    end

    if dc == 2
        fprintf("%d\n",n);
        i = i + 1;
    end
    n = n + 1;
end
%-------------------------------------------------------
```

6.1 WORKSHOP #2 – LOOPS IN MATLAB

In this workshop, the topic about loops is covered. Loops are algorithmic structures that allow the iteration −repetition− of one or more instructions of the program from a starting point −start−, the evaluation of a logical expression −comparison of completion− and the progress of this through increments or control variable decrements. Within the theoretical development, regardless of the programming language, a foundation in Boolean algebra is required; expressions depend on the syntax of the programming language. At this point, abstraction takes a leading role when designing cyclic structures.

Proposed Exercises:

```
clc; clear; close all; %Clear sreen and delete variables

disp ("sum of even and odd numbers")

vector = 1:1:200;
even = 1;
odd = 1;
for i = 1 :length(vector)

    if (mod(vector(i),2) == 0);
        V_even(even) = (vector(i));
        sum_even = sum(V_even);
        even = even + 1;
    else
        V_odd(odd) = (vector(i));
        sum_odd = sum(V_odd);
        odd = odd +1;
    end

end

disp(" ")
disp("sum of even numbers")
sum_even
disp(" ")
disp("sum of odd numbers")
sum odd
```

• We want to calculate independently the sum of the numbers even and odd between 1 and 200.

Step 1: we open a new script and execute the clc, clear and close commands to clear the screen and delete variables.

Step 2: with the disp command we print the title of our project, in this case "sum of even and odd numbers."

Step 3: the variable "vector" is defined that corresponds to the vector of values from 1 to 200. In addition, two variables are added (even and odd) that will be the counters for the for loop.

Step4: a for loop is created to go through each of the positions of the vector and with a conditional if it is determined if the number is odd or even.

Conditional: for the conditional we used the mod that returns what is left over in a division, to determine if the number is even, we use the mod 2 and if we obtain a residue equal to zero the number is stored in the vector V_even that corresponds to the even numbers, if it does not meet the condition the number is stored in the vector V_odd that corresponds to the odd numbers.

Step 5: Once the even and odd numbers have been obtained and their corresponding vector has been stored, the sum is performed with the sum command, which allows the sum of the elements in a vector.

Step 6: When the program is executed the results are printed on the screen, the disp command is used to display them on the right side of the screen.

Step 7: Some of the results obtained in the exercise can also be consulted, such as the vectors of the odd and even numbers (V_even, V_odd) in the lower left part of the screen.

• Read a series of non-zero numbers (the last number of the series is –99) and get the greatest number. As a result, the number should be displayed. greater and a negative number indication message, in case a negative number has been read.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case "*Read a series of non-zero numbers.*"

```
disp("Read a series of non-zero numbers")
```

Step 3: A variable **n** is defined with **input** that allows the user to determine how many numbers he will need for the series; v is also defined as an empty vector that will serve to store the series of numbers. And a message with the **disp** command for the user to type the numbers of the series.

```
n = input("Enter how many numbers you require for the series: ")
v = [];
disp("enter the numbers")
```

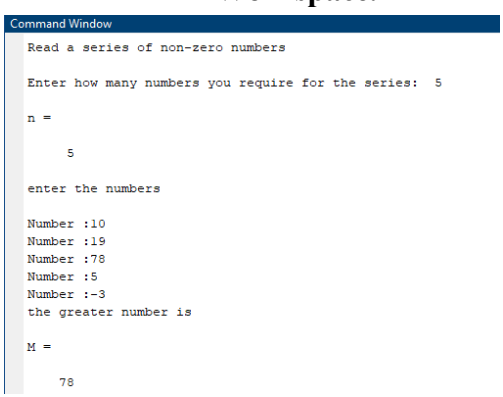
Step 4: the **for loop** is created to store the data of the series entered by the user, the conditional **if** allows to verify the information required in the problem which is: non-zero numbers and the last number of the series is -99, if any of the conditions are not met the program will send a message and must be executed again, on the other hand if the conditions are met, the values of the series will be stored in the vector **v**

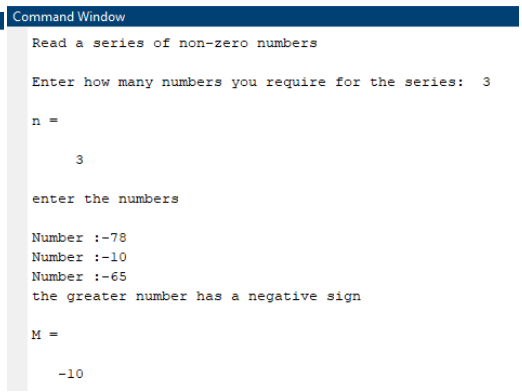
```
for i = 1:n
    x = input("Number: ");

    if ( x == 0 )||(x < -99)
        disp("A number out of the series was entered, re-run
the program")
        break
    else
        v = [v,x];
    end

end
```

Step 5: the variable **M** is defined, and the **max** command is used to determine the maximum value in the vector, the conditional **if** will validate if the greatest number has negative sign, if it meets the condition, it will show the following message "*the greater number has a negative sign* " otherwise it will print the following message" *the greater number is*" and will display the number.

```
M = max(v);
if M<0
    disp("the greater number has a negative sign")
    M

else
    disp("the greater number")
    M
end
```

Step 6: Finally, some tests are performed in **Command Window** to check the correct functioning of the program. Also, the variables used can be consulted in the **Workspace.**





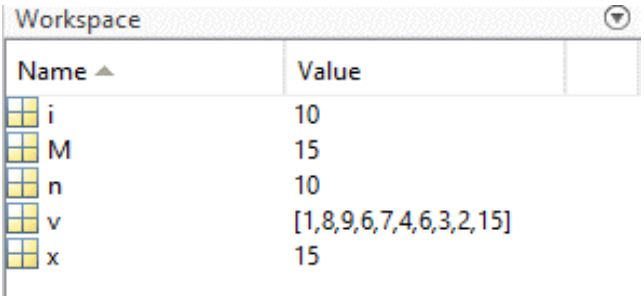


Figure 6-4. Command window and Workspace. LOOPS IN MATLAB: Read a series of non-zeros numbers

• Calculate and display the sum and product of the even numbers between 20 and 400, both inclusive.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case "*Calculate and display the sum and product of the even numbers between 20 and 400, both inclusive.*"

```
disp("Calculate and display the sum and product of the even
numbers")
disp("between 20 and 400, both inclusive")
```

Step 3: the **vector** variable is defined with the corresponding even numbers from 20 to 400 and a counter with the name **sum_even** that will be used later to sum the numbers in the for loop.

```
vector = 20:2:400;
sum_even = 0;
```

Step 4: the **for** loop will allow to go through the positions of the vector containing the even numbers and perform the sum of each of these values that will be stored in the variable **sum_even**. By means of the **prod** command, the product of the corresponding even numbers stored in the **vector** variable is performed.

```
for i=1:length(vector)
    sum_even = sum_even + vector(i);
end

prod_even = prod(vector)
```

Step 5: By means of the **disp** command, a message and its corresponding result are printed on the screen.

```
disp(" ")
disp("Sum of even numbers between 20 and 400 is: ")
sum_even
```

```
disp(" ")
disp("Product of even numbers between 20 and 400 is: ")
prod_even
```

Step 6: finally, the program is executed, and its correct operation is verified in the **Command Window** screen, Also, the variables used can be consulted in the **Workspace**. The product, having a very large result, prints **inf** (infinity).

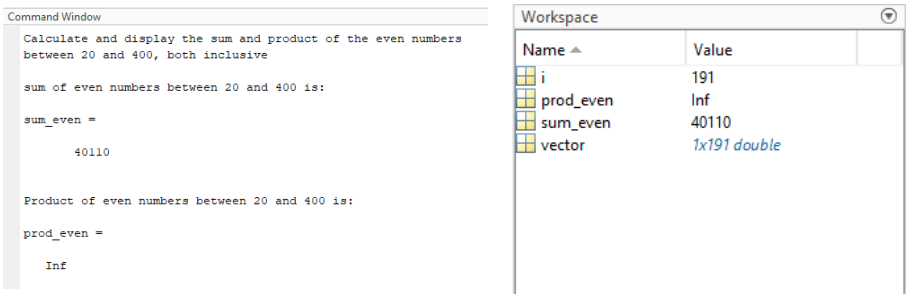


Figure 6-5. Command window and Workspace. LOOPS IN MATLAB: Calculate and display the sum and product of the even numbers between 20 and 400

• Calculate the sum of the squares of the first hundred natural numbers.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case "*Calculate the sum of the squares.*"

```
disp("Calculate the sum of the squares")
```

Step 3: The **vector** variable is defined to obtain the first hundred numbers to later find the square. A **sum_squares** counter is also defined to perform the operation corresponding to the sum of the squares.

