Synthesis Lectures on Computer Science

SYNTHESIS
COLLECTION OF TECHNOLOGY

Paul A. Gagniuc

Coding Examples from Simple to Complex

Applications in MATLAB®

Springer

Synthesis Lectures on Computer Science

The series publishes short books on general computer science topics that will appeal to advanced students, researchers, and practitioners in a variety of areas within computer science.

Paul A. Gagniuc

Coding Examples from Simple to Complex

Applications in MATLAB®

Springer

Paul A. Gagniuc
Department of Engineering in Foreign
Languages
Faculty of Engineering in Foreign Languages
National University of Science and Technology
Politehnica Bucharest
Bucharest, Romania

ISSN 1932-1228 ISSN 1932-1686 (electronic)
Synthesis Lectures on Computer Science
ISBN 978-3-031-53804-9 ISBN 978-3-031-53805-6 (eBook)
https://doi.org/10.1007/978-3-031-53805-6

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Foreword

The book *Coding Examples from Simple to Complex—Applications in MATLAB®* by Paul Aurelian Gagniuc is a very hands-on introduction to programming in MATLAB, appealing to readers ranging from novices making their first steps in the universe of programming to more seasoned developers, that can use a very rich reference of code examples. Because this is the main feature of this work, teaching through examples, over 200, each chapter exemplifying the key concepts by exercises which are implemented, commented, and explained in great detail.

The chosen language is MATLAB, a programming language oriented towards scientists and engineers, focusing especially on numeric computation. This makes the book especially useful for students in engineering domains, such as mechanical engineering and electrical engineering, who often lack an introductory course in their language of choice.

The structure is well-thought, starting with traditional starting points in variable declaration, expressions, control statements, arrays, functions and continuing with objects and advanced techniques.

The examples support the chapters in a logical succession, one advantage being that a simplified solution is shown before an optimized one, useful for a deeper understanding of the problem.

The book continues with the moderate examples section, in which more real-world usages are shown, ranging from topics such as string manipulation, more advanced matrix operations, sorting algorithms, bitwise operations and encodings and statistics. As examples are implemented without the use of other libraries except standard library, they are of great teaching value, in helping practitioners truly understand the inner workings of concepts.

Where the book is of interest to more advanced developers or researchers in different fields, is in the complex examples section, covering novel, state-of-the-art algorithms such

as spectral forest or complex usage of Markov Chains, an area in which the author is a renowned expert.

Andrei Vasilateanu
Professor
Faculty of Engineering in Foreign
Languages
National University of Science
and Technology Politehnica Bucharest
Bucharest, Romania

Preface

Among the fields involving numerical computing and advanced data analysis, MATLAB stands as the cornerstone of modern engineering and scientific computation. Investigate the rich ecosystem of MATLAB with this work, a comprehensive guide that escorts the reader on an excursion from the fundamentals to advanced applications, further arming the reader with the knowledge and skills needed to become a proficient MATLAB user. From the rudiments of variable naming and program structure to complex data structures, algorithms, and graphical user interfaces, this book covers it all.

Key Features

Hands-on Learning. Engage with over 200 examples, thoughtfully designed to reinforce your grasp of MATLAB concepts and numerical computing.

Comprehensive Coverage. Traverse through the essentials of MATLAB, including matrix manipulations, plotting, control statements, functions, and advanced data types.

Advanced Techniques. Enhances the knowledge of the reader with sophisticated topics on optimization, simulation, statistical analysis, and machine learning in MATLAB.

Real-World Applications. Look into the practical uses of MATLAB in various domains such as signal processing, image analysis, and control systems. Learn also how to implement complex mathematical models in code.

Throughout the ensuing chapters, readers will cultivate a deeper understanding of MATLAB and its vast potential across numerous applications. This comprehensive exploration caters to novice learners and seasoned researchers alike, equipping them with the necessary acumen to use the MATLAB's full potential in their projects. Thus, unfolding the layers of MATLAB programming as a systematic and rigorous adventure into the world of technical computing.

Bucharest, Romania — Paul A. Gagniuc

Contents

Introduction

1

MATLAB, an abbreviation for "Matrix Laboratory," is a high-level computer language and an interactive environment used by engineers and scientists worldwide. It was developed by *MathWorks* and first released in the mid-1980s. MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, the creation of user interfaces, and intercommunication with programs written in other languages. The platform was created to provide an easy and efficient environment for numerical computation, visualization, and programming. With its built-in mathematical functions, MATLAB became popular among engineers and scientists for technical computing. The capabilities of this language have expanded over time to include data analysis, development of algorithms, modeling, simulation, and prototyping. The MATLAB ecosystem has significantly expanded its capabilities, allowing for specialized applications in fields such as signal processing, image and video processing, control systems, test and measurement, computational finance, and computational biology. Among these toolboxes, *Simulink*, a MATLAB-based graphical programming environment for modeling, simulating, and analyzing multidomain dynamical systems, stands out for its impact on various engineering fields. Thus, this language has been instrumental in shaping the way complex mathematical problems are approached and solved in various industries. Obviously, one of its primary advantages is the ease with which users can express computational mathematics. The platform is particularly popular among engineers and scientists for its robust toolboxes that simplify tasks such as signal processing, image analysis, and control systems design, among others. Another strong point of MATLAB is its rich library of built-in functions for both numerical analysis and symbolic computations. Thus, as the reader will witness in the chapters below, the MATLAB environment is highly optimized for matrix operations, which makes it particularly efficient for problems and algorithms that are

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_1

matrix and vector-based. However, MATLAB can be considered less efficient for general-purpose programming due to its relatively slow execution speed for certain types of tasks, especially when compared to compiled languages like *C* or *Fortran*. It is also proprietary software, which may limit its accessibility to individuals or institutions on a tight budget, as the cost of licenses can be prohibitive. The syntax and functionality of MATLAB are unique, and while this is an advantage in its specific context, it can be a disadvantage for users who need to integrate or migrate to different programming environments, as the portability of MATLAB code to other computer languages is not straightforward.

1.1 Future of MATLAB

Today, MATLAB is a cornerstone of engineering and scientific computing, and it enables professionals to analyze data, develop algorithms, and create models that are integral to various high-tech fields. The versatility of MATLAB has led to its adoption across various domains, including signal processing, control systems, computational finance, and computational biology. As for the future, MATLAB is expected to grow in several key areas. Its compatibility with high-performance computing environments is increasing, and with technologies such as parallel computing and GPU-enabled functions, MATLAB allows researchers and developers to run computationally intensive code more efficiently. With the advent of powerful toolboxes for machine learning and artificial intelligence (AI), MATLAB is well-positioned to drive advancements in these fields. Libraries like *Deep Learning Toolbox*™ (formerly *Neural Network Toolbox*™) provide a comprehensive platform for neural networks and deep learning in MATLAB. Users can train models with an extensive set of algorithms designed for applications ranging from image recognition to time series prediction. Moreover, the role of MATLAB in data science is expanding, as it offers a robust environment for statistical analysis and visualization, enabling researchers to handle large datasets and complex analyses. Also, it is expected to continue its integration with big data platforms and databases, providing tools for accessing and processing massive sets of information efficiently. Moreover, the potential of MATLAB for the *Internet of Things* (IoT) and embedded systems, is also noteworthy. The language offers an array of features for designing, prototyping, and deploying IoT applications, with support for connecting to a wide range of hardware devices. In the area of education, MATLAB is likely to maintain its status as a fundamental teaching tool for scientific and engineering disciplines, introducing students to mathematical modeling, simulation, and the principles of algorithm design. One area of future development that is unexpected for MATLAB is the field of security, were algorithms are improved to address new computational challenges. MATLAB adaptability and continuous development ensure that it will remain a vital tool for engineers, scientists, and researchers, driving the next generation of scientific discovery and technological breakthroughs. The future of MATLAB is indeed bright, as it continues to empower smart people to turn their ideas into reality.

1.2 The Content is Native

This work showcases native MATLAB implementations from basic to complex, and is addressed to a large audience, from beginners to PhD students and even experienced scientists and engineers. The first part of this book describes the use of variables, conditional branching, and loops. Variables, as foundational elements of computer languages, form the focus of the first chapter. Topics covered include variable initialization, nomenclature conventions, and the composition of a basic MATLAB program. Additionally, discussions will encompass assignment, variable types, fundamental arithmetic operations, and related subjects. Also, conditional branching mechanisms, which facilitate decision-making processes and the execution of different code segments based on predetermined conditions, are explored in detail. Emphasis is placed on a variety of conditional statements such as "if," "if-else," "if-elseif-else," and the "switch-case" construct. These fundamental constructs enable the manipulation of program flow and responsiveness to various scenarios. Next, the concept of loops is explored in detail, as it is instrumental in iteratively executing code blocks and enhancing program efficiency. A comprehensive exploration of both "while" and "for" loops is undertaken. Topics of interest include count-controlled loops, vector and matrix traversal, and complex mathematical computations. The utilization of control statements like "break" and "continue" within loops is also addressed. In the second part of the book, more complex variables such as matrices and cell arrays are described by example. The subject of multidimensional traversal of matrices is also covered, and then some matrix operations are shown. Matrices, as fundamental data structures for organizing and manipulating numerical data, are scrutinized in a dedicated chapter. Topics encompass basic matrix operations such as element addition and retrieval, dimension calculation, and matrix traversal. The employment of various loop types for matrix traversal is discussed in detail. Moreover, the traversal and manipulation of these multidimensional matrices are explored comprehensively. The discussion extends to encompass 2D and 3D matrices, matrix operations, and transformations including transposition and rotation. Furthermore, matrix operations are shown as pivotal in mathematical contexts and for applications such as image processing. Subjects addressed include summation, multiplication, eigenvalues and eigenvectors extraction, inversion, and related matrix operations. In the third part of this book, functions and object-oriented programming are thoroughly explored from several perspectives. Functions, instrumental in code reusability and modularity, are the primary focus of an extensive chapter. Both built-in and user-defined functions are explored in depth. Topics encompass function creation, argument passing, and return value handling. Additionally, discussions extend to recursion, logical operations, optimization algorithms, statistical computations, and diverse practical examples showcasing the versatility of functions in MATLAB. The conceptualization and implementation of classes, their properties, and methods are expounded upon in a separate chapter. Class definitions, object instantiation, and the inclusion of methods within classes are thoroughly explored. Practical examples underscore the principles of object-oriented

programming in MATLAB. In the fourth part of the book, the reader encounters moderate and complex examples that demonstrate the capabilities of MATLAB in tackling real-world problems. Topics include statistical analysis, numerical methods, signal processing, and data visualization, offering insights into advanced MATLAB programming techniques. The last part of the book showcases specialized MATLAB applications. This includes discussions on MATLAB interoperability with other programming languages, its powerful toolboxes for different engineering and scientific domains, and its capabilities for developing graphical user interfaces (GUIs). Additionally, readers are introduced to advanced graphics programming in MATLAB, covering the creation of sophisticated visualizations, 3D rendering, and interactive graphics using the MATLAB built-in functions and toolboxes.

1.3 MATLAB Particularities

At the core of MATLAB programming is the concept of variables, which are storage containers for data values. Variable names in MATLAB must start with a letter, followed by any combination of letters, digits, and underscores. MATLAB is case-sensitive, meaning that "Variable" and "variable" are distinct identifiers. Comments are introduced with a percent sign (%) and are essential for explaining code behavior, but they have no effect on the execution of the program. Assignment in MATLAB is denoted by the equals sign (=), where the variable on the left side is assigned the value of the expression on the right. Unlike some other computer languages, MATLAB uses a straightforward assignment statement to declare and initialize variables. Basic mathematical operations follow conventional symbols: plus (+), minus (-), multiplication (*), division (/), and exponentiation (^). The modulo operator, represented by the percent sign (%), calculates the remainder of a division between two numbers. This operator is common in many computer languages, including MATLAB. Increment operations, such as "$a = a + 1$", update the value of a variable by adding one, which is a frequent operation in loops and iterative processes. Swapping values between two variables in MATLAB does not require a temporary third variable, as is common in some languages. One can simply write [*a*, *b*] = [*b*, *a*]. In order to empty a variable, or clear its value, one sets it to an empty array using []. Line continuation in MATLAB is achieved with three dots (…), allowing for an expression to continue over multiple lines for better readability. Formatted output in MATLAB is handled through various functions, like *fprintf*, which allows for the specification of the format in which data is displayed. Conditional branching with *if–then-else* and *switch-case* statements enables decision-making processes based on the evaluation of conditions. Loop structures, such as *while*, *for*, and nested loops, facilitate the repetition of code blocks. MATLAB *for-loop*, in particular, can be used to iterate over ranges of values, including iterating backwards by decrementing a counter, or to perform summation and counting within a range. Moreover, arrays are fundamental in MATLAB, with

support for array operations including element-wise addition, multiplication, and logical operations. MATLAB powerful indexing features allow for the manipulation and traversal of arrays, extracting and modifying individual elements. Arrays can grow dynamically; their size is mutable, and one can add elements to them without the need to declare a fixed size upfront. As the name clearly indicates, MATLAB strength is also apparent in matrix operations, a natural extension of arrays. One can perform a variety of operations on matrices, from basic arithmetic to complex mathematical functions, with ease. These operations include matrix multiplication, finding the sum or maximum of elements along different dimensions, and accessing diagonals or specific parts of matrices. Traversal of multidimensional arrays is straightforward with MATLAB indexing capabilities. One can iterate over each element of a matrix or 3D array with *for loops*, or in some cases, with a single loop using arithmetic tricks for index calculation. Thus, this functionality makes it easier to perform operations on higher-dimensional data without cumbersome coding. As mentioned far above, the language offers a suite of built-in functions for both basic and complex mathematical calculations. For instance, functions like *sin* and *exp* are readily available to compute the sine of an angle or the exponential of a value respectively. In terms of array operations, MATLAB provides functions to find the maximum value, such as determining the largest element between two integers, within an array, or comparing arrays. Additionally, MATLAB supports generating arrays of random integers ranging from 0 to 99 and inserting random values into a matrix. Just like in the majority of computer languages, string handling in MATLAB includes functions for splitting strings into integers or arrays, based on specified delimiters (), and cascading operations like splitting, joining, and determining the length of strings can be performed with ease. Sorting is another fundamental operation in MATLAB, for which it has a built-in function to order the elements of an array. The creation of custom functions in MATLAB allows for encapsulation and reuse of code. Functions can accept multiple parameters, call other functions, and even support recursion, which can replace certain loops for tasks like repeating strings, summing series, calculating factorials, and generating sequences like the *Fibonacci* numbers. Objects in MATLAB are constructed with properties and methods, allowing for object-oriented programming practices. Constructors define how objects are initialized, and methods dictate the actions that these objects can perform. Complex objects can be generated and manipulated, and multiple objects can be created with their methods for use within programs. MATLAB also supports JSON data representation, allowing for conversion between JSON strings and MATLAB objects. This functionality is critical for applications that involve data interchange. Complex JSON strings can be parsed, and objects can be decomposed into arrays or matrices as required. For more involved tasks, MATLAB can define functions to load data from strings into arrays or matrices, display them appropriately, and even perform matrix operations like addition, transposition, and rotation. Logical operations such as AND, OR, XOR, and their negations are also available and are essential for decision-making processes within

code. Beyond these, MATLAB may include built-in miscellaneous functions for calculations such as logarithms, signal smoothing, and finding the greatest common divisor. Also, it can perform tasks like *alphabet detection* (nover algorithm, please see the corresponding examples) via user-defined functions and it can support more advanced algorithms and sorting techniques, including built-in *sort* and *permutations* functions. Statistical functions are integral to MATLAB, offering the capability to compute averages, standard deviations, correlation coefficients, and more. Visualization tools in MATLAB allow for these statistics to be represented graphically, such as through vertical charts, with the data being generated either statically or randomly at runtime. Conversions between different data representations, such as text to hexadecimal, are supported in MATLAB, with functions available for direct conversions, or more sophisticated ones that can detect input types and perform the appropriate conversion. MATLAB capabilities can be extended if no built-in functions exist. Thus, in this work, complex problems can be understood both natively and in a shortcut, MATLAB-like manner.