```
vector = 1:1:100;
sum_squares = 0;
```

Step 4: the **for loop** is used to go through the numbers from one to one hundred and by means of the **sum_squares** counter, the result obtained by squaring each number and adding it to the previous one is accumulated.

```
for i =1:length(vector)
    sum_squares = sum_squares + (i^2);
end
```

Step 5: The result obtained with the **disp** command is printed on the screen.

```
disp(" ")
disp("The sum of the squares of the first hundred natural
numbers is: ")
sum_squares
```

Step 6: The program is executed, and the correct operation is verified in the **Command Window**, and the variables used in the **Workspace** can also be observed.

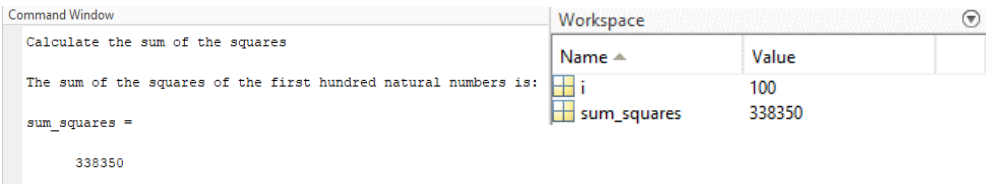


Figure 6-6. Command window and Workspace. LOOPS IN MATLAB: Calculate the sum of the squares

• Add ten numbers entered by keyboard.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with disp command we print the title of project, in this case "*Add ten numbers entered by keyboard*".

```
disp("Add ten numbers entered by keyboard")
```

Step 3: A variable **L** is defined to determine the ten numbers that the user will enter and store by keyboard, **v** is defined as an empty vector that will serve to store the data entered by the user, a message is also displayed on the screen for the user to enter the numbers using the **disp** command.

```
L = 10;
v = [];
disp("enter the numbers")
disp(" ")
```

Step 4: the **for loop** allows the user to enter the ten numbers, by means of the variable **x** with the **input** command the user will enter the desired numbers, and in the vector **v** the values will be stored. The loop will end when the user has typed the ten numbers.

```
for i = 1:L
    x = input("Number: ");
    v = [v,x];
end
```

Step 5: The numbers entered by the user are displayed on the screen.

```
disp("The numbers entered are")
v
```

Step 6: The program is executed, and the correct operation is verified in the **Command Window**, and the variables used in the **Workspace** can also be observed. The ten numbers entered by the user are displayed in the command window.

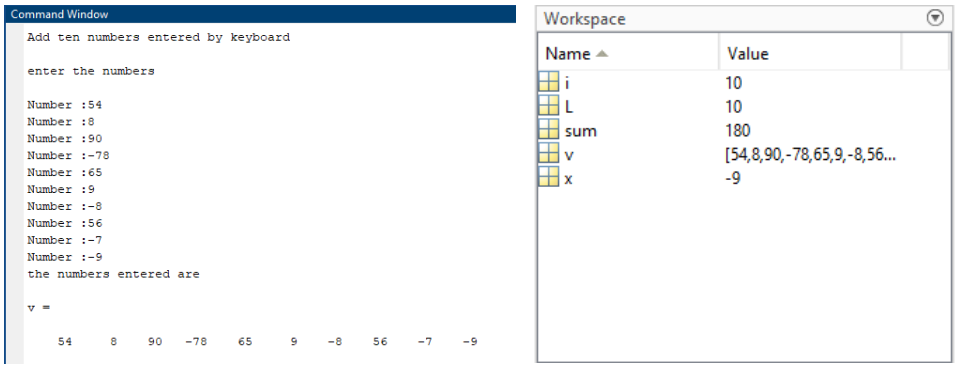


Figure 6-7. Command Window and Workspace. LOOPS IN MATLAB: Add ten numbers entered by keyboard

A function is a block of code in computer programming that performs a specific task. Functions are designed to be reusable, meaning that they can be called from different parts of a program and with different inputs (*Kochenderfer & Wheeler, 2019*).

Functions can be defined in most programming languages, including MATLAB, Python, Java, and many others. Here are some common characteristics of functions in programming:

- A function has a name, which is used to identify it when it is called from other parts of the program.
- A function can take inputs, also known as parameters, which are values that the function uses to perform its task. These inputs can be of different types, depending on the language and the needs of the function.
- A function can return a value or set of values as its output. This output can be of different types, depending on the language and the needs of the function.
- A function can contain any number of statements, which are executed when the function is called.
- A function can be defined anywhere in a program, but it is usually defined before it is called.

In computer programming, there are several types of functions, each with its own specific purpose. Here are some common types of functions:

- Built-in functions: These are functions that are built into the programming language or provided by the programming environment. Examples of built-in functions include print and input in Python, plot and rand in MATLAB, and System.out.println in Java.
- User-defined functions: These are functions that are created by the user in the program to perform specific tasks. In many programming languages, you can define your own functions using the function keyword. These functions can be designed to take input, perform calculations, and return output.
- Recursive functions: These are functions that call themselves repeatedly until a condition is met. Recursive functions are useful for solving problems that can be broken down into smaller sub-problems. Examples of recursive functions include the Fibonacci sequence and calculating the factorial of a number.
- Lambda functions (also known as anonymous functions): These are functions that are defined without a name and can be used as arguments to other functions. Lambda functions are useful for performing simple operations and can help simplify code. Examples of functions that can take lambda functions as arguments include map and filter in Python and arrayfun in MATLAB.
- Higher-order functions: These are functions that take one or more functions as arguments or return a function as output. Higher-order functions are useful for creating more modular and flexible code. Examples of higher-order functions include sort and map in Python and arrayfun and feval in MATLAB.

These are just a few examples of the types of functions that exist in computer programming. Different programming languages may have their own specific types of functions or use different terminology, but the underlying concepts are similar.

In MATLAB, we can follow the steps below in order to create and use functions.

1. Open a new script in MATLAB and type the function keyword followed by the name of the function you want to create. The name of the function should be descriptive and meaningful, as it will be used to call the function from other parts of your code.
2. After the function name, list the input arguments inside parentheses. Input arguments are the values that your function will take as input when it is called. You can have zero or more input arguments.
3. Next, use the = operator to define the output arguments of the function, if any. Output arguments are the values that your function will return as output when it is called. You can have zero or more output arguments.
4. Write the code that you want your function to execute. This can include any valid MATLAB code, such as loops, conditionals, and calculations.
5. When your function is done, use the end keyword to indicate the end of the function.

Here's an example of a simple function that takes two input arguments and returns their sum:

```
funtion result = add(a,b)
%this function adds wo numbers and returns the result
result = a + b;
end
```

To call this function from other parts of your code, you simply use its name and pass in the necessary input arguments, like this:

```
x = add(3, 4);
```

In this example, the function add is called with the arguments 3 and 4. The result of the function is assigned to the variable x, which will have the value 7.

When calling a function, you can also use the [] operator to capture the output arguments of the function, like this:

```
[a, b]= myFunction(input1, input2);
```

In this example, myFunction is called with two input arguments, input1 and input2. The two output arguments of the function are assigned to the variables a and b.

In addition to creating your own functions, MATLAB has many built-in functions that you can use in your code. You can find a list of these functions in the MATLAB documentation.

The flowing MATLAB script defines a function called GoldenRatio that takes one input argument fn, which represents the number of Fibonacci numbers to use for the calculation.

Inside the function, the script initializes variables i, a, b, and c to 0. It then uses a for loop to calculate the fnth number of the Fibonacci sequence by adding the previous two numbers together.

Finally, the script divides the last two numbers of the sequence to obtain the golden ratio, and stores it in a variable called golden_ratio.

After defining the GoldenRatio function, the script prompts the user to enter the number of Fibonacci numbers to use by calling the input function with the string "Type the number of Fibonacci's to use: " as an argument. It then passes the user's input to the GoldenRatio function, and displays the resulting golden ratio using the disp function.

```
%-------------------------------------------------------------
% >>> CALCULATING THE GOLDEN RATIO BASED ON FIBONACCI'S <<<
% by Jairo Guerrero, University of Nariño
%-------------------------------------------------------------
function [golden_ratio] = GoldenRatio(fn)
    i = 0;
    a = 0;
    b = 1;
    c = 0;
    for i = 1:fn
        c = a + b;
        a = b;
        b = c;
    end
    c = b / a;
    golden_ratio = c;
end
%-------------------------------------------------------------
```

The following script is a MATLAB script that simulates rolling a six-sided die a specified number of times and counts the number of even and odd rolls.

The script begins by initializing two variables, even and odds, to 0.

The script then defines a function called DrawDice that takes one input argument dice, which represents the value rolled on the die. The function outputs an ASCII art representation of the die face with the corresponding number of dots displayed.

After defining the DrawDice function, the script prompts the user to enter the number of rolls they want by calling the input function with the string "How many rolls do you want (1..10)? " as an argument. It then uses a for loop to simulate rolling the die the specified number of times.

Inside the loop, the script generates a random integer between 1 and 6 (inclusive) using the randi function, and passes it to the DrawDice function to display the corresponding die face.

The script then uses the mod function to determine whether the roll is even or odd. If the roll is even, it increments the even counter by 1. Otherwise, it increments the odds counter by 1.

After the loop finishes, the script displays the number of rolls, the number of even rolls, and the number of odd rolls using the disp function and string concatenation.