Variables

2

A variable can be conceptualized as a symbolic representation or an abstract entity that holds information [1]. This information can take various forms, from simple numerical values, strings of text, to more complex data structures. The central essence of a variable lies in its ability to change or vary, making it indispensable in algorithms and computational processes [1]. Variables are foundational to computer programming because they allow for the storage and manipulation of data. Each variable has an associated data type, which dictates the kind of information the variable can store. For instance, an integer data type variable can store whole numbers, whereas a floating-point data type might store numbers with decimal points. When a variable is declared in a computer program, a specific portion of the computer memory is allocated to store its value. This allocation ensures that when the value of the variable is called upon or modified, the program knows exactly where to look in memory. Each variable has a unique memory address, which acts like a reference point for any computational operation involving that variable. Variables also possess attributes such as scope (determining where in a program a variable can be accessed) and lifetime (indicating how long the variable remains in memory). The importance of these attributes becomes evident in more advanced programming tasks, such as managing memory or optimizing code for performance. In scientific computing, variables often represent physical quantities or abstract mathematical constructs. Their ability to change values dynamically allows for the simulation of real-world systems, from modeling the motion of celestial bodies to the prediction of weather patterns. Scientists can run multiple scenarios or simulations to analyze different outcomes and derive meaningful conclusions just by adjusting the values of these variables. Thus, variables serve as the backbone of computational algorithms and programs. Their dynamic nature, combined with the precise control they offer over data manipulation, makes them a cornerstone in

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_2

the world of computer science and scientific computing. Thus, the examples shown below start from basic exercises that familiarize the reader with the notion of variables. In MATLAB, variables are used just like in other computer languages, serving as containers to store data values:

```
% Example of variable assignment
% and manipulation in MATLAB.

% Assign values to variables
distance = 10;              % A simple numeric variable.
time = 2.5;                 % A floating-point number.
name = 'MATLAB';            % A string variable.
matrix = [1, 2; 3, 4];      % A 2x2 matrix.

% Perform calculations
speed = distance / time;    % Calculate speed.

% Display results
```

```
disp(['The calculated speed is ', num2str(speed), ' units per time.']);

% Variables can change type dynamically
% Now 'distance' is a string

distance = 'Ten';

% The access and modification of elements of an array
% Change value in the second row, first column

matrix(2,1) = 5;

% Displaying the modified matrix
disp('The modified matrix is:');
disp(matrix);
```

As mentioned many times before, MATLAB is particularly fit in handling matrices and arrays, which is reflected in its name—*MATrix LABoratory*. MATLAB is dynamically typed, which means variables do not need explicit declarations to reserve memory space. The type and size of a variable are determined automatically when the program is run, based on the assigned value. For example, if one assigns a whole number to a variable, MATLAB treats it as an integer, but if one assigns a number with a decimal point, MATLAB treats it as a floating-point number by default. Also, when one assigns a value to a new variable in MATLAB, the necessary memory is allocated to store the value. MATLAB handles the memory management behind the scenes, allowing the user to focus on the algorithms and computations without worrying about the details of memory allocation. Variables in MATLAB have a scope and a lifetime that define their visibility and duration in memory. The scope is determined by the function/script in which the variable

is created, while the lifetime of a variable extends from the time it is created until the function or script finishes execution, or until it is explicitly cleared from memory using commands like clear.

2.1.1 Ex. (1) – Commenting inside code

```
% this is a comment in Matlab.

%{ Multi-line
comments %}
```

In Matlab, the "%" character is used to denote a single-line comment. Anything that follows the "%" on that same line is treated as a comment and will not be executed or interpreted as code by the Matlab engine. Instead, it is meant to provide context or explanations for developers reading the code. Also, multi-line comments are possible, and these start with two characters, namely "%{", and end with "%}".

2.1.2 Ex. (2) – Naming variables

```
A = 1;
a = 2;
a1 = 3;
a_1 = 4;

disp(A);
disp(a);
disp(a1);
disp(a_1);

%{ Alternatively, if only variable names are
written without semicolons or disp function,
MATLAB will also print the values: %}

A = 1
a = 2
a1 = 3
a_1 = 4
```

Output:

```
1
2
3
4
```

The above code initializes four variables with distinct names and values. The variable *A* is assigned the value 1, while the variable *a* is assigned the value 2. Similarly, *a1* is given the value 3, and *a_1* is assigned the value 4. Following these assignments, the values of these variables are printed out sequentially using the *disp* function. First, the value of *A* is printed, followed by the values of *a*, *a1*, and finally *a_1*. It is worth noting that Matlab is case-sensitive, so the variable *A* is different from the variable *a*. Alternatively, if only variable names are written without semicolons and without the help of the *disp* function, MATLAB will still print out the values.

2.1.3 Ex. (3) – Write your first Matlab program

```
a = 3;
b = 5;
c = a + b;
disp(c); % display the value of c.

%{ Alternatively, one may
omit the semicolon and
the result will be printed: %}

a = 3;
b = 5;
c = a + b
```

Output:

```
8
```

The given Matlab code from above initializes a variable *a* with a value of 3 and a variable *b* with a value of 5. It then calculates the sum of these two variables and assigns the result to a third variable named *c*. Next, the value of *c* is printed to the console or displayed using a function named *disp*. The output of this code is 8. Note again that omitting the semicolon at the end of the line automatically displays the result of the operation when the script is run. However, in functions, the value will not be displayed without using *disp* or some form of output argument.

2.1.4 Ex. (4) – The meaning of "a = b"

```
a = 3;
b = a;
disp(b);
```

Output:

```
3
```

The Matlab statements from above begin by assigning the value 3 to the variable *a*. Following that, the value of *a* (which is 3) is assigned to another variable named *b*. Next, the *disp*(*b*) statement outputs the value of *b*, which would display 3. This example demonstrated the copy of data from one variable to another by assignment.

2.1.5 Ex. (5) – Assign and reassign with type change

```
% In MATLAB, one simply assigns a
% value to a variable to declare it.

% Assigns a string 'text'
% to variable a.
a = 'text';

% a = 0; % For sport, one
% can uncomment this line to
% reassign a to a numeric value.

% Assigns a string 'text'
% to variable b.
b = 'text';

% Reassigns b to
% a numeric value.
b = 0
```

Output:

```
0
```

In MATLAB, there is no special keyword for declaring a variable, and the type of a variable is automatically determined by the value assigned to it. Thus, MATLAB is dynamically typed, and the user can change the value and the type of a variable at any time. This nice feature is now present in many computer languages and is perhaps one of the most useful additions to modern computer programming because it allows software developers to focus on method rather than syntax. Therefore, unlike the old times, MATLAB will not raise any errors when the user changes the type of a variable by simply assigning a new value of a different type.

2.1.6 Ex. (6) – Basic mathematical operations

```
a = 3;
b = 2;
c = a + b / 2 - a * b;

% MATLAB can use the 'disp' function
% to display the value of a variable.

disp(c);
```

Output:

```
-2
```

The above Matlab code first assigns the value 3 to the variable *a* and the value 2 to the variable *b*. Next, it performs a series of arithmetic operations using these two variables. Specifically, it divides *b* by 2, then adds the result to *a*, and from that sum, it subtracts the product of *a* multiplied by *b*. The final result of these calculations is assigned to the variable *c*. Lastly, the value of *c* is printed out. Please notice that MATLAB will follow the standard order of operations (PEMDAS/BODMAS), thus, in this case, *b* / 2 is calculated

first, then *a* * *b*, and then the additions and subtractions are performed from left to right. Parentheses can be used in order to change the order of operations. Note that PEMDAS stands for Parentheses, Exponents, Multiplication and Division (from left to right), and Addition and Subtraction (left to right). Also, BODMAS stands for Brackets, Orders (i.e., powers and square roots, etc.), Division and Multiplication (left to right), and Addition and Subtraction (left to right).

2.1.7 Ex. (7) – The meaning of *modulo* operator

```
a = 3;
a = mod(a, 2); % Modulus of a divided by 2.
disp(a);       % Displays the value of a.
```

Output:

```
1
```

The code starts by assigning the value 3 to the variable *a*. Next, it modifies the value of *a* by setting it to the remainder when a is divided by 2, which is done using the modulus (*mod*) function. The modulus operation determines the remainder of the division of *a* by 2. Thus, 3 divided by 2 gives a quotient of 1 and a remainder of 1. Therefore, after the modulus operation, the value of *a* becomes 1. Next, it uses the *disp*(*a*) statement to display the value of *a*.

2.1.8 Ex. (8) – The meaning of "a = a + 1"

```
a = 2;
a = a + 1;
disp(a);
```

Output:

```
3
```

The given Matlab code showcases the incrementation process, and starts by assigning the value 2 to the variable *a*. It then increments the value of *a* by 1. Namely, the future value of *a* is the current value of *a* plus 1. Next, it then prints the value of *a* via the *disp* function, which would now be 3.

2.1.9 Ex. (9) – The meaning of "a = a - 1"

```
a = 2;
a = a - 1; % Decrease a by 1.
a = a - 1; % Decrease a again by 1.
disp(a);   % Print the value of a.
```

Output:

```
0
```

The Matlab code initializes a variable *a* with the value of 2. Then, it decrements the value of *a* using the "-" operator, which decreases the value of *a* by 1 before assigning

it back to *a*. This decrement operation is done twice in succession. Therefore, after the two decrement operations, the value of *a* is decreased by a total of 2. Next, the *disp(a)*; statement will display the final value of *a*, which is 0.

2.1.10 Ex. (10) – The plus operator

```
a = 2;
a = a + a; % Increment a by a.
disp(a);   % Display the value of a.

% Or, without a semicolon, MATLAB will
% display the value of a in the command window:

a = 2;
a = a + a;
a
```

Output:

```
4
```

The given Matlab code first assigns the value 2 to the variable *a*. Then, it increases the value of *a* by *a* using the "+" operator. Namely, the future value of *a* is the current value of *a* plus the current value of *a*, which is shorthand for $a = a + a$. After these operations, the value of *a* becomes 4. Next, it prints *a* using the *disp(a)* statement.

2.1.11 Ex. (11) – The minus operator

```
a = 2;
a = a - a;  % Decrement a by a.
disp(a);    % Print the value of a.
```

Output:

```
0
```

This code is complementary to the previous example and it starts by assigning value 2 to variable *a*. Next, it decreases the value of *a* by *a* using the "-" operator. Namely, the future value of *a* will be the current value of *a* minus the current value of *a*, which is short for $a = a-a$. Next, the *disp(a)* statement displays the value of *a*, which is 0.

2.1.12 Ex. (12) – Variable playground

```
a = 2;
a = a - 1;
a = a + a;
a = a - 1;
a = a + a;
disp(a);
```

Output:

```
2
```

MATLAB does not have aggregate assignments as found in other C-like computer languages. However, these can be simulated using regular operators. In this case, plus

and minus operators are used. Thus, the code from above performs a series of operations on variable a. Initially, a is assigned a value of 2. In the next line, a is decremented by 1 to become 1 (i.e. $a = a-1 = 2-1 = 1$). Then, the content of variable a is added to itself, which would result in a value of 2 (i.e. $a = a + a = 1 + 1 = 2$). Next, a is decremented by 1 again to become 1 (i.e. $a = a-1 = 2-1 = 1$). In the last operation, the content of variable a is added to itself again (i.e. $a = a + a = 1 + 1 = 2$). Thus, this MATLAB code outputs value 2 to the Command Window.

2.1.13 Ex. (13) – Swap values

```
a = 3;
b = 7;
t = 0;

t = a;
a = b;
b = t;

% In MATLAB, the swap can be
% written as: [a, b] = [b, a]

disp(['a = ', num2str(a)]);
disp(['b = ', num2str(b)]);
```

Output:

```
a = 7
b = 3
```

The given code initializes three variables: a, b, and t, with the values 3, 7, and 0 respectively. The purpose of the code is to swap the values of a and b without using any direct arithmetic operations or additional variables. To achieve this, the value of a is first stored in the temporary variable t. Then, the value of b is assigned to a, effectively overwriting a original value. Lastly, the value stored in t (which is the original value of a) is assigned to b, completing the swap. After the swapping operation, two *disp* statements display the updated values of a and b, showing that their values have indeed been exchanged. Thus, after the code is executed, the output will be "a = 7" and "b = 3". Also, in MATLAB, concatenation of strings with variables is made with square brackets [] and the *num2str* function (or *int2str*, *sprintf*, etc.) for converting numerical values to strings. Also, the print operation is typically done using the *disp* function as before (however, this function is not mandatory for printing the results to the Command Window). Nevertheless, the *disp* function is used for displaying the result, and the *num2str* function converts the numeric values of a and b to strings so they can be concatenated with the 'a = ' and 'b = ' strings for display. Swapping values between two variables in MATLAB does not require a temporary third variable, as is common in many computer languages. To make a swap, one can simply write $[a, b] = [b, a]$. Thus, this way variable t can be eliminated from the code.

2.1.14 Ex. (14) – Empty a variable

```
a = 3;
b = a + 7;
a = [];  % empty array to signify "no value".
disp(a); % Uses 'disp' to display the values.
disp(b);
```

Output:

```
null
10
```

This Matlab code initializes a variable a with the value 3. Then, it initializes another variable b and assigns it the result of adding a to 7, making the value of b equal to 10. Afterward, the value of a is set to empty array to a ($a = []$; empty represent the absence of data or no value). Lastly, the code prints the value of a, which is *null*, and then prints the value of b, which remains 10. (*disp* prints the values to the Command Window).

2.1.15 Ex. (15) – Line continuation

```
s = 1 + 2 + 3 + ...
4 + 5 + 6 + 7 + 8;
disp(s);
```

Output:

```
36
```

MATLAB uses "…" as the line continuation operator, allowing an expression to continue over multiple lines. The given code is performing an arithmetic operation where multiple numbers are being added together. It starts by adding the numbers 1, 2, and 3. The addition then continues on the next line (…) with the numbers 4 through 8. After computing the sum, which is stored in the variable s, the result is printed to the console using the *disp(s)* statement.

2.1.16 Ex. (16) – Formated output

```
a = 3;
b = 7;
c = 10;
r = ['a = ', num2str(a), ' and b = ', num2str(b)];
t = ' is a number.';
l = [(a + b / c), t];
disp([l, r]);

% or:

a = 3;
b = 7;
c = 10;

% Creates the string with the variables.
r = sprintf('a = %d and b = %d', a, b);
t = ' is a number.\n';

% Formats the calculation and t together.
l = sprintf('%f%s', (a+b/c), t);

% Prints out l followed by r
fprintf('%s%s', l, r);
```

Output:

```
3.7 is a number.
a = 3 and b = 7
```

In this code snippet, several variables are declared and manipulated. First, variables *a*, *b*, and *c* are declared and initialized with numerical values 3, 7, and 10, respectively. Next, a string variable *r* is created and assigned a value that concatenates the string "a = " with the value of *a*, then " and b = " with the value of *b*. This will create a string that describes the values of variables *a* and *b*. The string concatenation in MATLAB is made using square brackets [], and printing to the console, as stated many times, is done using the *disp* function. Additionally, MATLAB uses *num2str* or *int2str* to convert numbers to strings for concatenation. Another string variable *t* is initialized with the string " is a number.". The purpose of this line is to show that text can be added later too. Next, the variable *l* is then created, and it stores the result of an arithmetic operation that adds *a* to the division of *b* by *c*. This result is then concatenated with the string stored in *t*. Next, the *disp* function is called to display the combined value of *l* and *r*. The *sprintf* function in MATLAB is used to format strings, and it can handle the interpretation of the "\n" newline character properly. When this code is executed, it will display the result of the calculation $(a + b/c)$ followed by the string *t* and *r*.

Conditional Branching

3

Decision-making in source code, is a fundamental concept in computer science and algorithm design, enabling systems to perform different computations depending on whether a specified Boolean condition evaluates to true or false [1]. At its core, conditional branching simulates the logical reasoning humans naturally employ in decision-making processes. In MATLAB, conditional branching is performed using *if*, *elseif*, and *else* keywords, much like in other structured computer languages. These keywords allow the programmer to define conditions that control the flow of execution in the program. The *if* statement checks a condition and executes the subsequent block of code only if the condition is true. If the condition is false, MATLAB will skip that block of code and continue executing the rest of the program. The *elseif* and *else* statements can be used after an *if* statement to provide additional conditions and a default block of code if none of the conditions are true. MATLAB also supports the *switch* statement, which is used for conditional branching where one out of many blocks of code is executed based on the value of a variable or expression. Within a *switch* block, case statements define the blocks of code that correspond to specific values, and the otherwise statement defines the block of code that is executed if none of the case conditions match. In terms of performance, MATLAB is designed with vectorization and matrix operations in mind, which are generally more efficient than loop-based approaches that might involve a lot of conditional branching. However, when conditionals are necessary, it is still important to structure them in a way that makes efficient use of the hardware, considering that branching can have an impact on performance due to pipeline stalls in the CPU, as mentioned. Optimal MATLAB code will often use logical indexing or matrix operations to perform operations that might otherwise require conditional branching, thereby avoiding the potential performance pitfalls

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_3

of excessive branching. Nonetheless, for many tasks, particularly those involving algorithmic decision-making, conditional branching remains an essential tool. It is important for MATLAB programmers to understand how to use *if*, *elseif*, *else*, and *switch* constructs effectively while being mindful of their potential impact on computational efficiency.

3.1.1 Ex. (17) – *If then else* – conditional statements (I)

```
a = 4;
b = 7;
if a < b
    disp(a);
else
    disp(b);
end
```

```
Output:

4
```

The code from above defines two variables, a with a value of 4 and b with a value of 7. It then checks if the value of a is less than the value of b using an *if-else* statement. If the condition is true, meaning if a is indeed less than b, it will print the value of a to the console or screen. If the condition is false, it will print the value of b. In this case, since 4 is less than 7, it will print the value of a, which is 4. Note that the end of a control structure block (like an if statement) is marked with the keyword *end*.

3.1.2 Ex. (18) – *If then else* – conditional statements (II)

```
a = 2;
b = 3;
c = 1;

if a < b
    c = 0;
end

disp(c);
```

```
Output:

0
```

This Matlab code initializes three variables: a is assigned the value 2, b is assigned the value 3, and c is assigned the value 1. It then checks if the value of a is less than the value of b. If this condition is true, which it is in this case since 2 is less than 3, the value of c is updated to 0. Next, the value of c is printed. Given the initial values and the condition provided, the output will be 0.