```
%-------------------------------------------------------------
% >>> ROLLING A DICE <<<
% by Jairo Guerrero, University of Nariño
%-------------------------------------------------------------
even = 0;
odd = 0;
%-------------------------------------------------------------
function DrawDice(dice)
    disp("+-------+");
    if dice == 1
        disp("|       |");
        disp("|   *   |");
        disp("|       |");
    elseif dice == 2
        disp("| *     |");
        disp("|       |");
        disp("|     * |");
    elseif dice == 3
        disp("| *     |");
        disp("|   *   |");
        disp("|     * |");
    elseif dice == 4
        disp("| *   * |");
        disp("|       |");
        disp("| *   * |");
    elseif dice == 5
        disp("| *   * |");
        disp("|   *   |");
        disp("| *   * |");
```

```
    elseif dice == 6
        disp("| *   * |");
        disp("| *   * |");
        disp("| *   * |");
    end
    disp("+-------+");
end
%-------------------------------------------------------------
disp("");
disp(">>> ROLLING DICE <<<");
disp("");
rolls = input("How many rolls do you want (1...10)? ");
for i = 1:rolls
    dice = randi([1 6]);
    DrawDice(dice);
    if mod(dice, 2) == 0
        even = even + 1;
    else
        odds = odds + 1;
    end
end
disp("");
disp("Rolls: " + string(rolls) + ", even: " + string(even) +
", odd: " + string(odds));
%-------------------------------------------------------------
```

7.1 WORKSHOP #3 – FUNCTIONS INS MATLAB

In this workshop, the concept of function will be worked on as a part of the program that performs specific tasks and has the facility of being called in different instances within the program; they can manage input parameters and can return output values. The concept of function is the most complex of the objects selected in this investigation, it involves the integration of all the previous concepts. Therefore, several practical exercises are necessary to strengthen your learning. It is here that students in their role as professional programmers exploit their creative potential using functions.

Proposed Exercises:

• Building a script with a function for asking a given temperature and a two-character string (like these ‘cf’,’ck’,’kf’,’kc’,’fc’, and ‘fk’). Such a function must convert the temperature according to the two-character string; for instance: cf stands for Celsius to Fahrenheit, fk stands for Fahrenheit to Kelvin, and so on.

Step 1: A new script with function is created.

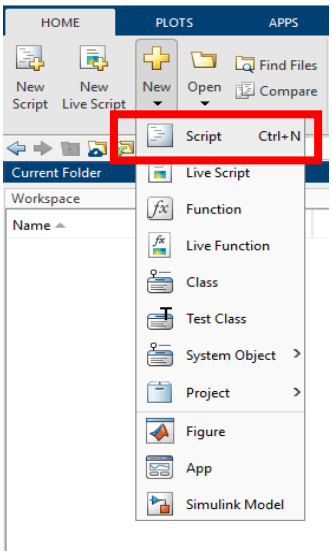


Figure 7-1. Icon for creating a new script

Step 2: click on **function** and we get the following:

```
function [outputArg1, outputArg2] = untitled(inputArg1,
inputArg2)
%    UNITITLED Summary of this function goes here
%    Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

It contains the output parameters, the input parameters, the title and the body for the function syntax.

Step 3: For this case the output parameter of the function is **Tout** (Output temperature), the input parameters are **Tin** and **C** (Input temperature and Conversion) and the title for the function is **ConversionT.**

```
function [Tout] = Conversion(Tin, C)
```

Step 4: The variables **Tin** and **C** with **input** are created so that the user can enter the input temperature value and the desired conversion type by keyboard.

```
Tin = input("Enter Temperature: ");
C = input("Enter Conversion: ",'s');
```

Step 5: With the **fprintf** command a menu is created for the user to select the type of conversion.

```
fprintf("Temperature conversion type: \n")
fprintf("cf -- Celcius - Fahrenheit\nck -- Celcius -
Kelvin\nkf -- Kelvin - Fahrenheit\n")
fprintf("kc -- Kelvin - Celcius\nfc - Fahrenheit - Celcius\nfk
-- Faherenheit - Kelvin\n")
disp(" ")
```

Step 6: A **switch case** sentence is created to execute one of several groups of instructions. For this program, the input variable C entered by the user is evaluated, which corresponds to the conversion type.

```
switch c
    case 'cf'
        Tout = (Tin*(9/5)+32);

        fprintf("Celcius: %d\n", Tin)
        fprintf("Fahrenheit: %", Tout)
```

```
    case 'ck'
        Tout = (Tin+273.15);

        fprintf("Celcius: %d\n", Tin)
        fprintf("Kelvin: %", Tout)

    case 'kf'
        Tout = ((Tin - 273.15)*(9/5) + 32);

        fprintf("Kelvin: %d\n", Tin)
        fprintf("Fahrenheit: %", Tout)

    case 'kc'
        Tout = (Tin - 273.15);

        fprintf("Kelvin: %d\n", Tin)
        fprintf("Celcius: %", Tout)

    case 'fc'
        Tout = (Tin-32)*(5/9);

        fprintf("Fahrenheit: %d\n", Tin)
        fprintf("Celcius: %", Tout)

    case 'fk'
        Tout = ((Tin-32)*(5/9) + 273.15);

        fprintf("Fahrenheit: %d\n", Tin)
        fprintf("Kelvin: %", Tout)

    otherwise
        fprintf("Incorrect entered  conversion")
```

if the variable entered by the user meets any of the cases of the statement, the program performs the required conversion and prints the values, if it does not meet any of the cases, the program displays the following message “incorrect entered conversion.”

Step 7: Finally in the command window the function is called by typing its name (ConversionT), it will ask to enter the input values (Tin and C) and will print the menu and display the result of the temperature conversion.

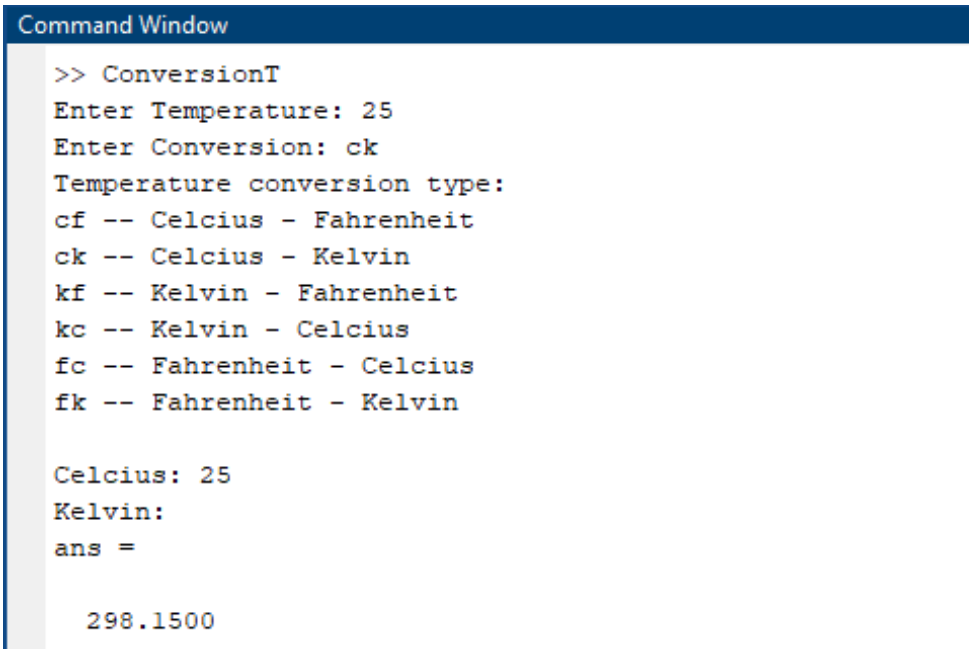


Figure 7-2. Command window. FUNCTIONS: Conversion Temperature

• Building a script with a function for converting cartesian coordinates into polar coordinates and vice versa.

Step 1: A new script with function is created.

Step 2: click on **function** and we get the following:

```
function[outputArg1, outputArg2] = untitled(inputArg1,
inputArg2)
%    UNITITLED Summary of this function goes here
%    Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

It contains the output parameters, the input parameters, the title and the body for the function syntax.

Step 3: For this case the output parameters of the function are **x, y, r, t** (Cartesian and polar coordinates), the input parameters are **x, y, r, t** (Cartesian and polar coordinates) and the title for the function is **CartesianPolar.**

```
function[x,y,r,t] = CartesianPolar(x,y,r,t)
```

Step 4: With **fprintf** a menu is created for the user to select the type of conversion desired, 1 (Cartesian - Polar), 2 (Polar - Cartesian), and an **S** variable is created with the **input** command to store the user's selection.