3.1.3 Ex. (19) – *If then else* - conditional statements (III)

```
a = 1;
b = 2;
c = 3;

if a < b
    c = c + 1; % increment c by 1.
else
    c = c - 1; % decrement c by 1.
end

disp(['c=' num2str(c)]);
```

Output:

```
c=4
```

The given Matlab source code from above initializes three variables: a is assigned a value of 1, b is assigned a value of 2, and c is assigned a value of 3. Then, there is an if-else conditional statement that checks if the value of a is less than the value of b. If this condition is true, the value of c is incremented by 1. If the condition is not true (i.e., if a is not less than b), the value of c is decremented by 1. After evaluating this conditional statement, the code prints the value of c with a prefix "c=" to the console. After the *if-else-end* statement, the value of c would be 4 because the condition $a < b$ (1 is less than 2) is true. Note that the concatenation of "c=" with the string representation of c is done using *num2str*(c).

3.1.4 Ex. (20) – *If then elseif else* - conditional statments

```
a = 1;
b = 2;
c = 3;

if a < b
    c = c - 1;
elseif b == c
    c = c + 1;
else
    c = 0;
end

disp(['c=' num2str(c)]);
```

Output:

```
c=2
```

The example initializes three variables: a with a value of 1, b with a value of 2, and c with a value of 3. The code then contains a conditional structure to manipulate the value of c based on certain conditions. If the value of a is less than b, then 1 is subtracted from the current value of c. If that condition is not met but the value of b is equal to the value of c, then 1 is added to the current value of c. If neither of these conditions is satisfied, the value of c is set to 0. After evaluating these conditions, the code prints the string "c=" followed by the current value of c. Note that in MATLAB, parentheses around conditions

in an *if* or *elseif* statement are optional if the condition is a simple expression. However, they are often included for clarity, especially when the expressions get more complex.

3.1.5 Ex. (21) – *Switch* - conditional statments

```
a = 1;
b = 0;

switch a
  case 0
    b = 11;
  case 1
    b = 64;
  case 2
    b = 33;
end

disp(b);
```

Output:

```
64
```

This Matlab code starts by declaring two variables, *a* and *b*, and assigning them the values of 1 and 0 respectively. Then, a switch statement is used to evaluate the value of *a*. If *a* is 0, the value 11 will be assigned to *b*. If *a* is 1, then *b* will be assigned the value 64. If *a* is 2, the value 33 will be assigned to *b*. The break statements ensure that once a match is found in the switch statement, the program exits out of the switch block without checking the remaining cases. Thus, the value of *b* is printed out. Given the initial value of *a* is 1, the printed value of *b* will be 64.

Loops

4

In computational theory and the field of computer programming, loops hold a paramount position [1]. They are fundamental structures that facilitate the repeated execution of a set of instructions, enabling efficient automation and repetitive task handling. From a conceptual viewpoint, a loop is a mechanism by which a process can be reiterated until a specific condition or set of conditions is met. This cyclic execution allows for the efficient handling of tasks that follow a recurrent nature. There are several types of loops, primarily distinguished by their control mechanisms: (a) For-Loop, that is generally used when the number of iterations is known in advance. The loop contains an initializer, a condition, and an iterator. It commences with the initialization, checks the condition, and post-execution, the iterator modifies the loop variable, leading to the next iteration or exit. (b) While-Loop, that is predominantly used when the number of iterations is not predetermined, the while loop checks a condition before every iteration. If the condition evaluates to true, the loop body is executed. (c) Do-While Loop, that is a variant of the while loop, ensures that the loop body is executed at least once before checking the condition, given that the condition is evaluated post the execution of the loop body. Each iteration within a loop is commonly referred to as a "cycle". In every cycle, the computational instructions are reevaluated, often with altered variables, leading to different outcomes in each iteration. It is vital to ensure that loops have a definitive termination point or a condition that will be met, to prevent infinite looping, which can lead to system hang-ups or overconsumption of computational resources. Loops are foundational constructs in computer programming that harness the power of computation by enabling repetitive execution based on conditions. Some basic examples are shown here in order to familiarize the reader with these structures.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_4

4.1.1 Ex. (22) – *While loop*

```
i = 0;

while i < 5
    fprintf('i = %d\n', i);  % Print the value of i.
    i = i + 1;               % Increment i by 1.
end
```

Output:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

The code initializes a variable named i with the value of 0. Then, a while loop starts, which continues executing its body as long as the condition $i < 5$ holds true. Inside the loop, a *fprintf* function is called, which displays the current value of i in the format "i = [value of i]". Once the value is printed in the Command Window, the value of i is incremented by 1 using the i++ statement. As a result, the loop will print the values of i from 0 to 4. Once i reaches 5, the condition $i < 5$ will no longer be true, and the loop will terminate. Note again that the *fprintf* function is used for formatted text output in MATLAB, and it is more flexible than the *disp* function, which can also be used for displaying information.

4.1.2 Ex. (23) – *Do while*

```
i = 0;

while true     % Begin an infinite loop.
    disp(['i = ', num2str(i)]);
    i = i + 1;
    if ~(i < 5)
        break; % Break the loop if false.
    end
end
```

Output:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

This Matlab code initializes a variable i with a value of 0. It then enters a "do-while" loop. Inside the loop, a *disp* function is called to display the current value of i, which is concatenated with the string "i = ". Notice that *num2str*(i) function is used to convert the numeric value of i to a string so it can be concatenated with another string. Next, the value of i is incremented by 1. The loop continues to execute as long as the value of i is less than 5. Once i reaches 5, the loop stops. Thus, the output of this code would be a sequence of printed statements displaying "i = 0", "i = 1", "i = 2", "i = 3", and "i = 4". In other words, the *while true* loop creates an infinite loop, which is only exited when the *if* condition inside the loop fails (i < 5) and break is executed. Note that this MATLAB code mimics the behavior of a *do-while* loop where the loop body is guaranteed to execute

at least once. Note that in MATLAB, the tilde "~" is used as the logical NOT operator. When used in an *if* statement, or any logical expression, it negates the condition.

4.1.2 Ex. (24) – Simple *for loop*

```
for i = 0:4
    disp(i);
end
```

Output:

```
0 1 2 3 4
```

The code above is a *for-loop* that initializes a variable *i* to 0. It then checks if the value of *i* is less than 5. If the condition is true, the code inside the loop is executed. Inside the loop, there is a function call to *disp(i)*, which presumably would display or log the value of *i*. After the execution of the loop body, *i* is incremented by 1. The loop will continue to execute as long as *i* is less than 5. As a result, the numbers 0 through 4 will be passed to the *disp* function one at a time. From a syntax point of view, the *for-loop* in MATLAB starts with the keyword *for* and is followed by the loop variable, which is *i* in this case. Notice that in MATLAB, the loop range is defined by the colon operator ":", which creates a row vector with a default increment of 1. Thus, "0:4" generates a vector [0 1 2 3 4].

4.1.4 Ex. (25) – Reverse by substraction from the upper limit

```
for i = 0:4
    disp(5 - i);
end
```

Output:

```
5 4 3 2 1
```

The given code initializes a loop with a variable *i* set to 0. The loop continues to run as long as the value of *i* is less than 5. With each iteration of the loop, the value of *i* increases by 1 due to the *i*++ increment operation. Inside the loop, there is a function call to *disp()* which takes the expression 5–*i* as its argument. This means that for each iteration of the loop, the function will print a value that starts from 5 (when *i* is 0) and decrements by 1 with each subsequent iteration. Thus, the sequence of numbers printed will be 5, 4, 3, 2, and finally 1.

4.1.5 Ex. (26) – The meaning of steps

```
for i = 10:-1:6
    disp(i)
end
```

Output:

```
10 9 8 7 6
```

The given Matlab code the loop vector is written as "10:−1:6", where 10 is the starting value, −1 is the step indicating decrement (in this case), and 6 is the end value. Thus, in MATLAB, *for-loop* syntax is for *variable* = *start*:*step*:*end*. With each iteration of the loop, the value of *i* is decremented by 1. Inside the loop body, there is a *disp(i)* statement, which would ideally print the current value of *i*.

4.1.6 Ex. (27) –The immutable counter variable

```
for i = 10:-2:6
    i = i - 1;   % Decrement i by one more.
    disp(i);     % Display the value of i.
end
```

Output:

```
9 7 5
```

The given code consists of a *for-loop* that initializes a variable *i* with the value of 10. The loop will continue executing as long as the value of *i* is greater than 5. After each iteration of the loop, the value of *i* is decremented twice (−2). Inside the loop body, there is a call to function *disp*. This function receives a value which is the result of decrementing *i* by 1 ($i = i - 1$;). However, unlike many C-like languages, in MATLAB this decrement of *i* is temporary. In MATLAB, the loop variable in a *for-loop* is controlled by the loop itself. Each iteration of the loop automatically updates the loop variable based on the range specified in the loop declaration. Any changes made to the loop variable inside the loop body are typically overwritten when the loop progresses to the next iteration. Therefore, while one can technically modify the loop variable within the loop body, it is generally not effective because the loop control statement will reset it at the beginning of each iteration. This behavior ensures that the loop progresses predictably through the range specified in the loop declaration. The modification of *i* within the loop body will not affect the sequence of *i* as dictated by the for statement. After each iteration, *i* will be set to the next value in the sequence, regardless of any changes made to *i* in the previous iteration. In order to achieve specific behaviors, one should structure the loop declaration accordingly or use a *while-loop* where one has full control over the loop variable. Thus, for our current example, the values printed by the loop will be 9, 7 and 5, after which the loop will terminate since *i* will be equal to 6.

4.1.7 Ex. (28) – Revers by substraction from the upper limit variable

```
a = 5;

for i = 0:(a-1)
    disp(a - i);
end
```

Output:

```
5 4 3 2 1
```

The provided Matlab code initializes a variable *a* with the value of 5. It then uses a *for-loop* to iterate from 0 up to, but not including, the value of *a* (which is 5). During each iteration of the loop, it calls the function named *disp* with the argument *a–i*. This means that with each iteration, it will print the result of subtracting the current loop index *i* from *a*. Thus, the sequence of numbers that will be printed is 5, 4, 3, 2, and 1.

4.1.8 Ex. (29) – *For loop* sumation

```
a = 0;

for i = 1:5
    a = a + (i + 4 * 3);
end

disp(a);
```

Output:

```
75
```

The above code initializes a variable *a* with a value of 0. Following this initialization, there is a *for-loop* that runs 5 times. MATLAB uses the colon operator ":" to create a range of numbers, which is what the *for-loop* iterates over. In this case, 1:5 generates a vector [1 2 3 4 5]. Within each iteration of the loop, the value of *i* (which starts from 1 and increments by 1 each time) is added to the product of 4 multiplied by 3. The result of this addition is then added to the current value of *a*. Essentially, during each loop iteration, 12 (which is 4 multiplied by 3) is added to the value of *i* and the sum is added to *a*. After the loop completes its 5 iterations, the *disp* function is called to display the final value of *a*. The *disp* function here outputs the final value of *a* to the console.

4.1.9 Ex. (30) – Simple counter sumation

```
a = 0;

for i = 0:10
    a = a + i;
end

disp(a);
```

Output:

```
55
```

The given Matlab code initializes a variable named *a* with a value of 0. Then, there is a *for-loop* that iterates 11 times, starting with the value 0 up to, but not including, the value 11 (MATLAB uses "1:n" to create a range from 1 to *n*. In this case, 0:10 to get the range from 0 to 10). Within each iteration of the loop, the value of *i* (which is the loop counter) is added to the current value of *a*. As a result, *a* accumulates the sum of integers from 0 to 10. After the loop completes its execution, the value of *a* (which will be the sum of the integers from 0 to 10) is printed out.

4.1.10 Ex. (31) – Sum all results of addition of 1 in a $n \times n$ cycle

```
r = 0;
for i = 1:10
    for j = 1:10
        r = r + 1;
    end
end

disp(r);
```

Output:

```
100
```

The Matlab code initializes a variable r with the value 0. It then sets up a nested loop structure: the outer loop runs with the variable i from 1 up to 10, and for each iteration of this outer loop, an inner loop runs with the variable j also from 1 up to 10. During each iteration of the inner loop, the value of r is incremented by 1. As a result, the inner loop runs a total of 100 times (10 times for each of the 10 iterations of the outer loop). Hence, by the end of the nested loops, the value of r will be 100. After the loops are finished, the code prints the value of r, which will display the number 100. Note: in MATLAB, the loop indices typically start at 1, not 0. This is because MATLAB is one-indexed, unlike many other programming languages which are zero-indexed.

4.1.11 Ex. (32) – Sum all results of addition of 3 in a $n \times n$ cycle

```
r = 0;

for i = 0:3
    for j = 0:3
        r = r + 3;
    end
end

disp(r);
```

Output:

```
48
```

The given Matlab code initializes a variable r with the value of 0. It then uses a nested for-loop, where the outer loop runs 4 times (with the loop variable i ranging from 0 to 3) and the inner loop also runs 4 times (with the loop variable j ranging from 0 to 3). For each iteration of the inner loop, the value of r is incremented by 3. Since there are a total of 16 iterations (4 from the outer loop multiplied by 4 from the inner loop), r is incremented by 3 a total of 16 times. As a result, by the end of these nested loops, the value of r becomes 48. Next, the code uses the *disp* function to display the value of r, which would output the number 48.

4.1.12 Ex. (33) – Sum all results of the multiplication between i and j

```
r = 0;
for i = 0:9
    for j = 0:9
        r = r + j * i;
    end
end

disp(r);
```

Output:

```
2025
```

This code initializes a variable r with a value of 0. It then has a nested loop where the outer loop uses a variable i which runs from 0 to 9, and the inner loop uses a variable j which also runs from 0 to 9. For each combination of i and j, the product of i and j is calculated and added to the value of r. After both loops have completed their iterations, the accumulated total in r is then printed out. Thus, the code computes the sum of products of all possible combinations of i and j within the range specified.

4.1.13 Ex. (34) – Nested for loops and summation of counter variables

```
a = 0;
m = 3;
n = 5;

for j=1:m
    for i=1:n
        a = a + (i + j * 3);
    end
end

disp(a);
```

Output:

```
135
```

The above Matlab code initializes three variables: a with a value of 0, m with a value of 3, and n with a value of 5. It then contains a nested loop where the outer loop runs as long as j is less than or equal to m (3 times in this case), and for each iteration of this outer loop, an inner loop runs as long as i is less than or equal to n (5 times). Within the inner loop, the value of a is incremented by the sum of i and three times j. After both loops have finished executing, the accumulated value of a is printed out. The purpose of this code is to accumulate a sum based on the conditions and limits set by the variables m and n.

4.1.14 Ex. (35) – Nested for loops and summation bases on the inner counter

```
a = 0;

for j = 1:3
    for i = 1:5
        a = a + (i + 1 * 3);
    end
end

disp(a);
```

Output:

```
90
```

The given Matlab code initializes a variable a with the value of 0. It then uses a nested *for-loop* structure to iterate and modify the value of a. The outer loop, controlled by the variable j, runs three times, as j goes from 1 through 3 inclusive. Inside this outer loop, there is an inner loop controlled by the variable i, which runs five times for each iteration of the outer loop, since i goes from 1 through 5 inclusive. For every iteration of the inner loop, the value of a is incremented by the result of the expression $(i + 1 * 3)$. This expression adds i to the product of 1 and 3. Given the rules of arithmetic operation precedence, the multiplication is performed before the addition, thus, the expression is equivalent to $(i + 3)$. Given the loop structures, this means the operation $(i + 3)$ is executed a total of 15 times (3 times for the outer loop multiplied by 5 times for the inner loop). Next, after both loops have completed their iterations, the value of a is printed out.

4.1.15 Ex. (36) – Nested for loops & summation based on counters and upper limits (I)

```
a = 0;
m = 3;
n = 5;

for j = 1:m
    for i = 1:n
        a = a + (i + j * m);
    end
end

disp(a);
```

Output:

```
135
```

The above Matlab code initializes three variables: a with a value of 0, m with a value of 3, and n with a value of 5. The code then sets up a nested loop structure with an outer loop running from 1 through the value of m (inclusive) and an inner loop running from 1 through the value of n (inclusive). Inside the innermost part of this nested loop, the value of a is incremented by the sum of the current value of i and the product of the current value of j and m. After both loops have fully executed, the final value of a is printed out. Essentially, this code performs a computation based on the two loop counters and accumulates the result in the variable a.

4.1.16 Ex. (37) – Nested for loops & summation based on counters and upper limits (II)

```
a = 0;
m = 4;

for j = 1:m
    for i = 1:j
        a = a + (i + j * m);
    end
end

disp(a);
```

Output:

```
140
```

The given Matlab code begins by initializing two variables: a is set to 0, and m is set to 4. Following this, there is a nested loop structure. The outer loop runs with the variable j, starting from 1 up to and including the value of m, which is 4. Inside this outer loop, there is an inner loop that runs with the variable i, starting from 1 and going up to the current value of j from the outer loop. Within the inner loop, the code calculates a value by adding i and the product of j and m. This calculated value is then added to the current value of a, effectively updating a with each iteration of the inner loop. Once both loops have completed their iterations, the final accumulated value of a is printed out using the *disp()* function.

4.1.17 Ex. (38) – Nested for loops & summation based on counters and upper limits (III)

```
a = 0;
m = 5;
n = 7;

for j=1:m
    for i=j:n
        a = a + (i + j * m);
    end
end
disp(a);
```

Output:

```
445
```

This Matlab code initializes three variables a, m, and n with the values 0, 5, and 7, respectively. The code then has a nested loop where the outer loop runs with the variable j iterating from 1 through the value of m (which is 5). For each iteration of the outer loop, the inner loop runs with the variable i starting from the current value of j up to the value of n (which is 7). Within the inner loop, the value of a is updated by adding the sum of i and the product of j and m. After both loops are completed, the value of a is printed. Essentially, this code is calculating a summation based on the provided formula and values of m and n.