```
    fprintf("Select conversion\n 1 Cartesian - Polar\n 2 Polar
- Cartesian\n")
    S = input ("conversion: ");
```

Step 5: A **switch case** sentence is created to execute one of several groups of instructions. For this program, the input variable S entered by the user is evaluated, which corresponds to the conversion type.

```
switch S
    case 1
        fprintf("conversion Cartesian - Polar\n")
        x = input("enter the value of x: ");
        disp(" ")
        y = input("enter the value of y: ");
        disp(" ")

        fprintf("The polar coordinates are\n")
        fprintf("Distance: \n")
        r = sqrt((x)^2+(y)^2)
        fprintf("Angle: ")
        t = atan(y/x)
```

```
    case 2
        fprintf("conversion Polar - Cartesian \n")
        x = input("enter the value of r: ");
        disp(" ")
        y = input("enter the value of t: ");
        disp(" ")

        fprintf("The cartesian coordinates are\n")
        fprintf("x-coordinate: \n")
        x = r*cos(t)
        fprintf("Angle: ")
        y = r*sin(t)

    otherwise
        fprintf("incorrect entered conversion")
```

If the user enters the value **1**, the program will ask him to enter the input values to the function of **x,y** (Cartesian coordinates) and perform the corresponding transformations to obtain the output values of the function **r and t** (polar coordinates) and display these values on the screen, on the other hand if the option is **2** the input values to the function that the user must enter are **r and t** (polar coordinates) and the output values of the function will be **x,y** (Cartesian coordinates).

Step 6: Finally in Command Window the function is called with its name to execute the program, it will ask to enter the type of conversion and the parameters for the function to print the corresponding values. The angle to be entered and the one calculated by the program is in radians**.**

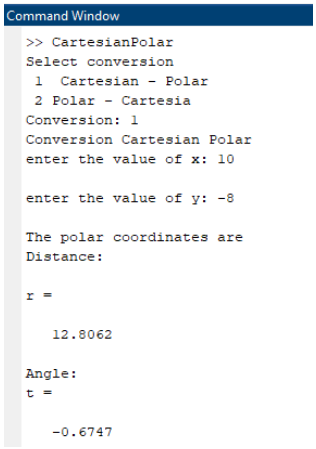


Figure 7-3. Command Window. FUNCTIONS: Conversion Coordinates Polar - Cartesian

• Building a script with a function for calculating the number of active, communicative extraterrestrial civilizations in the Milky Way Galaxy according to the Drake's equation.

Step 1: A new script with function is created.

Step 2: click on **function** and we get the following:

```
function[outputArg1, outputArg2] = untitled(inputArg1,
inputArg2)
%    UNITITLED Summary of this function goes here
%    Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

It contains the output parameters, the input parameters, the title and the body for the function syntax.

Step 3: For this case the output parameter of the function is **N** (Number of civilizations that could communicate in our galaxy, the Milky Way), the input parameters are **R, fp, ne, fl, fi, fc, L** (The other parameters that must be considered for the Drake's equation) and the title for the function is DrakeE**.**

```
function[N] = DrakeE(R,fp,ne,fl,fi,fc,L)
```

Step 4: A brief description of what each parameter means is given in the commentary.

```
 %R* Annual rate of  formation of "proper" stars in the galaxy
 %fp Fraction of stars that have planets in their orbit
 %ne Number of those planets orbiting within the habitable zone
of the star
 %fl Fraction of those planets within the habitable zone on
which life has developed
 %fi Fraction of those planets on which intelligent life has
developed
 %fc Fraction of those planets where intelligent life has
developed a technology and attempts to communicate
 %L Time span, measured in years, during which an intelligent
and communicative civilization can exist
```

Step 5: The corresponding values previously consulted are assigned to each input variable and Drake's equation is applied and print the result.
Drake´s equation is:

$$N = R * \boldsymbol{f_p} * \boldsymbol{n_e} * \boldsymbol{f_l} * \boldsymbol{f_i} * \boldsymbol{f_c} * \boldsymbol{L}$$

```
 R = 10; Fp =  1/2; ne = 2; fl = 1; fi = 0.01; fc = 0.01; L =
10000;

 fprintf("The number of civilization that could communicate in
the Milky way Galaxy are: ")
 N = R*fp*ne*fl*fi*fc*L;
 end
```

Step 6: Finally in Command Window the function is called with the name assigned to it and we can obtain the result of Drake's Equation.

```
Command Window
>> DrakeE

fprintf =

    "The number of civilizations that could communicate in the Milky Way Galaxy are: N "


ans =

    10
```

Figure 7-4. Command Window. FUNCTIONS: DrakeE

• Building a script with a function for determining if a number given by the user is prime number or not.

Step 1: A new script with function is created.

Step 2: click on **function** and we get the following:

```
function[outputArg1, outputArg2] = untitled(inputArg1,
inputArg2)
%    UNITITLED Summary of this function goes here
%    Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

It contains the output parameters, the input parameters, the title and the body for the function syntax.

Step 3: For this case the output parameter of the function is **P** (Variable that determines whether the number is prime or not), the input parameters is **N** (Number entered by user) and the title for the function is Prime**.**

```
function[P] = Prime(N)
```

Step 4: With the **fprintf** command we enter a title to be displayed on the screen. And with input we ask the user to enter the input value (**N**) to the function.

```
    fprintf("determine if a number is prime\n\n")
    N = input("Enter the number: ");
```

Step 5: By means of the **for** loop we validate if the number entered by the user is prime, this is done with the **mod** command that gives us the remainder of a division, the conditional **if** validates that if the remainder of that division is or, then the message that the number is not prime is printed, but if the condition is not met and the program exits the **for** loop it validates that the number is prime.

```
    for i = 2: N-1
        r = mod (N,i);
        if r==0
            fprintf("The number %i is not prime \n", N)
            return
        end
    end

    fprintf("The number %i is prime \n", N)

end
```

Step 6: Finally with Command Window the function is called with the assigned name and the program asks the user to enter a number to determine if it is prime or not.

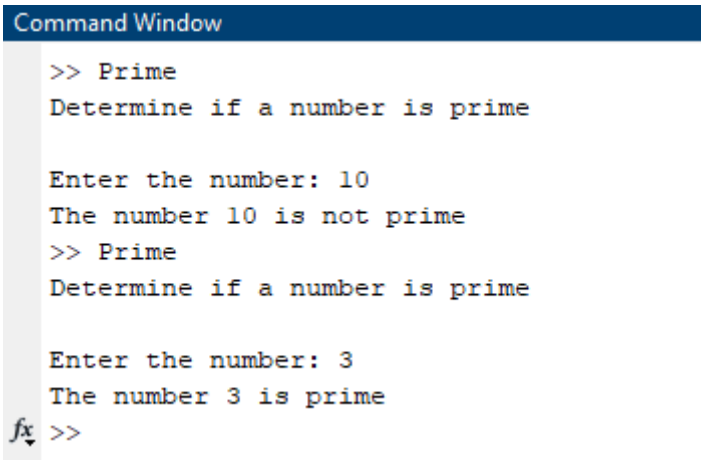


Figure 7-5. Command window. FUNCTIONS: Determinate if a number is prime

In computer programming, an array is a data structure that stores a collection of elements, all the same type, in contiguous memory locations. Each element in the array is identified by an index or a key that represents its position in the array.

Arrays are useful when you need to store a large amount of data of the same type and want to access it quickly and efficiently. By using an index or a key, you can easily access any element in the array, making it easy to search, sort, and manipulate the data stored in the array.

Arrays are commonly used in programming languages like C, C++, Java, Python, MATLAB, and others. There are different types of arrays such as one-dimensional arrays, multi-dimensional arrays, and jagged arrays, each with its own specific uses and properties.

There are several types of arrays in computer programming, including:

- One-dimensional arrays: Also known as a flat array or a vector, a one-dimensional array stores elements in a single row or sequence. Each element in the array is accessed using a single index or subscript, which represents its position in the sequence.
- Multi-dimensional arrays: Multi-dimensional arrays are arrays with more than one index or subscript. They are often used to represent matrices or tables. A two-dimensional array, for example, has rows and columns, and its elements are accessed using two indices.
- Jagged arrays: A jagged array is an array of arrays, where each element in the array is itself an array. Unlike multi-dimensional arrays, jagged arrays can have different lengths for each subarray.

- Dynamic arrays: A dynamic array is an array whose size can be dynamically adjusted during runtime. This allows you to add or remove elements from the array as needed. In some programming languages, dynamic arrays are implemented using data structures such as linked lists or resizable arrays.
- Associative arrays: Also known as a map, dictionary, or hash table, an associative array is an array where the elements are accessed using a key instead of an index. The key-value pairs are stored in the array, and the key is used to retrieve the corresponding value.
- Sparse arrays: A sparse array is an array that contains mostly empty or null values. To save memory, only non-empty values are stored, along with their corresponding indices.

The specific types of arrays available in a programming language may vary, and some languages may support additional types of arrays not listed here.

In MATLAB, you can declare and use arrays using the following syntax:

One-dimensional arrays: To declare a one-dimensional array, use square brackets [] to enclose a comma-separated list of elements. For example:

```
a = [1, 2, 3, 4, 5];
```

You can access elements in the array using their index:

```
disp(a(2)); % Output:2
```

Multi-dimensional arrays: To declare a multi-dimensional array, use square brackets [] and semicolons ; to separate rows. For example:

```
b = [1, 2, 3, 4, 5, 6, 7, 8, 9];
```

You can access elements in the array using their row and column indices:

```
disp(b(2, 3)); % Output:6
```

Jagged arrays: To declare a jagged array, use cell arrays, which are arrays that can hold elements of different data types. For example:

```
c = {1, [2, 3], [4, 5, 6, 7]};
```

You can access elements in the array using their index:

```
disp(c{2}(1)); % Output: 2
```

Dynamic arrays: In MATLAB, arrays are dynamic by default, so you can add or remove elements from an array using the following syntax:

```
a(6) = 6;
disp(a); % Output: [1, 2, 3, 4, 5, 6]
```

Associative arrays: To declare an associative array, use the containers.Map class, which allows you to map keys to values. For example:

```
d = containers.Map({'one', 'two', 'three'}, [1, 2, 3]);
```

You can access elements in the array using their keys:

```
disp(d('two')); % Output:2
```

These are just a few examples of how to declare and use arrays in MATLAB. There are many more features and functions available for working with arrays in MATLAB, including built-in functions for sorting, searching, and manipulating arrays.

MATLAB has several built-in functions for performing arithmetic calculations with vectors. Here are some common examples:

Vector Addition and Subtraction

You can add or subtract two vectors of the same size using the + and - operators, respectively.

```
v1 = [1 2 3];
v2 = [4 5 6];
v3 = v1 + v2; % Vector addition
v4 = v1 - v2; % Vector subtraction
```

Scalar Multiplication and Division

You can multiply or divide a vector by a scalar using the * and / operators, respectively.

```
v1 = [1 2 3];
s = 2;
v2 = s * v1; % Scalar multiplication
v3 = v1 / s; % Scalar division
```

Dot Product

You can calculate the dot product of two vectors using the dot() function.

```
v1 = [1 2 3];
v2 = [4 5 6];
dp = dot(v1, v2); % Dot product
```

Cross Product

You can calculate the cross product of two vectors using the cross() function.

```
v1 = [1 2 3];
v2 = [4 5 6];
cp = cross (v1, v2); % Cross product
```

Magnitude and Normalization

You can calculate the magnitude of a vector using the norm() function, and you can normalize a vector (i.e., make it a unit vector) using the normalize() function.

```
v1 = [1 2 3];
mag = norm(v1); % Magnitude of v1
v2 = normalize(v1); % Normalize v1
```

In order to work with matrices, here is a MATLAB script that can solve any linear system of equations using Gaussian elimination and back substitution. To use the script, simply run it in MATLAB and enter the coefficient matrix A and the constant terms b when prompted. The script will then solve the system of equations and print the solution vector x.

Note that the script assumes that the coefficient matrix is square and that there is a unique solution to the system of equations. If these assumptions are not met, the script will produce an error message.

```
%------------------------------------------------------------
% >>> Solving a linear equation system <<<
% by Jairo Guerrero, University of Nariño
%------------------------------------------------------------

A = input('Enter the coefficient matrix A: ');
b = input('Enter the constant terms b: ');
[m, n] = size(A);
if m ~= n
    error('The coefficient matrix must be square. ');
end
A = [A b];
for i = 1:n-1
    if A(i,i) == 0
        error( 'Zero pivot encountered. The system has no
unique solution. ');
    end
    for j = i+1:n
        factor = A(j,i)) / A(i,i);
        A(j,:) = A(j,:) - factor * A (i,:);
    end
end
x = zeros(n,1);
x(n) = A(n,n+1) / A(n,n);

for i = n-1:-1:1
    x(i) = (A(i,n+1) - A(i,i+1:n)*x(i+1:n)) / A(i,i);
end
fprintf( ' The solution is: \n');
disp(x);
%------------------------------------------------------------
```

For understanding the MATLAB script above, we need to check the code lines inside. This script solves a linear equation system of the form Ax = b using Gaussian elimination and back substitution. Here's how it works:

- The script prompts the user to input the coefficient matrix A and the constant terms b.
- The script checks that the coefficient matrix is square, meaning that it has the same number of rows and columns. If the matrix is not square, the script produces an error message.
- The script augments the coefficient matrix A with the constant terms b.
- The script performs Gaussian elimination to transform the augmented matrix A into row echelon form, with all elements below the diagonal set to zero.
- The script performs back substitution to solve for the unknowns x. Starting from the last row of the row echelon form, it solves for each unknown x(i) in terms of the previously solved unknowns x (i+1: n).
- Finally, the script prints the solution vector x.

8.1 WORKSHOP #4 — ARRAYS IN MATLAB

This workshop will work on arrays. An array—matrix—is an ordered collection of data—either primitives or objects depending on the language—. Arrays are used to store multiple values in a single variable, as opposed to variables that can only store one value—for each variable—. Each element of the array—matrix—has a number associated with it, called a "numeric index", which allows you to access it.

Proposed Exercises

• Write a script that allows to obtain the number of positive elements of an array (one-dimensional vector). Populate said vector with random values between -100 and 100.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case "Write a script that allows to obtain the number of positive elements of in an array."

```
disp(" Write a script that allows to obtain the number of
positive element of in an array")
```

Step 3: A variable **n** is created to determine the number of elements with which the vector will be filled, to enter the value it is done with the command **input.**

```
n = input ("enter the number of elements to populate the
vector: ")
```

Step 4: A vector (**v**) is created with the **randi** function that allows to obtain the random numbers between -100 and 100 with which the vector will be filled. **n** is the number of elements that the vector will have.

```
v = randi([-100,100], 1, n);
```

Step 5: To determine the number of positive elements in the vector, the **positive** variable is created and the condition that vector is greater than zero is added **(v>0)** to count only the positive elements.

```
disp("The number of positive elements is ")
positive = sum(v>0)
```

Step 6: Finally, some tests are performed in **Command Window** to check the correct functioning of the program. Also, the variables used can be consulted in the **Workspace.**

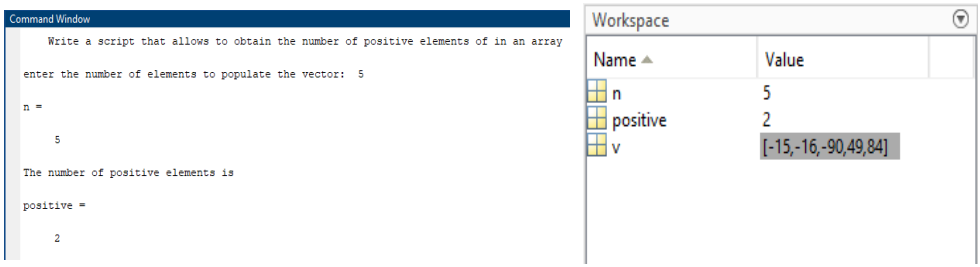


Figure 8-1. Command window and Workspace. ARRAYS: obtain the number of positive elements of an array.

• Fill a 4x4 identity matrix, element by element in an algorithmic way by using loops. Then create the same identity matrix with pre-defined functions.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case "Fill a 4x4 identity matrix, element by element".

```
disp("Fill a 4x4 identity matrix, element by element")
```

Step 3: variable **M** is created, which is a 4x4 matrix of zeros that will allow the creation of the identity matrix later.

```
M = zeros(4, 4);
```

Step 4: Subsequently, two loops **for** are used to create the rows and columns of the vector, and a conditional **if** is added to make it an identity matrix, when the row is equal to the column (**i=j**) a one is added, otherwise the rest is zero. Finally, **M** is printed.

```
for i = 1:4
    for j = 1:4
        if i == j
            M(i,j) = 1;
        else
            M(i,j) = 0;
        end
    end
end
M
```

Step 5: to perform the matrix identity with the default functions of the **eye** command and add the matrix dimension.