4.1.18 Ex. (39) – Show *i* and *j* coordinates at each strp

```
for i = 0:1
    for j = 0:2
        fprintf('i = %d, j = %d\n', i, j);
    end
end
```

Output:

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
```

The provided Matlab code consists of a nested loop. The outer loop, controlled by the variable *i*, runs for two iterations, with *i* taking values 0 and 1. Inside each iteration of this outer loop, there is an inner loop, controlled by the variable *j*, which runs for three iterations, making *j* take on the values 0, 1, and 2. During each iteration of the inner loop, the *fprintf* function is called to display the current values of both *i* and *j*. The "%d" is a placeholder for an integer, and the variables *i* and *j* are passed after the string. MATLAB uses "\n" to represent a newline character in the format string for *fprintf*. Thus, the message "i = [value of i], j = [value of j]" will be printed a total of six times, reflecting every combination of *i* and *j* within the specified ranges. For example, the first few messages will be "i = 0, j = 0", "i = 0, j = 1", "i = 0, j = 2", and so on.

4.1.19 Ex. (40) – One *for loop* that simulates two for loops

```
i = 0;
j = 0;
n1 = 2;
n2 = 3;
q = n1 * n2;

for v = 0:(q-1)
    j = mod(v, n2);
    if (j == 0) && (v ~= 0) && (i < n1) && (v ~= q)
        i = i + 1;
    end
    fprintf('i = %d, j = %d\n', i, j);
end
```

Output:

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
```

The Matlab code snippet initializes three variables *i*, *j*, and *v* to 0 and two other variables *n1* and *n2* to 2 and 3, respectively. It also calculates the product of *n1* and *n2*, storing the result in a variable named *q*. A *for-loop* then iterates *v* from 0 up to, but not including, the value of *q*. Inside the loop, the value of *j* is calculated as the remainder of the division of *v* by *n2*. The *mod*() function is used in MATLAB to find the modulus, which is equivalent to the "%" operator in many other computer languages. A conditional *if* statement checks if *j* is 0, *v* is not 0, *i* is less than *n1*, and *v* is not equal to *q*. If all

these conditions are met, i is incremented by 1. After each iteration of the loop, the *fprintf* function is called to output the current values of i and j. The purpose of the code is to explore the behavior of the variables i and j as v goes from 0 to $q - 1$, under certain conditions specified in the *if* statement. It showcases how the value of i can be incremented based on the other variables and conditions within the loop.

Arrays

5

Data structures are pivotal constructs that enable the systematic organization and management of data [1]. One of the quintessential and most universally utilized data structures is the array. An array can be aptly described as a collection of items stored at contiguous memory locations. Its salient feature is the direct access it offers to any indexed element, granting it significant computational advantages in specific scenarios. Characteristics of arrays encompass homogeneity, implying that all elements within an array are of the same data type, ensuring uniform memory footprint. Arrays are static in nature, unlike dynamic data structures, such as linked lists, and rely on indexing. Each element in the array is associated with a unique index, facilitating swift access to any element with known index. Arrays find applications in various domains within computer science. Also, they are fundamental for sorting algorithms like *QuickSort* and *MergeSort*, often employed for their direct access capabilities. Arrays also serve as foundational elements for complex data structures such as *heaps*, *hash tables*, and *dynamically resizable arrays*. Advantages of arrays include their speed in retrieval operations and efficient memory allocation due to contiguous storage. However, they have limitations, notably their fixed size and the relatively costly nature of insertion and deletion operations, particularly for elements in the middle of the array. Subsequent examples will demonstrate typical practices and techniques associated with arrays in MATLAB, highlighting ways to access and manipulate the information within these structured data types.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_5

5.1.1 Ex. (41) – Array addition and concatenation

```
% a(i) vector
% a(i,j) = matrix
% a(i,j,x) = tensor
% a(i,j,x,y,.......

% Define vectors.
a = [2, 5, 7];
b = [6, 8, 9];

% Concatenation.
c = [a, b];
disp(c);

% Addition.
disp(a+b);
```

Output:

```
2,5,7,6,8,9
8,13,16
```

The provided code starts with a series of comments that serve to explain the structure of nested arrays, often referred to as vectors, matrices, or multidimensional arrays. These comments lay out a conceptual hierarchy, illustrating how data can be organized in Matlab arrays. The a variable is declared as an array with three elements: [2, 5, 7]. In the comments, it is described as a vector, implying a one-dimensional array. The comment $a(i,j) = matrix$ suggests the potential for two levels of nesting within this array, implying a two-dimensional structure resembling a matrix. The comment $a(i,j,x)$ indicates a tridimensional mathematical structure that extends this hierarchy further, indicating that within this matrix-like structure, there is yet another level of nesting. The comment $a(i,j,x,y,$ …) suggests the possibility of even deeper levels of nesting, forming higher-dimensional structures. Following these comments, two arrays, a and b, are declared with numeric elements. The $c = [a, b]$; line allows the concatenation of the two arrays a and b. Next, the output is shown by using the $disp(c)$ function. However, concatenation works with arrays of different sizes, but in the case of addition, the sizes of the arrays must be the same. Thus, operations such as addition on two matrices (i.e. $a + b$) must have compatible dimensions.

5.1.2 Ex. (42) – Extracting individual values from the elements of an array

```
a = [2, 5, 7]; % Creating array a.
b = [6, 8];    % Creating array b.

% In MATLAB, array indices start from 1.
c = a(2) + b(1);

disp(c);
```

Output:

```
11
```

The provided code snippet demonstrates a series of operations involving arrays and variable assignments. Initially, three variables are declared: *a*, *b*, and *c*. The *a* variable is assigned an array [2, 5, 7], while the *b* variable is assigned another array [6, 8]. These arrays can store multiple values. The key operation occurs when the *c* variable is assigned a value. It calculates this value by adding together two specific elements from the arrays *a* and *b*. In short, it takes the second element from *a* (which is 5) and the first element from *b* (which is 6) and adds them together. The result of this addition, 11, is stored in the variable *c*. Next, the value of the *c* variable is shown in the output using *disp*(*c*). Therefore, this Matlab source code snippet involves array operations and variable assignments, culminating in the addition of specific elements from two arrays and the printing of the result.

5.1.3 Ex. (43) – Adding elements

```
% Initialize a cell array.
A = cell(1,3);

% Assign strings to each cell in the array.
A{1} = 'a';
A{2} = 'b';
A{3} = 'c';

% Concatenate and print the strings.
disp([A{1}, A{2}, A{3}]);
```

Output:

```
abc
```

The example initializes a variable *A* as an empty array. Namely, $A = cell(1,3)$, that creates a 1×3 cell array. Cell arrays are used in MATLAB to store arrays of strings or arrays of mixed types. Next, it proceeds to assign values to its elements. In this case, the values assigned are strings: “a” to $A\{1\}$, “b” to $A\{2\}$, and “c” to $A\{3\}$. It then uses the *disp* function to output the concatenation of these three elements. Note that in MATLAB, the square brackets [] are used for concatenation, which means it combines the three strings “a,” “b,” and “c” into a single string. Therefore, the output of this code will be “abc” when it is executed.

5.1.4 Ex. (44) – Using array literals of different data type

```
% Initializing cell arrays for
% strings and numeric values.

A = {};
B = [];

% Assigning values to the arrays.
A = {'a', 'b', 'c'};
B = [1, 2, 3];

% Concatenated string from A.
disp([A{1}, A{2}, A{3}]);

% Sum of elements in B.
disp(sum(B));
```

Output:

```
abc
6
```

In this code snippet, we are working with two arrays, *A* and *B*. However, these empty declarations are later overwritten with new values. The first array, *A*, is populated with three string elements: “a”, “b”, and “c”. Each of these elements is enclosed in double quotes and separated by commas. The second array, *B*, is filled with three numeric elements: 1, 2, and 3. These numeric values are not enclosed in quotes because they are treated as integers. After initializing these arrays, the code proceeds to print out the concatenation of elements within each array. For array *A*, the concatenation of its elements “a”, “b”, and “c” results in the string “abc”. This string is printed to the console using the *disp* function. Similarly, for array *B*, the example makes an addition of its elements 1, 2, and 3, that results in the numeric value 6 ($1 + 2 + 3$). This numeric value is also printed to the console using the *disp* function.

5.1.5 Ex. (45) – Accessing array elements

```
A = ["a", "b", "c"];

x = A(2);
y = A(3);

disp([x, y]);
```

Output:

```
bc
```

This Matlab code begins by defining an array called “A,” which contains three elements: “a,” “b,” and “c.” Next, the code declares two variables, x and y. The variable x is assigned the value of the second element in the array *A*, which is “b”. Similarly, the variable y is assigned the value of the third element in the array *A*, which is “c.” The code prints the result of concatenating the values of x and y using the square parentheses “[]”. Thus, “[x,y]” would result in “bc” because x holds “b” and y holds “c.”

5.1.6 Ex. (46) – Changing values in array elememts - swap values or replace

```
A = {'a', 'b', 'c'};

x = A{2};

A{1} = 'd';
A{2} = A{3};
A{3} = x;

disp([A{1}, A{2}, A{3}]);
```

Output:

```
dcb
```

The above code begins by defining an array called *A* containing three string elements: "a," "b," and "c." Next, it initializes a variable named *x* and assigns it the value at the index 2 of array *A*, which is "b." Following this, the code proceeds to modify the elements within the array *A*. It assigns the string "d" to the first element at index 0, replaces the second element at index 1 with the value from the third element at index 2, and finally, sets the third element at index 2 to the value of *x*, which is "b." In the last line of code, the *disp* function is used to print the concatenation of the elements at indexes 0, 1, and 2 of array *A*. The result of this concatenation would be "dbc" based on the modifications made to the array earlier in the code.

5.1.7 Ex. (47) – Extracting individual values from the elements of an array

```
a = [2, 5, 7];
b = [6, 8];

% Decrementing the
% second element of a.
a(2) = a(2) - 1;

% Decrementing the
% first element of b.
b(1) = b(1) - 1;

% Sum of the
% updated elements.
c = a(2) + b(1);

disp(c);
```

Output:

```
9
```

This example begins by declaring two arrays, *a* and *b*, containing the elements [2, 5, 7] and [6, 8], respectively. The code proceeds by decrementing the second element (index 2) of array *a* and the first element (index 1) of array *b*. This means that after these operations, the *a* array will become [2, 4, 7], and the *b* array will become [5, 8]. Next, a new variable *c* is declared and assigned the value of the sum of the updated second element of array *a* (which is now 4) and the updated first element of array *b* (which is now 5). Therefore,

c will be assigned the value 9. In summary, this code modifies two arrays, calculates the sum of specific elements from those arrays, and then prints the result to the console.

5.1.8 Ex. (48) – Array length

```
a = [5, 6, 8];
b = length(a); % Length of the array.

disp(b); % Displaying the length.
```

Output:

```
3
```

Here, a variable *a* is declared and initialized as an array containing three elements: 5, 6, and 8. This array is created using square brackets [] and the elements are separated by commas. Next, another variable *b* is declared and assigned the value of *length*(*a*). Here, *length*(*a*) is a property of the array *a*, which represents the number of elements in the array. In this case, since there are three elements in the array *a*, *b* will be assigned the value 3. Thus, the value of *b* is printed in the output. In short, this code snippet creates an array *a*, determines its length, and prints that length to the console.

5.1.9 Ex. (49) – Accessing the values from the components of an array

```
A = [1, 2, 3];

if A(1) < A(2)
    A(3) = A(3) + 1;
end

disp(['A[3]=', num2str(A(3))]);
```

Output:

```
A[3]=4
```

This code snippet begins by declaring a variable named *A* and assigning it an array containing three elements: 1, 2, and 3. This array is represented as [1, 2, 3]. Next, there is an *if* statement that checks a condition. It evaluates whether the value at the first index of array *A*, which is $A(1)$, is less than the value at the second index, $A(2)$. In this case, $A(1)$ contains 1, and $A(2)$ contains 2, which is indeed true, as 1 is less than 2. When the condition in the *if* statement is true, the code block inside the *if* statment is executed. In this case, it increments the value at the third index of array *A*, which is $A(3)$. Thus, $A(3) = A(3) + 1$; adds 1 to the existing value in $A(3)$, resulting in $A(3)$ being updated to 4. Next, it displays a message that includes the updated value of $A(3)$. The message is constructed by concatenating the string "A[3] = " with the value of $A(3)$, which is 4. Thus, when this code is executed, it will print "A[3] = 4" to the console.

5.1.10 Ex. (50) – Traverse a 1D array using a *while loop*

```
A = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};

i = 1;  % MATLAB indexing starts at 1.
t = ''; % Initialize an empty string.

while i <= length(A)
    % Concatenate strings:
    t = [t, '\n i[', num2str(i), ']=', A{i}];
    i = i + 1;
end

fprintf(t);
```

Output:

```
i[1]=a
i[2]=b
i[3]=c
i[4]=d
i[5]=e
i[6]=f
i[7]=g
```

This Matlab code begins by initializing an array called *A*, which contains the elements "a," "b," "c," "d," "e," "f," and "g." In MATLAB, curly braces {} are used to create cell arrays which can hold strings. This is because MATLAB has a different array type for strings and numeric data. Following this, the code sets up two variables, *i* and *t*, where *t* is initially set to empty. Next, *i* is initialized to 1, and *t* is set as an empty string. The variable *i* is set to 1, because, unlike other computer languages, MATLAB uses 1-based indexing. The code then enters a *while loop* that runs as long as the value of *i* is less than the length of the array *A*. Inside the loop, there is an assignment to the variable *t*. The assignment appends a newline character followed by a string constructed from the value of *i* and the corresponding element in the array *A*. This creates a string that looks like "\n i[1] = a", "\n i[2] = b", and so on. In MATLAB, strings are concatenated using the [] operator. Also, the *num2str* function is used to convert a number to a string. After the assignment, *i* is incremented by 1. Once the loop has iterated through all elements in the array *A*, the code shows the value of *t* by using the *fprintf* function, which is able to correctly interpret the "\n" as new line. In essence, this code is designed to loop through the elements of the array *A*, building a string *t* that contains information about the index *i* and the corresponding element in the array.

5.1.11 Ex. (51) –Traverse a 1D array using a *for loop* (I)

```
A = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};

t = '';

for i = 1:length(A)
    t = [t, '\n i[', num2str(i), ']=', A{i}];
end

fprintf(t);
```

Output:

```
i[1]=a
i[2]=b
i[3]=c
i[4]=d
```

```
i[5]=e
i[6]=f
i[7]=g
```

This Matlab code initializes an array named *A* containing seven elements: “a,” “b,” “c,” “d,” “e,” “f,” and “g.” It then declares two variables, *i* and *t*, and assigns them initial values of 1 and an empty string, respectively. The code enters a *for-loop*. Inside the loop, there is a string concatenation operation. It appends a newline character followed by a string that includes the current value of *i* and the corresponding element from the array *A*. This information is appended to the *t* string. The *i* variable is incremented with each iteration. The loop continues to execute as long as the value of *i* is less than the length of the array *A*. Next, the code prints the value of the *t* variable. Overall, this code demonstrates the *for-loop* on arrays. Namely, it builds a string *t* by concatenating information about each element in the array *A* along with its index and then displays the resulting string.

5.1.12 Ex. (52) – Traverse two 1D arrays using a *for loop* (II)

```
A = {'a', 'b', 'c', 'd', 'e'};
B = {'a', 'b', 'c', 'd', 'e'};

t = '';

for i = 1:length(A)
    t = [t, '\n A[', num2str(i), ']=', A{i}, B{i}];
end

fprintf(t);
```

Output:

```
A[1]=aa
A[2]=bb
A[3]=cc
A[4]=dd
A[5]=ee
```

This code begins by defining two constant arrays called *A* and *B*, each containing five elements, namely “a,” “b,” “c,” “d,” and “e.” Next, there is a declaration of an empty string variable called *t*. The code then enters a *for-loop* that initializes a loop counter *i* to 1. It sets a condition that checks if *i* is less than the length of the array *A*, which is 5 in this case. As long as this condition is true, the loop continues to execute. Inside the loop, there is a statement that appends a newline character (“\n”) to the *t* string, followed by the text “A[” concatenated with the current value of *i*, followed by “] = ” and the value at the corresponding index in the array *A* and *B*. In other words, during each iteration of the loop, it appends a line to the *t* string that displays the index and value of each element in the *A* and *B* arrays. Next, after the loop finishes, the code prints the *t* string using the *fprintf* function, that interprets the “\n” as a new line.

5.1.13 Ex. (53) – Print different data type values from array using a *for loop* (I)

```
a = {'x', 'y', 2};  % mixed cell array.

for i = 1:length(a)
    disp(a{i});
end
```

Output:

```
x
y
2
```

The provided Matlab code snippet demonstrates the creation of a variable *a* and the utilization of a *for- loop* to iterate through its elements. An array containing three elements is assigned to variable *a*, namely: the strings "x" and "y", and the number 2. Subsequently, a *for- loop* is used to traverse the elements of this array. Inside the loop, the *disp*(*a*{*b*}) statement displays each element of the array. Thus, the code primary purpose is to showcase the *for-loop* for iterating over a mixed data type array.