```
disp("with pre-defined functions")
disp(eye(4))
```

Step 6: Finally, some tests are performed in **Command Window** to check the correct functioning of the program. Also, the variables used can be consulted in the **Workspace.**

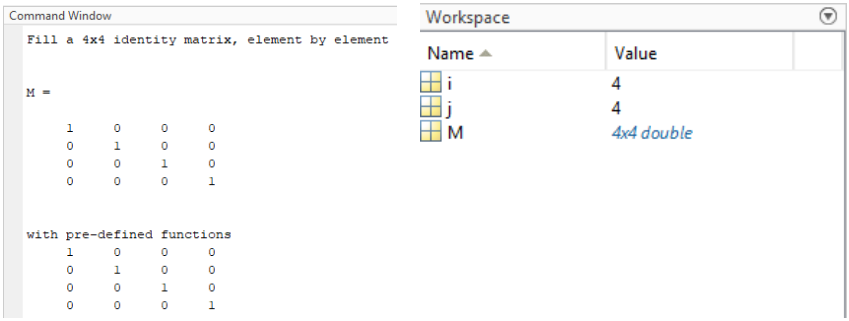


Figure 8-2. Command window and Workspace. ARRAYS: Fill a 4x4 identity matrix.

• Fill a 3 x 3 matrix of numbers typed by the user. Reading the elements from such a matrix and calculate the sum of each of its rows and columns, leaving these results in two vectors, one of the sums of the rows and another of the columns.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case "*Fill a 3x3 matrix of numbers typed by the user and sum and add up the elements of the rows and columns*".

```
disp("Fill a 3x3 matrix of numbers typed by the user and sum
and add up the elements of the rows and columns ")
```

Step 3: Variable **M** is created, which is a 3x3 matrix of zeros that will later be used to fill it with the values that the user enters by keyboard.

```
M= zeros(3, 3);
```

Step 4: Two **for** loops are performed to determine the rows and columns of the matrix, **i** corresponds to the **rows** and **j** are the **columns**, for the user to enter the values in each corresponding position the **input** command is added, also the position in which each value is entered is printed and the values are added to the matrix **M**. Finally, the result of the matrix is printed.

```
for i = 1:3

    for j=1:3
        fprintf('enter the element (%i,%i', i, j);
        M(i,j) = input(':');
    end

end
disp("Entered matrix")
M
```

Step 5: The next step is to create the row and column sum vectors, for the column sum vector we use the command **sum(M)**, which directly performs the sum of the columns and adds them to the **vsc** vector, for the arrows sum we use the command **sum (M,2)** which performs the sum of the rows but adds them to a column vector, so we add the **transpose** command to convert the **vsr** vector to a row vector. finally, we print the results.

```
disp(" ")
disp("Vector sum columns")
vsc = sum(M)

disp(" ")
disp("vector sum rows")
vsr = transpose(sum(M,2))
```

Step 6: Finally, some tests are performed in **Command Window** to check the correct functioning of the program. Also, the variables used can be consulted in the **Workspace.**

```
Command Window
Fill a 3x3 matrix of numbers typed by the user and sum and add up the elements of the rows and columns

enter the element (1,1):1
enter the element (1,2):2
enter the element (1,3):3
enter the element (2,1):4
enter the element (2,2):5
enter the element (2,3):6
enter the element (3,1):7
enter the element (3,2):8
enter the element (3,3):9
Entered matrix

M =

     1     2     3
     4     5     6
     7     8     9
```

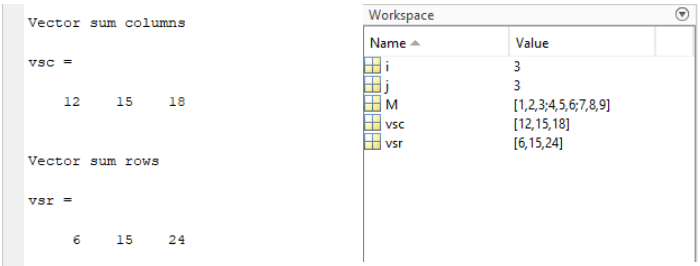


Figure 8-3. Command window and Workspace. ARRAYS: Fill a 3x3 matrix of numbers typed by the user

• Calculate the sum of all the elements of a vector of n elements of random numbers between 1 and 100, as well as the arithmetic mean (average); the calculations will be performed algorithmically step by step. At the end, use predefined functions.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case "*Calculate the sum of all the elements of a vector of n elements of random numbers between 1 and 100 as well as the arithmetic mean (average)"*

```
disp("Calculate the sum of all the elements of a vector of n
elements of random numbers between 1 and 100")
disp("as well as the arithmetic mean (averange)")
```

Step 3: The variable **n** is created with the input command so that the user can enter the number of values to fill the vector, also with the **randi** function the values will be added to the vector (**v**) with random numbers between 1 and 100.

```
n = input("enter the number of elements to populate the
vector: ")
v = randi([1,100],1,n);
```

Step 4: to calculate the sum and arithmetic mean with a predefined function, **sum(v)** and **mean(v)** were used and the values were stored in the variables **M** and **S1**.

```
M= mean(v);
S1 = sum(v);
```

Step 5: To perform the algorithm step by step, we define a counter **S** that allows us to keep the sum of the elements in the vector, we create a **for** loop to go through the vector (**v**), perform the sum and store the values in **S**; the sum is performed with **S = S + v(i)**, once this sum is obtained, the arithmetic mean is calculated with **S/n** and stored in the variable **mean**, the arithmetic mean corresponds to the sum of the elements of the vector over the number of elements. finally, the results are printed.

```
S = 0;
for i = 1:length(v)

    S = S + v(i);
end
disp("The sun is")
disp(" ")
S

disp("The arithmetic mean (averange) is: ")
mean = S/n
```

Step 6: The values obtained with the default functions are printed out.

```
disp("with predefined functions")
disp("The sum is")
S1
disp(" ")
disp("The arithmetic mean (averange) is: ")
M
```

Step 7: Finally, some tests are performed in **Command Window** to check the correct functioning of the program. Also, the variables used can be consulted in the **Workspace.**

```
Calculate the sum of all the elements of a vector of n elements of random numbers between 1 and 100
as well as the arithmetic mean (average)

enter the number of elements to populate the vector:  5

n =

     5

The sum is


S =

   189

The arithmetic mean (average) is:

mean =

   37.8000


with predefined functions
The sum is

S1 =

   189


The arithmetic mean (average) is:

M =

   37.8000
```

Workspace

Name	Value
i	5
M	37.8000
mean	37.8000
n	5
S	189
S1	189
v	[72,52,13,15,37]

Figure 8-4. Command window and Workspace. ARRAY: Calculate the sum of all the elements of a vector of n elements of random numbers between 1 and 100.

• Create a 4 x 4 matrix of numeric values typed by the user. Create a new matrix as the transposed matrix step by step algorithmically. Finally use the predefined functions.

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case "*Create a 4x4 matrix of numeric values typed by the user. Create a new matrix as the transposed.*"

```
disp("Create a 4x4 matrix of numeric values typed by the user.
Create a new metrix as the transposed")
```

Step 3: Variable **M** is created, which is a 4x4 matrix of zeros that will later be used to fill it with the values that the user enters by keyboard.

```
M = zeros(4,4;)
```

Step 4: Two **for** loops are performed to determine the rows and columns of the matrix, **i** corresponds to the rows and **j** are the columns, for the user to enter the values in each corresponding position the input command is added, also the position in which each value is entered is printed and the values are added to the matrix **M(i,j)**. Finally, the result of the matrix is printed.

```
for i = 1:4

    for j = 1:4
        fprintf('enter the element (%i,%i)', i, j);
        M(i,j) = input(':');
    end
end
```

Step 5: for the transposed matrix we create again two for loops that will allow us to add the elements to the rows and columns of the matrix, but now we define **T(i,j) = M(j,i)** so that the columns become rows and the condition of the transposed matrix.

```
for i = 1:4

    for j = 1:4
        T(i,j) = M(j,i);
    end
end
```

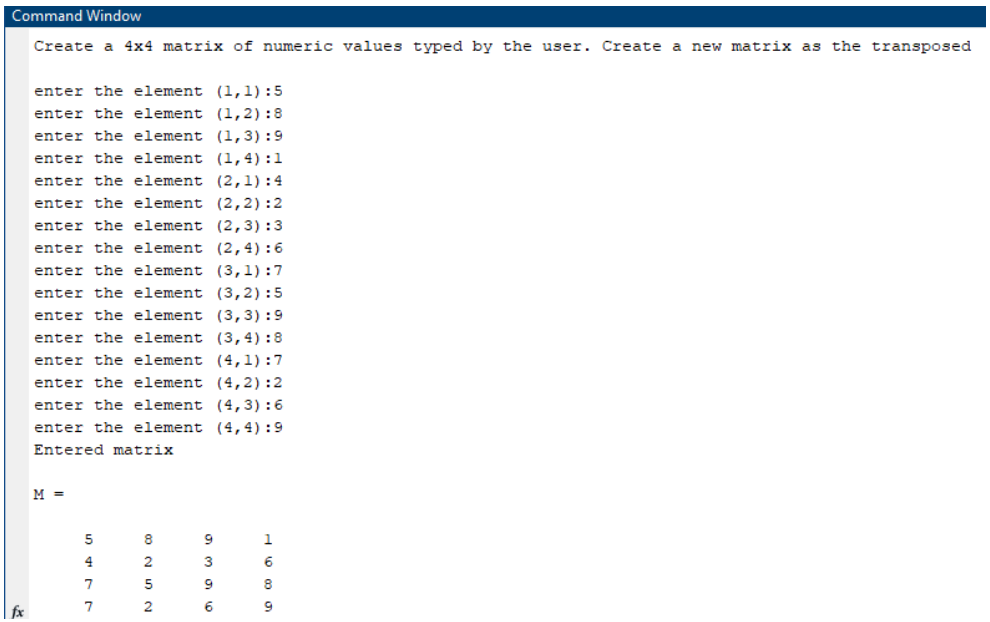
Step 6: for this step, the normal and transposed matrices are printed on the screen.