5.1.14 Ex. (54) – Print all integers from array using a *for loop* (II)

```
a = [5, 6, 8];

for j = 1:length(a)
    disp(a(j));
end
```

```
Output:

5
6
8
```

This example initializes an array called *a* containing three numeric values: 5, 6, and 8. It then proceeds to iterate through the elements of this array using a *for-loop*. The *for-loop* is configured to start with an index variable *j* set to 1, which corresponds to the first element in the array. It continues looping as long as *j* is less than or equal to the length of the array (i.e., *length*(*a*)). This condition ensures that the loop iterates over all the elements in the array. Within the loop, there is a *disp* statement. This statement outputs the value at the current index *j* in the array *a*. The loop will run for each element in the array, starting with 5, then 6, and finally 8. Thus, this code snippet is a basic example of how to iterate through an array in Matlab using a *for-loop* and print the value of each element to the console. When executed, it will display the values 5, 6, and 8 in the console, each on a separate line.

5.1.15 Ex. (55) – Sum all values from array

```
a = [5, 6, 8];
b = 0;

for j = 1:length(a)
    b = b + a(j);
end

disp(b);
```

```
Output:

19
```

This code snippet begins by defining two variables. The first variable, *a*, is an array containing three numerical values: 5, 6, and 8. The second variable, *b*, is initialized with the value 0. The code then enters a *for-loop*, which is a control structure used for iterating through the elements of an array. In this loop, a variable *j* is initialized to 1, and the loop continues as long as *j* is less than or equal to the length of array *a*. Thus, the loop iterates

through each element of the *a* array. Inside the loop, there is an assignment statement that increments the *b* variable. Namely, it adds the current value of the element of the *a* array, which is indexed by *j*, to the *b* variable. This effectively accumulates in the *b* variable, the sum of all the elements in array *a*. Next, the code calls function *disp* with the argument *b*. Thus, the code calculates the sum of the elements in the *a* array and prints the result to the console.

5.1.16 Ex. (56) – Multiplication involving a scalar and a 1D array

```
a = [5, 6, 8];

for j = 1:length(a)
    a(j) = 2 * a(j); % Doubling each element of a.
end

disp(a);
```

Output:

```
10,12,16
```

This MATLAB snippet initializes one variable, namely *a*, and then performs a loop operation over *a*. The variable *a* is an array containing the elements 5, 6, and 8. Next, a *for-loop* iterates through the elements of array *a*. The loop starts with an index variable *j* set to 1 (MATLAB arrays are 1-based) and continues as long as *j* is less than or equal to the length of array *a*. In each iteration of the loop, the value at the *j*-th index of array *a* is doubled (multiplied by 2) and then assigned back to the same position in the array. After the loop completes, the values from *a* are shown in the console by using the *disp*(*a*) function.

5.1.17 Ex. (57) – Insert values into an array

```
% Pre-allocating an array with 11 elements.
a = zeros(1, 11);

for j = 1:11
    a(j) = j - 1;
end

disp(a);
```

Output:

```
0,1,2,3,4,5,6,7,8,9,10
```

The provided Matlab code initializes an empty array called *a* by pre-allocating an array with 11 elements (*zeros*(1, 11)). This array will be used to store a series of values. Next, there is a *for-loop* that iterates from 1 to 11, with the variable *j* starting at 1. The counter *j* of course increments by 1 in each iteration. Within the loop, the code assigns the value of *j* to the corresponding index in the array *a*. This means that during each iteration of the loop, the value of *j* minus 1 is added to the array *a* (to hold values from 0 to 10). Next, the contents of the array *a* is printed to the output.

5.1.18 Ex. (58) – Insert ascending and descending integer values into arrays

```
a = zeros(1, 11);
b = zeros(1, 11);

for j = 1:11
    a(j) = j - 1;
    b(j) = 10 - (j - 1);
end

disp(['a = ', mat2str(a)]);
disp(['b = ', mat2str(b)]);
```

Output:

```
a = 0,1,2,3,4,5,6,7,8,9,10
b = 10,9,8,7,6,5,4,3,2,1,0
```

This code initializes two arrays, *a* and *b*, as empty arrays. Arrays *a* and *b* are preallocated with zeros for efficiency. Their size is 11 to accommodate elements from 0 to 10. The code then enters a *for-loop* that iterates from 1 to 11. During each iteration of the loop, it assigns values to the *a* and *b* arrays based on the loop index *j*. Specifically, within the loop, the code sets $a(j)$ to the current value of *j*, which corresponds to the numbers from 0 to 10. Simultaneously, it sets $b(j)$ to the value of $10-j$, effectively counting down from 10 to 0. These assignments continue for each iteration of the loop. Once the loop completes, it prints the contents of arrays *a* and *b*.

5.1.19 Ex. (59) – Add forward and reverse values and subtract max

```
a = zeros(1, 11);
b = zeros(1, 11);
c = zeros(1, 11);

for j = 1:11
    a(j) = j - 1;
    b(j) = 10 - (j - 1);
    c(j) = a(j) + b(j) - 10;
end

disp(['a = ', mat2str(a)]);
disp(['b = ', mat2str(b)]);
disp(['c = ', mat2str(c)]);
```

Output:

```
a = 0,1,2,3,4,5,6,7,8,9,10
b = 10,9,8,7,6,5,4,3,2,1,0
c = 0,0,0,0,0,0,0,0,0,0,0
```

This code initializes three arrays, *a*, *b*, and *c*, and then populates them using a *for-loop*. In the beginning, three empty arrays, *a*, *b*, and *c*, are declared to store integer values. The *for-loop* runs from *j* equal to 1 to 11. During each iteration of the loop, the value of *j* is used as an index to populate the arrays *a* and *b*. Specifically, $a(j)$ is assigned the current value of *j*-1, and $b(j)$ is assigned 10-*j*-1. Subsequently, the array *c* is populated based on the values of *a* and *b*. For each index *j*-1, $c(j)$ is calculated as the sum of $a(j)$, $b(j)$, and -10. Thus, the code prints the values of arrays *a*, *b*, and *c*, displaying their contents as strings along with their respective names.

5.1.20 Ex. (60) – Pointless equilibrium

```
a = [];
l = 10;

for j = 1:l+1
    a(j) = j + (l - j));
end

disp(['a = ', mat2str(a)]);
```

Output:

```
a = 10,10,10,10,10,10,10,10,10,10,10
```

In this code snippet, a program initializes an empty array called a and a variable l with the value 10. The purpose of this code is to populate the array a with values based on the iteration variable j using a *for-loop*. The *for-loop* runs from j equals 1 to $l + 1$. During each iteration of the loop, an expression is evaluated and assigned to the j index of the array a. The expression being assigned consists of two parts: (1) Variable j: This represents the current value of the iteration variable j, and (2) is $(l-j)$ that represents the result of subtracting j from l. These two parts are added together, and the result is stored in the array a at the index j. Essentially, it calculates the sum of j and $(l-j)$ for each j from 1 to l-1 storing these values in consecutive elements of the array. Thus, after the loop completes execution, the array a will contain values where each element at index j holds the sum of j and $(l-j)$. The resulting array will be $a = [10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10]$. In this case, all elements of the array a are assigned the value 10 because each iteration calculates $j + (l-j)$, and since l is fixed at 10, the sum remains constant across all iterations, that is, j and $(l-j)$ are complementary.

5.1.21 Ex. (61) – Max value from array

```
a = [2, 3, 4, 5, 9, 8, 3];
l = length(a)

maxV = 0;

for k = 1:l
    if a(k) > maxV
        maxV = a(k);
    end
end

disp(maxV);
```

Output:

```
9
```

This example initializes an array a with a series of numerical values [2, 3, 4, 5, 9, 8, 3]. It then calculates the length of this array and stores it in a variable l. Additionally, it initializes a variable *maxV* with an initial value of 0. Next, there is a *for-loop* that iterates over the elements of the array a. The loop is controlled by a variable k, which starts at

1 and continues until it reaches the value of *l*, which is the length of the array. Within each iteration of the loop, there is an if statement that checks if the current element *a*(*k*) is greater than the current maximum value *maxV*. If *a*(*k*) is indeed greater, it updates the *max*V variable with the value of *a*(*k*), effectively keeping track of the maximum value encountered in the array. Next, after the loop has completed, it prints out the maximum value *maxV* to the console. In essence, this code finds and prints the maximum value present in the array *a*. Note that in MATLAB the *max* keyword is reserved for the *max* built-in function. Thus, the vaiable is called *maxV* instead of just *max*, in order to avoid the conflict with the *max* function.

5.1.22 Ex. (62) – Min value from array

```
a = [3, 3, 4, 2, 9, 8, 3];
l = length(a);

min = a(1);

for k = 1:l
    if a(k) < min
        min = a(k);
    end
end

disp(min);
```

Output:

```
2
```

In the above code, there is an array *a* initialized with values, namely [3, 3, 4, 2, 9, 8, 3]. The variable *l* is assigned the value of the length of the array *a*, which helps determine the range for looping through the array elements. Next, there is the declaration of the variable *min*, which is initially assigned the value of the first element of the array *a*, in this case, 3. This variable will be used to store the minimum value found in the array. The code then enters a *for-loop* with the variable *k* starting from 1 and iterating until *k* is less than or equal to *l*, which means it will go through each element in the array. Inside the loop, there is an *if* statement that checks if the current element *a*[*k*] is less than the current minimum value *min*. If *a*(*k*) is indeed smaller than *min*, the value of *min* is updated to *a*(*k*), effectively finding the minimum value in the array. Next, outside the loop, the code prints the minimum value found in the array using the *disp* statement. This code in complementary to the previous one and calculates and prints the minimum value from the array *a* by iterating through its elements and updating the *min* variable whenever a smaller value is encountered. Note that the *min* keyword is not reserved in MATLAB. Therefore, the vaiable is called *min*.

5.1.23 Ex. (63) – Max value above two arrays of the same size

```
a = [2, 3, 4, 5, 9, 8, 3];
b = [1, 2, 3, 4, 5, 6, 7];
l = length(a);

maxVal = 0;
maxA = 0;
maxB = 0;

for k = 1:l
    if a(k) > maxA
        maxA = a(k);
    end
    if b(k) > maxB
        maxB = b(k);
    end
    if maxA > maxVal
        maxVal = maxA;
    end
    if maxB > maxVal
        maxVal = maxB;
    end
end

disp(maxVal);
```

```
Output:

9
```

The above code starts by defining two arrays, *a* and *b,* each containing a sequence of numbers. Then, it calculates the length of the array *a* and stores it in the variable *l*. Next, it initializes three variables: *maxVal*, *maxA*, and *maxB*, all initially set to 0. These variables will be used to keep track of the maximum values in the arrays. The code enters a *for-loop*, where it iterates through the arrays *a* and *b* simultaneously using the loop variable *k*. It begins at index 1 and goes up to *l*. Within the loop, it checks if the value at index *k* in array *a* (*a*(*k*)) is greater than the current maximum value in *maxA*, and if so, updates *maxA* to this new maximum value. Similarly, it checks if the value at index *k* in array *b* (*b*(*k*)) is greater than the current maximum value in *maxB* and updates *maxB* accordingly. After updating *maxA* and *maxB*, the code also checks if either *maxA* or *maxB* is greater than the current maximum value in *maxVal*. If either of them is greater, it updates *maxVal* to the larger of the two. And last, the code prints the value stored in the *maxVal* variable, which represents the maximum value among both arrays *a* and *b*. Thus, this code finds and prints the maximum value from the combined elements of arrays *a* and *b*.

5.1.24 Ex. (64) – Max value above two arrays of different sizes

```
a = [2, 3, 4, 5, 9, 8, 3];
b = [14, 2, 3, 41, 5, 6, 77];

l = zeros(1,2);
l(1) = length(a);
l(2) = length(b);

% Determine the larger length.
r = l(1);
if l(1) < l(2)
    r = l(2);
end

max_val = 0;

for k = 1:r
    if k <= l(1) && max_val < a(k)
        max_val = a(k);
    end
    if k <= l(2) && max_val < b(k)
        max_val = b(k);
    end
end

disp(max_val);
```

Output:

```
77
```

This code begins by defining two arrays, *a* and *b*, each containing a series of numeric values. An empty array *l* is also initialized for further use. Next, the code assigns the length of arrays *a* and *b* to the elements of the array *l*. Specifically, *l*(0) stores the length of array *a*, and *l*(1) stores the length of array *b*. The code then compares the lengths of arrays *a* and *b* to determine the maximum length and stores it in the variable *r*. If the length of *b* is greater than the length of *a*, *r* is assigned the value of the length of *b*; otherwise, it retains the value of the length of *a*. Subsequently, the variable *max_val* is initialized to 0. The code enters a *for-loop* that iterates from 0 to *r* (inclusive). Inside the loop, it checks if the current index *k* is within the bounds of the lengths of arrays *a* and *b*. If so, it compares the value at index *k* in the respective array with the current maximum value stored in *max_val*. If the value at index *k* is greater than the current maximum, *max_val* is updated to hold that value. Next, the example code prints the maximum value (*max_val*) to the output. In essence, this code finds and prints the maximum value among the elements of arrays *a* and *b*, considering both arrays up to the length of the longer one.

5.1.25 Ex. (65) – Which is bigger between n and $n + 1$?

```
a = [2, 3, 4, 5, 9, 8, 3, 8, 3];
l = length(a) - 1;

t = '';

for k = 1:l
    if a(k) > a(k+1)
        t = [t '>'];
    else
        t = [t '<'];
    end
end

disp(t);
```

```
Output:

<<<<>><><
```

The given source code begins by initializing an array a with a set of numeric values: [2, 3, 4, 5, 9, 8, 3, 8, 3]. Next, it calculates the length of the array minus one and stores it in a variable l. In this case, l becomes 8 since there are nine elements in the array. The code then declares an empty string t, which will be used to accumulate symbols as the code iterates through the array. A *for-loop* is set up to iterate through the elements of the array a. The loop variable k starts at 0 and continues until it reaches the value of l. During each iteration, the code checks if the element at index k in the array a is greater than the element at the next index, $k + 1$. If this condition is true, it appends the " > " symbol to the string t, indicating that the current element is greater than the next one. If the condition is false, it appends the " < " symbol to t, indicating that the current element is less than or equal to the next one. After the loop has processed all elements in the array, the code prints the resulting string t to the output. The output will be a sequence of " > " and " < " symbols, indicating the comparison results between consecutive elements in the array. The exact output will depend on the values in the array a.

5.1.26 Ex. (66) – Which is bigger between n and $n + 1$? (optimisation)

```
a = [2, 3, 4, 5, 9, 8, 3, 8, 3];

l = length(a) - 1;

t = '';
r = '<';

for k = 1:l
    if a(k) > a(k+1)
        r = '>';
    end
    t = [t r];
    r = '<';
end

disp(t);
```

Output:

```
<<<<>><><
```

Like before, this code begins by defining an array a containing a sequence of integers, namely [2, 3, 4, 5, 9, 8, 3, 8, 3]. The variable l is then initialized to store the length of the array minus 1, which is the last valid index within the array. Next, two variables t and r are declared and initialized. Variable t is initialized with a space character, and r is initialized with the less-than symbol (“ < ”). Following the variable declarations, there is a *for-loop* that iterates from k equals 1 up to l. During each iteration of the loop, it checks if the element at index k in the array a is greater than the element at the next index ($k + 1$). If this condition is true, it assigns the greater-than symbol (“ > ”) to the variable r. Otherwise, it assigns the less-than symbol (“ < ”) to r. Once the value of r is determined, it is then appended to the string stored in the variable t. This process continues through each iteration of the loop, building a string in t where each character corresponds to whether the element at the current index is greater than or less than the next element. The code prints the resulting string t to the output, which represents a sequence of “ > ” and “ < ” symbols based on the comparisons between adjacent elements in the array a.