```
disp("Entered matrix")
M
disp(" ")
disp("transposed matrix")
T
```

Step 7: To calculate the transposed matrix with default function, the **transpose** command is used.

```
disp("with pre-defined functions")
Trans= transpose(M)
```

Step 8: Finally, some tests are performed in **Command Window** to check the correct functioning of the program. Also, the variables used can be consulted in the **Workspace.**



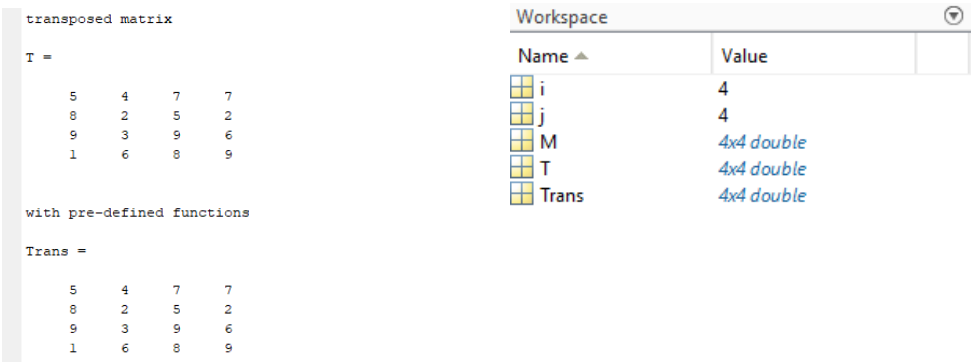


Figure 8-5. Command Window and Workspace. ARRAYS: Create a 4x4 matrix of numeric values typed by the user.

• Write a script that allows to type a number N. Create a NxN square matrix in order to represent the Pascal's triangle within it, for instance:

Table 8-1. Example of square matrix of order 6

N = 6

1	0	0	0	0	0
1	1	0	0	0	0
1	2	1	0	0	0
1	3	3	1	0	0
1	4	6	4	1	0
1	5	10	10	5	1

Step 1: we open a new script and execute the **clc**, **clear**, and **close** commands to clear the screen and delete variables.

```
clc; clear; close all; %Clear sreen and delete variables
```

Step 2: with **disp** command we print the title of project, in this case "*Create a NxN square matrix in order to represent the Pascal's triangle".*

```
 disp("Create a NxN square matrix in order to represent the
Pascal's triangle")
```

Step 3: A variable **N** is created with the input command for the user to define the dimension of the square matrix NxN.

```
N = input("index to create the square matrix :");
```

Step 4: To create the pascal triangle we use two for lop, the first one allows us to determine the row in which the values are located and for the second for and determine the values that will be found in each row we use the following equation:

$$PT = \frac{i!}{j!\,(i-j)!}$$

The values calculated in the equation are stored in the vector **a(1,j+1)**, the rows do not change, what we are interested in knowing are the values calculated in j, therefore **j+1** is added.

Then we create the cell **A{i+1}** that stores and organizes the vectors to form the pascal triangle, we also set a=**0** so that each time the vector is reset and stores the new values.

```
for i = 0:N

    for j = 0: i
        a(1,j+1)=factorial(i)/(factorial(j)*factorial(i-j));
    end
    A{i+1} = a;
    a = 0;
end
```

Step 5: to print the pascal triangle, a **for** loop is created to display on the screen what we have stored in cell **A.**

```
disp("Pascal's triangle is: "
for k = 1:N
    disp(A{k})
end
```

Step 6: Finally, some tests are performed in **Command Window** to check the correct functioning of the program. Also, the variables used can be consulted in the **Workspace.**

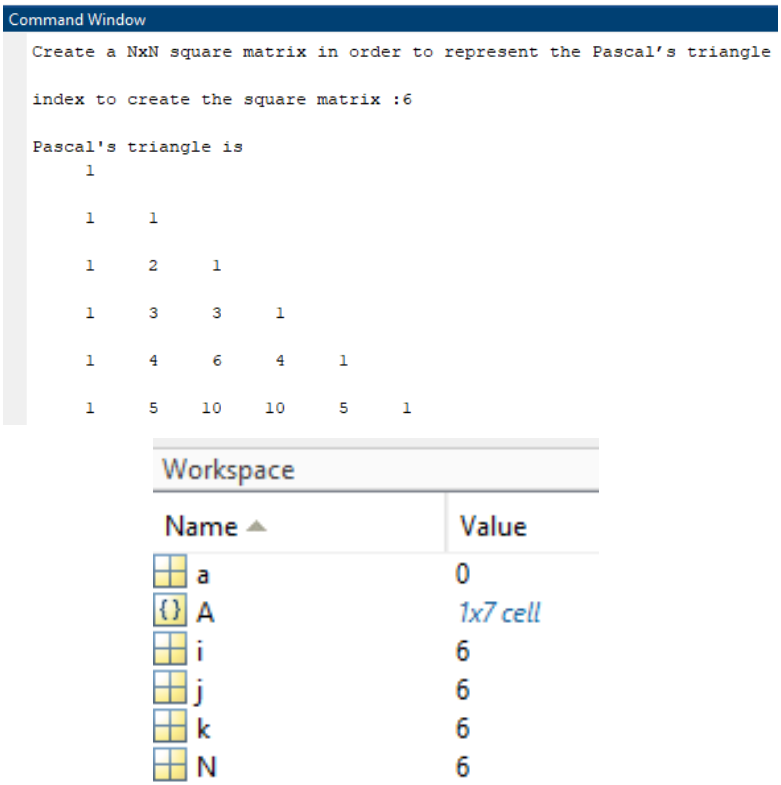


Figure 8-6. Command Window and Workspace. ARRAY: Create a NxN square matrix in order to represent the Pascal's triangle.

MATLAB is a high-level programming language widely used for numerical computing and scientific data analysis. Some of the main characteristics of programming with MATLAB are:

Easy to use: MATLAB is known for its user-friendly syntax and interactive development environment. Its simple and intuitive syntax allows developers to express complex mathematical operations with ease.

Powerful data visualization capabilities: MATLAB provides powerful tools for data visualization, making it easy to create plots, graphs, and other visual representations of data.

Numerical computing capabilities: MATLAB is designed for numerical computing, making it easy to perform complex calculations and manipulate large data sets. It comes with a wide range of built-in functions for linear algebra, statistics, optimization, and signal processing.

Wide range of toolboxes: MATLAB provides a wide range of toolboxes that extend its functionality for specialized applications. These toolboxes include image processing, control systems, signal processing, and optimization.

Interoperability: MATLAB can be easily integrated with other programming languages and applications, such as Python, Java, C, and Excel.

Community support: MATLAB has a large and active community of users and developers, providing access to a wealth of resources, including forums, tutorials, and documentation.

Overall, MATLAB is a powerful and versatile programming language that is well-suited for numerical computing and data analysis in a wide range of fields, including engineering, science, and finance.

This book was created as an important part of academia for several reasons:

Learning resource: This book provides a comprehensive and organized resource for students to learn from. They cover a range of topics and provide a structured approach to understanding a subject, in this case: Mathematical Foundations in computer programming. This book can also provide examples, explanations, and illustrations that help students better understand the material.

Reference material: This book can serve as a reference material for students throughout their academic careers related to computing. They can be used to refresh knowledge, review concepts, and prepare for exams.

Integration: This book can integrate multiple disciplines, theories, and approaches to a subject, but always focusing on mathematical foundations for computing. This can provide a holistic understanding of a topic, and help students connect different ideas and concepts.

Overall, this book is an important resource in academia that provide a structured approach to learning, standardize information, serve as a reference material, and can integrate multiple disciplines related with computing.

ACM & IEEE-CS. (2020). Computing Curricula 2020 CC2020, Paradigms for Global Computing Education. [Internet] https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2020.pdf

Aigner, M. (2023). Discrete mathematics. American Mathematical Society.

Albaugh, L., McCann, J., Yao, L., & Hudson, S. E. (2021). Enabling Personal Computational Handweaving with a Low-Cost Jacquard Loom. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. 1-10.

Avella-Medina, M. (2020). The role of robust statistics in private data analysis. Chance, 33(4), 37-42.

Baker, A. (2022). Transcendental number theory. Cambridge university press.

Bowen, J. (2019). The impact of Alan Turing: Formal methods and beyond. Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures 4, 202-235.

Farin, G. & Hansford, D. (2021). Practical linear algebra: a geometry toolbox. Chapman and Hall/CRC.

Fortuna, L., Frasca, M., & Buscarino, A. (2021). Optimal and robust control: Advanced topics with Matlab®. CRC press.

Haecker, R. (2022). Sacramental Engines: The Trinitarian Ontology of Computers in Charles Babbage's Analytical Engine. Religions, 13(8), 757.

Hosseini, M., Ouaknine, J. & Worrell, J. (2019). Termination of linear loops over the integers. arXiv preprint arXiv:1902.07465.

Kidron, I. (2020). Calculus teaching and learning. Encyclopedia of mathematics education, 87-94.

Kirchner, D., Benzmüller, C. & Zalta, E. (2019). Computer science and metaphysics: A cross-fertilization. Open Philosophy, 2(1), 230-251.

Kochenderfer, M. & Wheeler, T. (2019). Algorithms for optimization. Mit Press.

Kossovsky, A. (2020). The Bitter Dispute with Leibniz over Calculus Priority. The Birth of Science, 161-161.

Kumar, S., Azar, A., Inbarani, H., Liyaskar, O. & Almustafa, K. (2019). Weighted Rough Set Theory for Fetal Heart Rate Classification. International Journal of Sociotechnology and Knowledge Development (IJSKD), 11(4), 1-19.

Macrae, N. (2019). John von Neumann: The scientific genius who pioneered the modern computer, game theory, nuclear deterrence, and much more. Plunkett Lake Press.

Matloff, N. (2019). Probability and statistics for data science: Math+ R+ data. CRC Press.

Magoun, A. (2019). The Mystery of Claude Shannon's Personal Computer. In 2019 6th IEEE History of Electrotechnology Conference (HISTELCON). 85-86

Maričić, S. & Lazić, B. (2020). Abacus computing tool: From history to application in mathematical education. Inovacije u nastavi-časopis za savremenu nastavu, 33(1), 57-71.

Martínez, F., Martínez, I., Kaabar, M., Ortíz-Munuera, R. & Paredes, S. (2020). Note on the conformable fractional derivatives and integrals of complex-valued functions of a real variable. IAENG International Journal of Applied Mathematics, 50(3), 609-615.

Miller, S. & Takloo-Bighash, R. (2021). An invitation to modern number theory. In An Invitation to Modern Number Theory. Princeton University Press.

Mount, J. & Zumel, N. (2019). Practical data science with R. Simon and Schuster.

National Academies of Sciences, Engineering, and Medicine. (2019). Quantum computing: progress and prospects.

O'Regan, G. (2013). Giants of computing. Springer

O'Regan, G. (2018). World of computing. Springer

Potters, M., & Bouchaud, J. P. (2020). A First Course in Random Matrix Theory: For Physicists, Engineers and Data Scientists. Cambridge University Press.

Pucik, A. (2022). Not Damsels in Distress: Women and the Video Game Industry. Arkansas State University.

Roth Jr, C., Kinney, L. & John, E. (2020). Fundamentals of logic design. Cengage Learning.

Rushdi, A. (2023). An Overview of Recent Developments in Big Boolean Equations. arXiv preprint arXiv:2302.09118.

Steiglitz, K. (2020). Digital Signal Processing Primer. Courier Dover Publications.

Sporns, O. (2022). Graph theory methods: applications in brain networks. Dialogues in clinical neuroscience.

Sprunger, D. & Jacobs, B. (2019). The differential calculus of causal functions. arXiv preprint arXiv:1904.10611.

The MathWorks. (2023). MATLAB & Simulink. [Internet]
https://www.mathworks.com/products/matlab.html

Tissenbaum, M., Sheldon, J. & Abelson, H. (2019). From computational thinking to computational action. Communications of the ACM, 62(3), 34-36.

Trefethen, L. & Bau, D. (2022). Numerical linear algebra (Vol. 181). Siam.

Ulmann, B. (2022). Analog computing. Walter de Gruyter GmbH & Co KG.

West, D. (2020). Combinatorial mathematics. Cambridge University Press.

Woodford, C. (2021). A brief history of computers. Explain that Stuff.

Table of Figures

Figure 4 1. MATLAB main screen ……..…..... 33

Figure 5 1. Command Window –
Exercise 1 Comparisons in MATLAB 42
Figure 5 2. Command Window –
Exercise 2 Comparisons in MATLAB 43
Figure 5 3. Command Window and Workspace –
Exercise 3 Comparisons in MATLAB 45
Figure 5 4. Command Window and Workspace –
Exercise 4 Comparisons in MATLAB 47
Figure 5 5 Command Window –
Exercise 5 Comparisons in MATLAB 48
Figure 5 6. Command Window –
Exercise 6 Comparisons in MATLAB 51
Figure 6 1. MATLAB script calculates the factorial of
a positive integer entered by the user 55
Figure 6 2. MATLAB script calculates the divisors of
a given number entered by the user 56
Figure 6 3. MATLAB script generates a random
integer between 1 and 100 58
Figure 6 4. Comand window and Workspace.
LOOPS IN MATLAB: Read a series of non-zeros numbers 64
Figure 6 5. Command window and Workspace.
LOOPS IN MATLAB: Calculate and display thesum and
product of the even numbers between 20 and 400 65
Figure 6 6. Command window and Workspace.
LOOPS IN MATLAB: Calculate the sum of the squares 66
Figure 6 7. Command Window and Workspace.
LOOPS IN MATLAB: Add ten numbers entered by keyboard 68
Figure 7 1. Icon for creating a new script 76

Figure 7 2. Command window. FUNCTIONS:
Conversion Temperature 79

Figure 7 3. Command Window. FUNCTIONS:
Conversion Coordinates Polar - Cartesian 82
Figure 7 4. Command Window. FUNCTIONS: DrakeE 84

Figure 7 5. Command window. FUNCTIONS:
Determinate if a number is prime 86
Figure 8 1. Command window and Workspace. ARRAYS:
obtain the number of positive elements of an array. 94
Figure 8 2. Command window and Workspace. ARRAYS:
Fill a 4x4 identity matrix. 95
Figure 8 3. Command window and Workspace. ARRAYS:
Fill a 3x3 matrix of numbers typed by the user 98
Figure 8 4. Command window and Workspace. ARRAY:
Calculate the sum of all the elements of a vector of n
elements of random numbers between 1 and 100. 100
Figure 8 5. Command Window and Workspace. ARRAYS:
Create a 4x4 matrix of numeric values typed by the user. 103
Figure 8 6. Command Window and Workspace. ARRAY:
Create an NxN square matrix to represent Pascal's triangle. 105

Table of Tables

Table 5-1. Truth table based on three logical operators 36

Table 8-1. Example of a square matrix of order 6 103

êditorial
Universidad de **Nariño**

Publication date: October 2025
San Juan de Pasto - Nariño - Colombia

This book serves as a comprehensive guide for understanding the mathematical foundations critical to computer programming. It underscores the intrinsic relationship between mathematics and Computer Science, emphasizing that programming, as a discipline, cannot be effectively mastered without a solid grasp of underlying mathematical principles. The author argues that MATLAB is particularly well-suited for this educational purpose due to its power in scientific computing and its ability to illustrate complex mathematical concepts clearly. By using MATLAB as the central tool, the book provides an accessible and practical approach for students and educators seeking to bridge theoretical mathematics with hands-on programming applications.

Aimed primarily at students and educators in Computer Science, the book is designed to reinforce academic learning by integrating mathematical theory into programming practice. It is intentionally written in English to familiarize students with the language predominantly used in scientific and technical literature. Drawing on the author's extensive experience as a professor at the University of Nariño, the content reflects a deep understanding of how mathematical thinking enhances computational problem-solving. By highlighting key historical moments where mathematics shaped computing innovations, the book situates its lessons within a broader context, reinforcing the ongoing relevance of mathematical literacy in modern software development.

Furthermore, the book promotes effective learning strategies, such as manually typing code, to enhance comprehension and retention. Citing educational research, the author explains how this practice deepens engagement, fosters critical thinking, and develops essential debugging and problem-solving skills. This mindful interaction with the material not only solidifies programming knowledge but also builds learner autonomy and confidence. Ultimately, the book equips readers with both theoretical insights and practical tools, empowering them to tackle complex computational challenges and laying a strong foundation for advanced studies in scientific computing and software engineering.

ISBN: 978-628-7771-82-6

9 786287 771826

Universidad de Nariño

êditorial Universidad de Nariño