5.1.27 Ex. (67) – Sum two arrays

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

l = length(a) - 1;

for k = 1 : l + 1
    c(k) = a(k) + b(k);
end

disp(['c = ', mat2str(c)]);
```

Output:

```
c = 3,5,7,9,14,14,10
```

This code begins by defining three arrays: *a*, *b*, and *c*. The *a* array contains the values [2, 3, 4, 5, 9, 8, 3], while the *b* array holds [1, 2, 3, 4, 5, 6, 7]. An empty array *c* is also initialized, which will be used to store the result of adding corresponding elements from *a* and *b*. The variable *l* is assigned with the value of *length*(*a*)-1, which represents the length of the array *a* minus one, effectively representing the last index of the arrays. The code then enters a *for-loop* that iterates through indices from 1 to $l + 1$. Above each iteration, it adds the elements at the current index *k* from arrays *a* and *b*, and stores the result in the corresponding index *k* of the *c* array. This process effectively performs element-wise addition between arrays *a* and *b*. At the end of the *for-loop*, the code prints the result by concatenating the string "c = " with the *c* array, which will display the content of array *c* after all the additions have been performed. Note again that in MATLAB string concatenation is made using the "[]" characters.

5.1.28 Ex. (68) – Simple array mapping

```
a = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0];
b = [2, 2, 2, 3, 3, 3, 2, 2, 2, 2, 2];
c = zeros(1, length(a));

for j = 1:length(a)
    c(j) = a(b(j));
end

disp(['c = ', mat2str(c)]);
```

Output:

```
c = 9,9,9,8,8,8,9,9,9,9,9
```

The given code snippet begins by defining the same three arrays: *a*, *b*, and an empty array *c*. Array *a* contains 11 integer elements in descending order from 10 to 0. Array *b* contains 11 integer elements, which represent positions in array *a*. The values in array *b* suggest the possibility of a pattern where some indices may repeat. An empty array *c* is declared to store the results. The code then enters a *for-loop* that iterates from 1 to *length*(*a*), using the variable *j* as the loop counter. Inside the loop, the value of $c(j)$ is assigned a value from array *a* at the index specified by $b(j)$. This means that for each iteration of the loop, $c(j)$ is assigned the value from *a* at the position specified by $b(j)$. The code then prints the contents of array *c* to the console, displaying the result of this operation. Thus, this code performs a series of value assignments from array *a* to array *c* based on the indices specified in array *b*, and then it prints the resulting array *c* to the console.

5.1.29 Ex. (69) – Sum by coordinates (I)

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

l = length(a) - 1;

for k = 1:l+1
    c(k) = a(c(k)) + b(k);
end

disp(['c = ', mat2str(c)]);
```

Output:

```
c = 4,5,6,13,14,15,10
```

In the given code, three arrays a, b, and c are defined with initial values. Array a contains the values [2, 3, 4, 5, 9, 8, 3], array b contains [1, 2, 3, 4, 5, 6, 7], and array c is initially set to [1, 1, 1, 4, 4, 4, 6]. The variable l is defined to store the length of array a minus one. The code then enters a *for-loop* with the variable k ranging from 1 to $l + 1$. Inside the loop, each element of array c at index k is updated. Namely, $c(k)$ is assigned the value of $a(c(k)) + b(k)$. This means that for each element $c(k)$, it looks up the value at the same index in array a, adds the corresponding value from array b, and stores the result back in array c at index k. This process is repeated for all elements in the specified range of k. At the end of the cycle, the code prints the updated array c using *disp*([‘c = ‘, *mat2str*(c)]). Thus, this code modifies the values in array c based on the values in arrays a and b by using their indices and then displays the updated c array.

5.1.30 Ex. (70) – Sum by coordinates (II)

```
a = [2, 3, 4, 5, 9, 8, 3];
b = [1, 2, 3, 4, 5, 6, 7];
c = [2, 2, 2, 5, 5, 5, 7];
l = length(a);

for k = 1:l
    c(k) = a(c(k)) + b(c(k));
end

disp(['c = ', mat2str(c)]);
```

Output:

```
c = 5,5,5,14,14,14,10
```

In this example, there are three arrays: *a*, *b*, and *c*, each containing a series of integer values. The arrays *a* and *b* are initialized with values, and *c* initially holds a set of integer values that represent coordinate positions above the other two arrays. The value returned by function *length*(*a*) is assigned to variable *l*, which represents the index of the last element in the array *a*. The code enters a *for-loop* that iterates from *k* equal to 1 to *l*, inclusive. Inside the loop, the elements of the *c* array are modified based on the values in arrays *a* and *b*. For each *k* in the loop, the value at index *k* in array *c* is updated to be the sum of *a*(*c*(*k*)) and *b*(*c*(*k*)). Next, outside the loop, the result is printed to the console, showing the updated values in array *c*. This code essentially modifies the *c* array by adding values from arrays *a* and *b* based on the indices specified in *c*, and it prints the resulting array *c* to the console.

5.1.31 Ex. (71) – Cutoff value

```
a = [2, 3, 4, 5, 9, 8, 3];
b = [1, 2, 3, 4, 5, 6, 7];
c = [2, 2, 2, 5, 5, 5, 7];
l = length(a);

for k = 1:l

    if a(c(k)) + b(c(k)) > 5
        c(k) = a(c(k)) + b(c(k));
    else
        c(k) = 0;
    end
end

disp(['c = ', mat2str(c)]);
```

Output:

```
c = 0,0,0,14,14,14,10
```

In the above code, there are three arrays *a*, *b*, and *c*, each containing a sequence of numeric values. The length of array *a* is stored in the variable *l*. The code then enters a loop that iterates from 1 to *l*. Inside the loop, for each value of *k*, it calculates the sum of *a*(*c*(*k*)) and *b*(*c*(*k*)). If this sum is greater than 5, it assigns this sum to *c*(*k*). Otherwise, it sets *c*(*k*) to 0. Lastly, after the loop completes, it uses *disp* to print the resulting array *c*, which has been modified based on the conditions described above.

5.1.32 Ex. (72) – Swap array elements by pattern

```
% swap array elements by pattern.

a = [2, 3, 4, 5, 9, 8, 3];
b = [1, 2, 3, 4, 5, 6, 7];
c = [0, 1, 1, 0, 0, 0, 1];

l = length(a);

for k = 1:l
    if c(k) == 1
        t = a(k);
        a(k) = b(k);
        b(k) = t;
    end
end

disp(['a = ', mat2str(a)]);
disp(['b = ', mat2str(b)]);
```

Output:

```
a = 2,2,3,5,9,8,7
b = 1,3,4,4,5,6,3
```

This code swaps elements between two arrays a and b based on a corresponding pattern defined in array c. The code begins by defining three arrays: a, b, and c, each containing a set of values. These arrays represent the data that will be manipulated. The variable l is assigned the length of array a, which is used as the loop termination condition in the subsequent *for-loop*. Inside the *for-loop*, a counter variable k is used to iterate through each element of the arrays. Within the loop, an *if* statement checks if the value of $c(k)$ is equal to 1. If $c(k)$ is indeed equal to 1, it means that a swap operation should be performed for the current elements. Inside the *if* block, the values of $a(k)$ and $b(k)$ are swapped using the temporary variable t. This is done to exchange the corresponding elements of arrays a and b when the pattern in array c dictates it. After the *for-loop* completes execution, the code prints the updated arrays a and b to the console, showing the result of the swapping operation. Note that this code swaps elements between arrays a and b based on the pattern defined in array c, effectively modifying the contents of arrays a and b accordingly.

5.1.33 Ex. (73) – Mix array based on pattern

```
% mix array based on pattern.

a = [2, 3, 4, 5, 9, 8, 3];
b = [1, 2, 3, 4, 5, 6, 7];
c = [0, 1, 1, 0, 0, 0, 1];
l = length(a);

for k = 1:l
    if c(k) == 1
        c(k) = a(k);
    else
        c(k) = b(k);
    end
end

disp(['c = ', num2str(c)]);
```

Output:

```
c = 1,3,4,4,5,6,3
```

The example combines two arrays, *a* and *b*, into a new array *c* based on a pattern defined by the *c* array. The *a* array contains the elements [2, 3, 4, 5, 9, 8, 3], and the *b* array contains [1, 2, 3, 4, 5, 6, 7]. Additionally, there is a *c* array with binary values [0, 1, 1, 0, 0, 0, 1]. The goal here is to create a new array *c* with the same length as *a* and *b* and populate it based on the pattern found in *c*. Thus, a *for-loop* iterates through each index from 1 to the length of the arrays (*l*), and at each iteration it checks the corresponding element in the *c* array (*c*(*k*)). If *c*(*k*) is equal to 1, it assigns the value from the array *a* at index *k*, to *c*(*k*). In contrast, if *c*(*k*) is not equal to 1 (i.e., 0), it assigns the value from the *b* array at index *k*, to *c*(*k*). After the loop completes, the resulting *c* array contains elements that are either from *a* or *b* depending on the pattern defined in *c*. At the end, the implementation prints the contents of the *c* array by using the *disp* function, thus displaying the mixed array based on the pattern.

5.1.34 Ex. (74) – Swap array values

```
% swap array values.

a = {'a', 'a', 'a', 'a', 'a', 'a'};
b = {'b', 'b', 'b', 'b', 'b', 'b'};
l = length(a) - 1;

for k = 1:l+1
    t = a{k};
    a{k} = b{k};
    b{k} = t;
end

disp(['a = ', strjoin(a, ', ')]);
disp(['b = ', strjoin(b, ', ')]);
```

Output:

```
a = b,b,b,b,b,b
b = a,a,a,a,a,a
```

The provided example is designed to swap values between two arrays, *a* and *b*, using a loop. Two arrays, *a* and *b*, are initially defined with six identical elements each, represented as strings. Additionally, a variable *l* is declared and assigned the value of the length of array *a* minus 1. The core of the code lies within a *for-loop* that iterates from 1 to *l*, inclusive, using the loop variable *k*. Within this loop the current element of *a* at index *k* is temporarily stored in a variable *t*. The element at index *k* in array *a* is replaced with the corresponding element from array *b*. The element at index *k* in array *b* is assigned the value stored in the temporary variable *t*. This effectively swaps the values of the two arrays at position *k*. Once the loop completes, the swapped arrays *a* and *b* are printed to the console, indicating the changes that occurred as a result of the swap operation. Please note the very first presence of the *strjoin* function which is used to concatenate strings and arrays.

5.1.35 Ex. (75) – Intermittent value swap

```
% ziperr - intermitent value swap.

a = {'a', 'a', 'a', 'a', 'a', 'a'};
b = {'b', 'b', 'b', 'b', 'b', 'b'};

l = length(a) - 1;


for k = 1:2:l+1
    t = a{k};
    a{k} = b{k};
    b{k} = t;
end

disp(['a = ', strjoin(a)])
disp(['b = ', strjoin(b)])
```

Output:

```
a = a,b,a,b,a,b,a,
b = b,a,b,a,b,a,b,
```

This code demonstrates an intermitent value swap operation between two arrays, *a* and *b*. Initially, both arrays, *a* and *b*, are defined and filled with identical values, where *a* contains multiple instances of *a* and *b* contains multiple instances of *b*. The variable *l* is assigned the value of the length of array *a* minus 1, which determines the limit for the loop. Inside the *for-loop*, the code iterates through the indices of the arrays from 1 to *l*. Within each iteration, the loop counter *k* is incremented by 1 (i.e., with a step of 2), effectively skipping every other index. Since MATLAB does not support modifying the loop variable inside the loop, the increment step is included in the *for-loop* definition (i.e., 1:2:l + 1). Then, the code performs a swap operation between the elements at the

current k index in arrays a and b. This swap operation exchanges the values of $a(k)$ and $b(k)$. Next, the code prints the contents of arrays a and b after the intermitent value swap operation has been completed, displaying the updated contents of both arrays. As a result, this code swaps values between the a and b arrays, but it does so intermittently by skipping every other index during the swap operation. Thus, the a and b arrays will have their values exchanged based on this pattern.

5.1.36 Ex. (76) – Reverse string

```
b = {'a', 'b', 'c', 'd', 'e', 'f'};

n = length(b);
c = cell(1, n);

for i = 1:n
    c{i} = b{n-i+1};
end

disp(c);
```

Output:

```
f,e,d,c,b,a
```

The following code snippet begins by defining an array b with elements a, b, c, d, e, f, essentially creating an array containing individual characters. Next, it calculates the length of the array b and stores it in the variable n. In this case, n will be equal to 6, as there are six elements in the array b. Then, a new empty array c is declared. This array c will be used to store the elements of array b in reverse order. The code enters a *for-loop* that iterates from $i = 1$ to $i = n$. Inside the loop, it assigns the value of $b\{n\text{-}i + 1\}$ to the corresponding index in array c. This effectively reverses the order of the elements from array b and stores them in array c. Next, the *disp*(c) statement is used to display the contents of the reversed array c in the output. Thus, this code takes an array b, which contains characters, and reverses the order of its elements, storing the reversed elements in a new array c.

5.1.37 Ex. (77) – The welding of array values

```
% intermitent melting.

a = [1, 2, 3, 4, 5, 6, 7];
b = [2, 2, 2, 2, 2, 2, 2];
l = length(a) - 1;

for k = 2:2:l
    a(k) = a(k) + b(k);
    b(k) = a(k);
end

disp(['a = ', mat2str(a)]);
disp(['b = ', mat2str(b)]);
```

Output:

```
a = 1,4,3,6,5,8,7
b = 2,4,2,6,2,8,2
```

This code is performing a series of operations on two arrays, *a* and *b*, both of which initially contain elements from 1 to 7. The code begins by initializing the two arrays, *a* and *b*, where *a* contains elements from 1 to 7, and *b* contains only the number 2 repeated seven times. Next, it calculates the length of the array *a* and stores it in the variable *l*. The variable *l* will be used to control the loop. The code then enters a *for-loop*, where it iterates through the indices of the arrays from 2 to *l*, with a step of 2 (2:2:*l*). This means that *k* will skip every other index in the loop. The element at index *k* of array *a* is updated by adding the element at the same index *k* of array *b* to it. This effectively accumulates the values from array *b* into array *a* for the skipped indices. The element at index *k* of array *b* is then updated to match the new value of *a*(*k*). This synchronizes the values in arrays *a* and *b* at the skipped indices. Once the loop is completed, the code prints out the contents of both arrays *a* and *b*. Thus, this code performs an "intermittent melting" operation by skipping every other index in the arrays *a* and *b*, updating the values in *a* with the corresponding values in *b*, and then synchronizing the values in *b* with the new values in *a*. The result is printed to the console via the *disp* function.

5.1.38 Ex. (78) – Static *modulo* - fill up array with *modulo*

```
a = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0];
b = zeros(1, length(a));

for j = 1:length(a)
    b(j) = mod(a(j), 3);
end

disp(['c = ', mat2str(b)]);
```

Output:

```
c = 1,0,2,1,0,2,1,0,2,1,0
```

The provided code begins by declaring two arrays, *a* and *b*. The array *a* is initialized with ten integer values in descending order from 10 to 0. The array *b* is initialized as an empty array. Namely, the array *b* is pre-allocated with zeros for efficiency. The size of *b* is set to match the size of *a*. The code then enters a *for-loop* with the variable *j* starting from 1 and continuing until it reaches 10 (inclusive). Within this loop, each element of the *b* array is assigned a value calculated as the remainder of dividing the corresponding element from the *a* array ($a(j)$) by 3. This effectively computes the modulo 3 of each element in *a* and stores the result in the corresponding position in *b*. Next, the code prints the value of *b* using the *disp* function.

5.1.39 Ex. (79) – Dynamic *modulo* - take $a(i)$ modulo j

```
a = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0];
b = zeros(size(a));

for j = 1:length(a)
    b(j) = mod(a(j), j);
end

disp(['c = ', mat2str(b)]);
```

```
Output:

c = 0,1,2,3,1,5,4,3,2,1,0
```

In the provided code snippet, we have two arrays *a* and *b* initialized, and a loop is used to perform some operations and populate the *b* array based on the values in *a*. Two arrays, *a* and *b*, are declared. Variable *a* contains eleven integer values ranging from 10 to 0, and *b* is initially an empty array. Here, *b* is preallocated to the same size as *a* with *zeros*(*size*(*a*)). A *for-loop* is set up with the variable *j* ranging from 1 to *length*(*a*). This loop will iterate a total of 11 times. Inside the loop, there is an assignment statement. For each iteration, the value at index *j* in array *a* is taken ($a(j)$), and the modulo function (i.e., *mod*) is applied to it. The divisor in the modulo operation is $(j + 1)$. The result of this operation is then assigned to the corresponding index *j* in array *b*. Essentially, it calculates the remainder when the value in *a* is divided by $j + 1$, and stores that remainder in *b*. Next, outside the loop, a *disp* statement is used to display the contents of array *b*. It creates a string concatenating "c = " and the array *b*. This will display the values of array *b* as a string with "c = " as a prefix.

5.1.40 Ex. (80) – Convert a string to an array

```
a = '0|13|55|56|1|30|123';
b = '5|33|55|90|1|22|127';

aa = strsplit(a, '|');
bb = strsplit(b, '|');
cc = zeros(1, length(aa));

for i = 1:length(aa)
    cc(i) = str2double(aa{i}) + ...
            str2double(bb{i});
end

disp(cc);

% Or, a second version without loops:

a = '0|13|55|56|1|30|123';
b = '5|33|55|90|1|22|127';

cc = arrayfun(
@(x, y) str2double(x) + ...
        str2double(y), ...
        strsplit(a, '|'), ...
        strsplit(b, '|')
             );

disp(cc);
```

```
Output:

5,46,110,146,2,52,250
```

The code from above performs a series of operations on two given strings *a* and *b*. These strings contain numerical values separated by the "|" character. The code aims to split these strings into arrays, calculate the sum of corresponding elements from both arrays, and store the results in a new array *cc*, which are then printed in the output. The code initially begins by defining two strings, *a* and *b*, each containing a series of numerical values separated by "|" characters. Next, it splits the strings *a* and *b* into arrays using the *strsplit* function. This operation separates the numerical values at each "|" character and stores them in the arrays *aa* and *bb*, respectively. Afterward, an empty array *cc* is declared to store the results of the element-wise addition of the values from aa and *bb*. A *for-loop* is used to iterate through the elements of *aa*. The loop runs from $i = 1$ to *length*(*aa*), ensuring that it goes through all elements in the arrays. During each iteration, the code converts the elements at the current index *i* in both *aa* and *bb* to numbers using *str2double*() and then adds them together. The result is stored in the *cc* array at the same index *i*. Essentially, this loop calculates the element-wise sum of the corresponding elements from *aa* and *bb* and populates the *cc* array with the results. Lastly, the code prints the contents of the *cc* array, which now holds the summed values of the corresponding elements from the original *a* and *b* strings. Note that MATLAB is matrix-oriented and

there are often more efficient ways (built-in functions) to perform operations like this without explicit loops.

5.1.41 Ex. (81) – The rule of three simples

```
a = [5, 1, 8, 4, 6, 2, 9, 8];

n = length(a);
maxVal = 0;
m = 100;

% Find the maximum value
for i = 1:n
    if a(i) > maxVal
        maxVal = a(i);
    end
end

% Calculate percentages
for i = 1:n
    p = (m / maxVal) * a(i);
    fprintf('%g%%\n', p);
end
```

Output:

```
55.5556%
11.1111%
88.8889%
44.4444%
66.6666%
22.2222%
100%
88.8889%
```

This code begins by defining an array *a* with a series of numeric values. It calculates the length of the array *n* and initializes some variables: *maxVal* is set to 0, *m* is assigned the value 100, and an empty array *t* is created. The first *for-loop* iterates through the elements of the array *a*. Within this loop, there is an *if* statement that checks if the current element $a(i)$ is greater than the current maximum value *maxVal*. If it is, the *maxVal* variable is updated to the value of $a(i)$, essentially finding the maximum value in the array. After finding the maximum value, a second *for-loop* goes through the elements of the array *a* again. Inside this loop, it calculates a new variable *p* by multiplying *m* by the ratio of the current element $a(i)$ to the maximum value *maxVal*. This is done to scale the values in the array *a* proportionally based on the maximum value. Next, the code prints out the calculated *p* value followed by the "%" symbol. This code essentially scales the values in array *a* so that they represent percentages relative to the maximum value found in the array. Note that the scaled values are then printed to the console by using *fprintf*. The "%g" format specifier automatically chooses between "%f" and "%e", whichever is more compact, for displaying numbers. The double "%%" is used to print a literal percent sign. Note that MATLAB has a built-in function to find the maximum value in an array. In practice, one can simply use $maxVal = max(a)$; to find the maximum value in the array *a*.

5.1.42 Ex. (82) – Average, standard deviation and coefficient of variation

```
a = [5, 6, 2, 9, 44, 200];
n = length(a);

b = 0;
e = 0;

for j = 1:n
    b = b + a(j);
end

x = b / n;

for j = 1:n
    e = e + (a(j) - x)^2;
end

s = sqrt(e / (n - 1));
c = s / x;

fprintf('AV = %.2f\n', x);
fprintf('SD = %.2f\n', s);
fprintf('CV = %.2f\n', c);

% Or, a second version:

a = [5, 6, 2, 9, 44, 200];

x = mean(a);
s = std(a);
c = s / x;

fprintf('AV = %.2f\n', x);
fprintf('SD = %.2f\n', s);
fprintf('CV = %.2f\n', c);
```

Output:

```
AV = 44.33
SD = 77.83
CV = 1.76
```

This source code is fundamental for any scientist or/and engineer, and it calculates and prints three statistical measures (average, standard deviation, and coefficient of variation) for a given array of numbers. The code begins by defining an array *a* containing a set of numerical values. It also initializes two variables, *b* and *e*, to zero. Variable *b* will be used to calculate the sum of all values in the array, and *e* will be used to calculate the sum of squared differences from the mean. Next, the code calculates the length of the array *a* and stores it in the variable *n*. A *for-loop* is used to iterate through the elements of the array *a*. Inside the loop, each element of the array is added to the variable *b*, effectively summing up all the values in the array. After the first loop, the code calculates the mean (average) *x* by dividing the sum *b* by the length *n* of the array *a*. A second *for-loop* is used to iterate through the elements of the array *a* once again. Inside this loop, the code calculates the sum of squared differences (*e*) from the mean for each element of the array. Once the second loop is complete, the code proceeds to calculate the standard deviation

s using the formula for sample standard deviation. It takes the square root of *e* divided by (*n*-1). Next, the code calculates the coefficient of variation *c* by dividing the standard deviation *s* by the mean *x*. The results are printed to the console using the *fprintf* function, displaying the average (AV), standard deviation (SD), and coefficient of variation (CV) along with their respective values. Thus, this code performs basic statistical calculations for a given array of numbers, providing insights into the central tendency, dispersion, and relative variability of the data. A second example (below the first) shows how the three values can be calculated with built-in MATLAB functions (similar to Excel). Note that a new operator is used, namely MATLAB uses "^" for exponentiation. Also, in the case of the *fprintf* function, "%f" is a placeholder for floating-point numbers.

5.1.43 Ex. (83) Horizontal chart from ASCII characters

```
a = [5, 1, 8, 4, 6, 2, 8, 9];

c = '#';
t = '';

n = length(a);

for i = 1:n
    for k = 1:a(i)
        t = [t c];
    end

    t = [t '\n'];
end

fprintf(t);
```

Output:

```
#####
#
########
####
######
##
########
#########
```

The above example initializes an array *a* with a sequence of integers. It also defines two strings, *c* and *t*, with initial values. The variable *c* is set to "#", and *t* is an empty string. The length of array *a* is stored in the variable *n*. The code then enters a nested loop structure. The outer loop, controlled by the variable *i*, iterates through each element of the array *a*. Inside this loop, there is an inner loop controlled by the variable *k*. The inner loop runs a number of times equal to the value at the current index of *a*(*i*). During each iteration of the inner loop, the character "#" is appended to the string *t*. After the inner loop finishes for a specific *i*, a newline character "\n" is appended to the string *t*. This creates a pattern of "#" characters on each line, where the number of "#" characters on a line is determined by the value at the current index of *a*(*i*). Therefore, the *t* string, is then printed to the command window.

5.1.44 Ex. (84) – Horizontal chart with bars proportional with max from array

```
a = [5, 2, 8, 4, 6, 22, 8, 9];

m = 15;
c = '#';
t = '';
maxVal = 0;

n = length(a);

for i = 1:n
    if a(i) > maxVal
        maxVal = a(i);
    end
end

for i = 1:n
    f = floor((m/maxVal) * a(i));
    for k = 1:f
        t = [t c];
    end
    t = [t '\n'];
end

fprintf(t);
```

Output:

```
###
#
#####
##
####
###############
#####
######
```

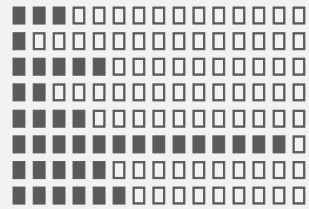
The above example performs several operations on an array *a* containing numeric values, and then generates a visual representation of the data using characters. First, an array *a* is defined, which contains the numbers [5, 2, 8, 4, 6, 22, 8, 9]. Next, some variables are initialized: (1) Variable *m* is set to 15. (2) Variable *c* is set to "#", representing a character used to visualize the data. (3) Variable *t* is an empty string that will accumulate characters to create a visual representation. (4) Variable *maxVal* is initialized to 0 and will be used to find the maximum value in the array *a*. (5) Variable *n* is assigned the length of the array *a*. The code then enters a loop to find the maximum value in the array *a*. It iterates through each element of *a* using a *for-loop*, and if the current element (*a*(*i*)) is greater than the current maximum (*maxVal*), it updates the *maxVal* variable with the value of *a*(*i*). This loop effectively finds the maximum value in the array. The detection of the maximum value, allows the code to enter another loop to generate a visual representation of the data. It iterates through each element of *a* again, using a *for-loop* with *i* as the loop variable. Inside this loop variable *f* is calculated as *floor*((*m*/*maxVal*) * *a*(*i*)). This computes a scaled value of *a*(*i*) relative to the maximum value (*maxVal*) and scales it to fit within the range of 0 to *m*. The *floor* function ensures that *f* is an integer. Another nested *for-loop* with *k* as the loop variable is used to append *f* copies of the character *c* to the string *t*. This creates a visual representation where the number of characters represents the scaled value of *a*(*i*). Once *f* characters are added, a newline character "\n" is injected to start a new line. Lastly, the code prints the accumulated string *t*, which displays the

visual representation of the scaled values of the elements in the array a using the character "#". The number of "#" characters on each line corresponds to the scaled value of the corresponding element in the array *a*.

5.1.45 Ex. (85) – Horizontal chart with UTF characters proportional with max array

```
a = [5, 2, 8, 4, 6, 22, 8, 9];
m = 15;
t = '';

maxVal = max(a);
n = length(a);

for i = 1:n
    f = floor((m / maxVal) * a(i));

    for k = 1:m
        if k <= f
            t = [t '■'];
        else
            t = [t '□'];
        end
    end

    t = [t '\n'];
end

fprintf(t);
```

Output:



This code is similar to the previous one, and performs several operations on an array *a* and generates a UTF-8 text-based bar chart, where the length of each bar is proportional to the corresponding element in the array *a* concerning the maximum value in the array. An array *a* is defined with a set of numeric values. Variables *m*, *t*, and *maxVal* are initialized. Variable *m* represents the total number of characters for each bar in the chart, *t* will store the final text result, and *maxVal* will keep track of the maximum value in the array. The length of the array *a* is determined and stored in the variable *n*. This time, instead of a *for-loop*, the *max* function is used to find the maximum value in the array. Thus, the *max* function updates *maxVal* with the current element. A *for-loop* is used to create the bar chart. It iterates through each element of *a* again. Inside the second loop, a variable *f* is calculated. It represents the length of the current bar and is calculated as (*m* / *maxVal*) * *a*(*i*), which scales the length of the bar based on the ratio of the current element value to the maximum value in the array. Within a nested loop (*for-loop* with variable *k*), the code checks if *k* is less than *f*. If *k* is less than *f*, it appends a filled square character ("■") to the string *t*, indicating the filled portion of the bar. Otherwise, it appends an empty square character ("□") to represent the empty portion of the bar. After each bar is constructed, the code adds a newline character to *t* to start a new line for the next bar.

Next, the resulting *t* variable, containing the text-based bar chart, is printed to the console. This code essentially visualizes the values in the array *a* by representing them as bars in a simple text format, where the length of each bar is proportional to the value it represents concerning the maximum value in the array.

Traversal of Multidimensional Arrays

6

Traversal of multidimensional arrays is a fundamental operation in computer programming, particularly when dealing with complex data structures and matrices [1]. A multidimensional array is essentially an array of arrays, where each element can be another array. This arrangement allows for the representation of data in a grid or matrix-like format, making it well-suited for various applications, including image processing, data analysis, and simulations. The iteration through multidimensional arrays involves systematically visiting each element within the structure. This process is crucial for performing tasks such as data manipulation, searching for specific values, or performing mathematical operations across the array. Depending on the programming language, there are various techniques and loops that can be used for efficient multidimensional array traversal. In the following examples, we will explore the common methods and techniques used to traverse multidimensional arrays in MATLAB, providing insights into how to access and manipulate data stored in these complex structures. Whether one is working with two-dimensional arrays, three-dimensional arrays, or even higher-dimensional arrays, understanding how to traverse them is a fundamental skill for a wide range of software development and data analysis tasks. The following examples show different cases. Some classic examples are presented first, using nested loops for multidimensional array traversals, and other more interesting cases show the same types of traversals using one *for-loop* and mathematical formulas for guidance.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_6

6.1.1 Ex. (86) – Accessing the elements of matrix *A*

```
% 2D

A = {
    'a', 88, 146;
    'b', 34, 124;
    'c', 96, 564;
    100, 12, 'd'
     };

disp(A{2, 3});
```

Output:

```
124
```

The above statement is working with a 2D array named *A*. This array is composed of several sub-arrays, each of which contains a mix of numbers and strings. In this code, we define a 2D array *A*. This array is structured as an array of arrays. Each sub-array represents a row of data, and within each sub-array, we have elements that can be either strings or numbers. The *A* array contains four sub-arrays, and each of them holds a combination of string and number values: The first sub-array contains three elements: the string "a", the number 88, and the number 146. The second sub-array contains three elements: the string "b", the number 34, and the number 124. The third sub-array contains three elements: the string "c", the number 96, and the number 564. The fourth sub-array contains three elements: the number 100, the number 12, and the string "d". After defining the 2D array *A*, the code prints an element from within the array. Specifically, it prints $A\{2, 3\}$. This notation means that it accesses the second sub-array within *A* (arrays are 1-indexed in Matlab), and from that sub-array, it retrieves the third element, which is the number 124. Thus, when this code is run, it will output 124 to the console.

6.1.2 Ex. (87) – Accessing the elements o matrix A using nested for loops

```
A = {
   'a', 88, 146;
   'b', 34, 124;
   'c', 96, 564;
   100, 12, 'd'
    };

t = '';

for i = 1:size(A, 1)
    for j = 1:size(A, 2)

        t = [t, '\n A[', num2str(i), ...
            '][', num2str(j), ']=', ...
            num2str(A{i, j})];

    end
end

fprintf(t);
```

Output:

```
A[1][1]=a
A[1][2]=88
A[1][3]=146
A[2][1]=b
A[2][2]=34
A[2][3]=124
A[3][1]=c
A[3][2]=96
A[3][3]=564
A[4][1]=100
A[4][2]=12
A[4][3]=d
```

This code initializes a two-dimensional array A containing various values, including strings and numbers. The array consists of four subarrays, each containing three elements. These subarrays represent rows and contain a mix of string and numeric values. Next, it declares an empty string variable t which will be used to store the result of the iteration over the A array. The code proceeds to iterate through the array A using nested for loops. The outer loop iterates through the rows of the A array using the variable i, and the inner loop iterates through the elements of each row using the variable j. Within the nested loops, the code appends to string t, information about each element of the array A. Namely, it creates a string in the format "$A[i][j] =$ value", where i and j are the indices of the element, and value is the actual value stored in the array A at that position. Thus, the code prints the accumulated string t to the console, displaying the information about each element in the array A, including their positions within the array.

6.1.3 Ex. (88) – Traverse a matrix with a single *for loop* (I)

```
m = [
    2, 4, 6;
    3, 5, 6;
    3, 5, 4
    ];

i = 1;
j = 1;

n1 = size(m, 1);
n2 = size(m, 2);

q = n1 * n2;

for v = 1:q
    j = mod(v - 1, n2) + 1;
    if j == 1 && v ~= 1 && i < n1 && v ~= q
        i = i + 1;
    end

    disp(['m(' num2str(i) ',' ...
        num2str(j) ')=' ...
        num2str(m(i, j))] ...
        );
end
```

Output:

```
m(1,1)=2
m(1,2)=4
m(1,3)=6
m(2,1)=3
m(2,2)=5
m(2,3)=6
m(3,1)=3
m(3,2)=5
m(3,3)=4
```

This source code defines a 2D array *m*, initializes several variables, and then uses a loop to iterate through the elements of the array *m*. The 2D array *m* is defined as a 3×3 matrix with specific integer values. Two variables *i* and *j* are initialized to 1, and two variables *n1* and *n2* are assigned the lengths of the matrix *m* (number of rows and columns, respectively). The variable *q* is calculated as the product of *n1* and *n2*. A *for-loop* is used to traverse the elements of the matrix. The loop runs from $v = 1$ to *q*. Within the loop, variable *j* is calculated as the remainder of *v* divided by *n2*, which effectively represents the column index within the matrix. An *if* condition checks if *j* is 1 (indicating a new row) and *v* is not 1 (to exclude the very first iteration) and if *i* is less than *n1* (indicating there are more rows to traverse), and *v* is not equal to *q*. If this condition is met, *i* is incremented by 1, which signifies moving to the next row in the matrix. Next, a *disp* statement is used to display the value of the matrix element at the current indices *i* and *j* in the format "m("+*i*+","+*j*+") =" + *m*(*i*,*j*). In essence, this code iterates through the elements of the 2D array *m* row by row, printing out each element along with its row and column indices.

6.1.4 Ex. (89) – Traverse a matrix with a single *for loop* (II)

```
A = {
   'a', 88, 146;
   'b', 34, 124;
   'c', 96, 564;
   100, 12, 'd'
    };

t = '';

% rows and columns.
[n, m] = size(A);

i = 1;
j = 1;

for v = 1:(n * m)
    j = mod(v - 1, m) + 1;

    if v ~= 1 && j == 1
        i = i + 1;
    end

    t = [t, ...
    num2str(v), ' A(', ...
    num2str(i), ',', ...
    num2str(j), ')=', ...
    num2str(A{i,j}), '\n' ...
        ];
end

fprintf(t);
```

Output:

```
1 A(1,1)=a
2 A(1,2)=88
3 A(1,3)=146
4 A(2,1)=b
5 A(2,2)=34
6 A(2,3)=124
7 A(3,1)=c
8 A(3,2)=96
9 A(3,3)=564
10 A(4,1)=100
11 A(4,2)=12
12 A(4,3)=d
```

The code defines a 2D array A containing a mix of strings and numbers. It initializes some variables and then enters a loop to iterate through each element of the 2D array and print its position and value. First, the code initializes an empty string t which will be used to accumulate the output. It also determines the number of rows n and columns m in the array A. Then, it sets up two counters, i and j, to keep track of the current row and column in the array. The code enters a nested loop that runs $n \times m$ times, where n is the number of rows and m is the number of columns in the array. Within the loop, j is updated to the current column index by taking the remainder of v (the loop counter) divided by m. An *if* statement checks if v is not 1 (i.e., we are not at the first element of the array) and if j is 1 (i.e., it reached the end of a row). If this condition is met, i is incremented to move to the next row. The code then constructs a string t that contains the current element (v), the coordinate position of the element on the matrix ($A\{i,j\}$) and the value found on it, where i and j represent the row and column indices, and appends it to the t string. Next, a newline character ("\n") is added to separate the information of each element. Once the loop finishes, the accumulated string t is printed to the console. Thus,

this code is used to display the indices and values of all elements in the 2D array *A*, with each element position in the array denoted by $A\{i,j\}$.

6.1.5 Ex. (90) – Accessing the elements of a 3D array

```
% 3D

A = {
    {
        {'a', 88, 146},
        {'b', 34, 124},
        {'c', 96, 564},
        {100, 12, 'd'}
    },
    {
        {'e', 48, 996},
        {'f', 34, 554},
        {'g', 26, 884},
        {111, 92, 'h'}
    }
};

disp(A{2}{3}{1});
```

Output:

```
g
```

This code defines a 3D array called *A* that contains nested arrays with multiple elements. Each of these nested arrays represents a 2D array, and they are grouped within the larger 3D structure. Inside the 3D array *A*, there are two main levels of nested arrays. The first level contains two sub-arrays, and the second level contains arrays with a mix of strings and numbers as their elements. For example, the first nested array $A\{1\}$ contains four sub-arrays. Each of these sub-arrays consists of three elements. The elements include strings like "a," "b," "c," and "d," as well as numeric values like 88, 146, 34, 124, 96, 564, 100, and 12. Some sub-arrays even contain a mix of both strings and numbers. Similarly, the second nested array $A\{2\}$ also contains four sub-arrays with similar structures. The last line of code, *disp*($A\{2\}\{3\}\{1\}$);, is trying to access an element within the *A* array. Specifically, it accesses the third sub-array ($\{3\}$) within the second nested array ($\{2\}$) and print the content of the first element ($\{1\}$). The value in this specific position would be "g".

6.1.6 Ex. (91) – Traverse a 3D object with a single *for loop*

```
A = {
    {
        {'a', 55, 146},
        {'b', 34, 124},
        {'c', 96, 564},
        {100, 12, 'd'}
    },
    {
        {'e', 88, 146},
        {'f', 34, 124},
        {'g', 96, 564},
        {100, 12, 'h'}
    },
    {
        {'i', 88, 146},
        {'j', 34, 124},
        {'k', 96, 564},
        {100, 12, 'k'}
    },
    {
        {'m', 88, 146},
        {'n', 34, 124},
        {'o', 96, 564},
        {100, 12, 'p'}
    },
    {
        {'q', 88, 146},
        {'r', 34, 124},
        {'s', 96, 564},
        {100, 12, 't'}
    }
};

t = "";

s = length(A);         % 5 matrices.
m = length(A{1});      % 4 rows.
n = length(A{1}{1});   % 3 columns.
```

```
Output:

1 A[1][1][1]=a
2 A[1][1][2]=55
3 A[1][1][3]=146
4 A[1][2][1]=b
5 A[1][2][2]=34
6 A[1][2][3]=124
7 A[1][3][1]=c
8 A[1][3][2]=96
9 A[1][3][3]=564
10 A[1][4][1]=100
11 A[1][4][2]=12
12 A[1][4][3]=d
13 A[2][1][1]=e
14 A[2][1][2]=88
15 A[2][1][3]=146
16 A[2][2][1]=f
17 A[2][2][2]=34
18 A[2][2][3]=124
19 A[2][3][1]=g
20 A[2][3][2]=96
21 A[2][3][3]=564
22 A[2][4][1]=100
23 A[2][4][2]=12
24 A[2][4][3]=h
25 A[3][1][1]=i
26 A[3][1][2]=88
27 A[3][1][3]=146
28 A[3][2][1]=j
29 A[3][2][2]=34
30 A[3][2][3]=124
31 A[3][3][1]=k
32 A[3][3][2]=96
33 A[3][3][3]=564
34 A[3][4][1]=100
```

```
i = 1;
j = 1;
d = 1;
k = 1;

q = n * m * s;

for v = 1:q

    k = mod(v - 1, m * n) + 1;
    j = mod(v - 1, n) + 1;

    if v ~= 1 && j == 1
        i = i + 1;
    end

    if v ~= 1 && k == 1
        i = 1;
        d = d + 1;
    end

    t = [t, ...
    num2str(v), ' A[', ...
    num2str(d), '][', ...
    num2str(i), '][', ...
    num2str(j), ']=' ...
    num2str(A{d}{i}{j}), ...
    newline
        ];

end

fprintf('%s', t);
```

```
35 A[3][4][2]=12
36 A[3][4][3]=k
37 A[4][1][1]=m
38 A[4][1][2]=88
39 A[4][1][3]=146
40 A[4][2][1]=n
41 A[4][2][2]=34
42 A[4][2][3]=124
43 A[4][3][1]=o
44 A[4][3][2]=96
45 A[4][3][3]=564
46 A[4][4][1]=100
47 A[4][4][2]=12
48 A[4][4][3]=p
49 A[5][1][1]=q
50 A[5][1][2]=88
51 A[5][1][3]=146
52 A[5][2][1]=r
53 A[5][2][2]=34
54 A[5][2][3]=124
55 A[5][3][1]=s
56 A[5][3][2]=96
57 A[5][3][3]=564
58 A[5][4][1]=100
59 A[5][4][2]=12
60 A[5][4][3]=t
```

This source code defines a multi-dimensional array *A*, consisting of 5 matrices, each containing 4 rows and 3 columns of elements. These elements can be a combination of strings and numbers. The code initializes variables *t*, *s*, *m*, and *n*. Variable *t* will be used to accumulate the results, *s* represents the number of matrices in the array, *m* represents the number of rows in each matrix, and *n* represents the number of columns in each matrix. Subsequently, the code uses nested loops to iterate through the elements of the multi-dimensional array *A*. It calculates the total number of elements *q* in the array, which is the product of the number of matrices, rows, and columns. Inside the loop, the code calculates the current position *k*, *j*, *i*, and *d* based on the current iteration *v*. The variables *i*, *j*, *d*, and *k* represent the current row index, column index, matrix index, and overall index, respectively. The code then appends a string to the variable *t* in the format "v A[d][i][j] = value" for each element in the array, where *v* is the current index, *d* is the current matrix index, *i* is the current row index, *j* is the current column index, and *value* is the value of the corresponding element in the array *A*. Next, the code prints the accumulated string *t* to

the console, displaying the index and values of all elements in the multi-dimensional array *A*. Please note the use of the *length* function to get the dimensions of the 3D structure.

6.1.7 Ex. (92) – Traverse a 2D object with a single *for loop* and integer division

```
% integer division 2D one-for loop
% no if then involved.

A = {
    {'a', 88, 146},
    {'b', 34, 124},
    {'c', 96, 564},
    {100, 12, 'd'}
    };

t = '';

% n: number of rows,
% m: number of columns.

n = numel(A);
m = numel(A{1});

for k = 1:(n*m)
    j = mod(k - 1, m) + 1;
    i = floor((k - 1) / m) + 1;

    t = [t, ...
        num2str(k), ' A[', ...
        num2str(i), '][', ...
        num2str(j), ']=' ...
        num2str(A{i}{j})
    ];

    t = [t, '\n'];
end

fprintf(t);
```

Output:

```
1 A[1][1]=a
2 A[1][2]=88
3 A[1][3]=146
4 A[2][1]=b
5 A[2][2]=34
6 A[2][3]=124
7 A[3][1]=c
8 A[3][2]=96
9 A[3][3]=564
10 A[4][1]=100
11 A[4][2]=12
12 A[4][3]=d
```

The above source code is designed to perform integer division within a two-dimensional array using a single *for-loop*. It starts with the definition of a two-dimensional array *A*, containing various values and strings. In MATLAB, we use cell arrays ("{}") to store mixed types of data like strings and numbers in a matrix-like structure. The code also initializes an empty string *t* to store the output and variables *n* and *m* to store the number of rows and columns in array *A*. Two variables *i* and *j* are also initialized. The primary goal of this code is to traverse the entire 2D array *A* using a single loop and output the indices and values of each element in a formatted string. The *for-loop* iterates from 1 to $n \times m$, where *n* is the number of rows and *m* is the number of columns in array *A*. Inside the loop the variable *j* is calculated as the remainder of dividing *k* by *m*, which determines the column index. The variable *i* is calculated as the result of dividing *k* by

m, rounded down to the nearest integer, which determines the row index. The code builds a string *t* that contains the current index *k* and the corresponding element from the 2D array *A* using the calculated *i* and *j* values. Next, after the loop, the *fprintf*(*t*); statement is used to display the formatted string *t*, which contains the indices and corresponding elements from the 2D array *A*. This code effectively provides a structured way to access and display the elements of a 2D array while using a single loop to traverse it. This time, please note the use of the *numel* function (NUMber of ELements) to get the total number of elements of *A* or the total number of elements of *A*{1}, thus essentially taken the $n \times m$ size of *A*.

6.1.8 Ex. (93) – Traverse a 2D object with a single for loop using arithmetic operators

```
% no if-then and no integer division (superb).

A = {
    'a', 88, 146;
    'b', 34, 124;
    'c', 96, 564;
    100, 12, 'd'
    };

t = "";
n = size(A, 1); % rows
m = size(A, 2); % columns

for k = 0:(n*m-1)

    j = mod(k, m) + 1;
    i = (k - j + 1) / m + 1;

    t = t + sprintf('%d A{%d,%d}=%s\n', ...
        k, i, j, mat2str(A{i,j}));
end

disp(t);
```

Output:

```
0 A{1,1}=a
1 A{1,2}=88
2 A{1,3}=146
3 A{2,1}=b
4 A{2,2}=34
5 A{2,3}=124
6 A{3,1}=c
7 A{3,2}=96
8 A{3,3}=564
9 A{4,1}=100
10 A{4,2}=12
11 A{4,3}=d
```

This source code defines a 2D array *A* with a mixture of strings and numbers. It initializes an empty string *t* and determines the number of rows in the *A* array (*n*) and the number of columns (*m*). The code then uses a *for-loop* to iterate through all elements of the array *A* using a single counter variable *k*. Inside the loop, it calculates the row *i* and column *j* based on the value of *k* and the dimensions of the array ($n \times m + 1$). The code concludes by printing the contents of string *t*. However, the novelty here is represented by the way variable *i* and *j* are calculated. Note that *i* and *j* are used to calculate the row and column indices for traversing a two-dimensional array *A*. Variable *n* is assigned the value *size*(*A*,1), which represents the number of rows in the array *A*. Variable *m* is assigned the value *size*(*A*,2), which represents the number of columns in the array *A*. The loop iterates over all the elements in the two-dimensional array, and for each iteration a series

of events unfold. The counter k is used as a linear index that ranges from 0 to $n*m-1$. Thus, variable k represents the current position within the flattened representation of the 2D array. The column index j is obtained using the *mod* function to find the remainder of k divided by m, and then adding 1 to adjust for 1-based indexing of MATLAB. In contrast, the row index i is calculated by subtracting $j - 1$ from k, dividing the result by m, and then adding 1, again to compensate for the MATLAB indexing system. This effectively calculates the row index based on the linear index k and the column index j. The values of k, i, and j are then used to access the corresponding element in the two-dimensional array A using $A\{i,j\}$. The values are concatenated to the string t by using the *sprintf* function, to display the current index and the corresponding element in the array. Thus, for each element of A, the code appends a line to the t string containing the following information: the value of k, the coordinates in the array A where the value is located ($A\{i,j\}$), and the actual value itself. Also, a *newline* "\n" is added to separate each line. Please note the first use of the *sprintf* function instead of the *fprintf*. The question now: is what is the difference between the two? Well, the *sprintf* and *fprintf* in MATLAB are used for string formatting, but they serve different purposes and functionalities. The *sprintf* is primarily used for creating formatted strings (thus, it can be used in conjunction with function *disp*). It formats data and returns the formatted string, which can then be stored in a variable for later use or further manipulation. For example, using *sprintf*('Value: %d', 10) creates a string "Value: 10" and stores it in a variable. On the other hand, *fprintf* is designed for outputting formatted strings either to a file or directly to the MATLAB command window. It prints the formatted data to the specified output stream, which by default is the command window, unless directed to a file. An example of its usage would be *fprintf*('Value: %d\n', 10), which displays "Value: 10" in the command window or writes it to a specified file (examples related to files are found at the end of the book). This makes *fprintf* ideal for situations where the immediate display or logging of the formatted string is required. But, the choice between the use of *sprintf* and *fprintf* depends on whether one needs to manipulate and store the formatted string for later (*sprintf*), or display or log it immediately (*fprintf*). Nevertheless, the above example, effectively traverses the 2D array by linearizing the indices i and j and using them to access elements in the array in a row-by-row fashion, regardless of the actual array structure.

6.1.9 Ex. (94) – 3D traversal with one for-loop using only arithmetic operators

```
% 3d scan - one for loop - no if then.

A = {
    {
        {'a', 55, 146},
        {'b', 34, 124},
        {'c', 96, 564},
        {100, 12, 'd'}
    },
    {
        {'e', 88, 146},
        {'f', 34, 124},
        {'g', 96, 564},
        {100, 12, 'h'}
    },
    {
        {'i', 88, 146},
        {'j', 34, 124},
        {'k', 96, 564},
        {100, 12, 'k'}
    },
    {
        {'m', 88, 146},
        {'n', 34, 124},
        {'o', 96, 564},
        {100, 12, 'p'}
    },
    {
        {'q', 88, 146},
        {'r', 34, 124},
        {'s', 96, 564},
        {100, 12, 't'}
    }
};

t = "";
```

Output:

```
0 A{1}{1}{1}=a
1 A{1}{1}{2}=55
2 A{1}{1}{3}=146
3 A{1}{2}{1}=b
4 A{1}{2}{2}=34
5 A{1}{2}{3}=124
6 A{1}{3}{1}=c
7 A{1}{3}{2}=96
8 A{1}{3}{3}=564
9 A{1}{4}{1}=100
10 A{1}{4}{2}=12
11 A{1}{4}{3}=d
12 A{2}{1}{1}=e
13 A{2}{1}{2}=88
14 A{2}{1}{3}=146
15 A{2}{2}{1}=f
16 A{2}{2}{2}=34
17 A{2}{2}{3}=124
18 A{2}{3}{1}=g
19 A{2}{3}{2}=96
20 A{2}{3}{3}=564
21 A{2}{4}{1}=100
22 A{2}{4}{2}=12
23 A{2}{4}{3}=h
24 A{3}{1}{1}=i
25 A{3}{1}{2}=88
26 A{3}{1}{3}=146
27 A{3}{2}{1}=j
28 A{3}{2}{2}=34
29 A{3}{2}{3}=124
30 A{3}{3}{1}=k
31 A{3}{3}{2}=96
32 A{3}{3}{3}=564
33 A{3}{4}{1}=100
```

```
s = size(A, 1);         % 5 matrices
m = size(A{1}, 1);      % 4 rows
n = size(A{1}{1}, 2);   % 3 columns

q = n * m * s;

for v = 0:(q-1)

    k = mod(v, m * n);

    j = mod(v, n) + 1;
    i = (k - j + 1) / n + 1;
    d = (v - k) / (m * n) + 1;

    t = t + sprintf('%d A{%d}{%d}{%d}=%s\n', ...
    v, d, i, j, mat2str(A{d}{i}{j}));
end

disp(t);
```

```
34 A{3}{4}{2}=12
35 A{3}{4}{3}=k
36 A{4}{1}{1}=m
37 A{4}{1}{2}=88
38 A{4}{1}{3}=146
39 A{4}{2}{1}=n
40 A{4}{2}{2}=34
41 A{4}{2}{3}=124
42 A{4}{3}{1}=o
43 A{4}{3}{2}=96
44 A{4}{3}{3}=564
45 A{4}{4}{1}=100
46 A{4}{4}{2}=12
47 A{4}{4}{3}=p
48 A{5}{1}{1}=q
49 A{5}{1}{2}=88
50 A{5}{1}{3}=146
51 A{5}{2}{1}=r
52 A{5}{2}{2}=34
53 A{5}{2}{3}=124
54 A{5}{3}{1}=s
55 A{5}{3}{2}=96
56 A{5}{3}{3}=564
57 A{5}{4}{1}=100
58 A{5}{4}{2}=12
59 A{5}{4}{3}=t
```

As the last example in this subchapter, here, an elegant way to navigate through multi-dimensional arrays is presented, without the need for nested loops, or built-in functions like *floor*. In this example, a 3D array A is defined, which represents a multi-dimensional structure containing strings and numbers. The code then initializes several variables, including t (a string for storing the result), s (the number of matrices or "layers" in A), m (the number of rows in each matrix), n (the number of columns in each matrix), and i, j, d, and k as iteration and indexing variables. A loop runs from 0 to $q–1$, where q is calculated as the product of n (number of columns), m (number of rows), and s (number of matrices), which effectively iterates through all elements in the 3D array A. Within the loop, k is assigned the remainder of v divided by the product of m and n. Variable j is calculated as the modulo of v divided by n, and i is calculated as $(k - j + 1)$ divided by $n + 1$, and d is calculated as $(v - k)$ divided by 1 plus the product of m and n. These calculations help determine the current position within the 3D array A. The code appends information to the t string for each iteration by using the *sprintf* function, thus, showing the current index v and the corresponding value in the A array at the position $\{d\}\{i\}\{j\}$. A line break is also added to separate the entries. At the end of the loop, the code prints the contents of the t string. Thus, this code iterates over the entire 3D array A and prints the indices d, i, j, and the corresponding element from the array, effectively displaying the entire content of the 3D array with their indices. Nevertheless, the main novelty here

is represented by the way variables i, j, and d are computed in order to iterate over the elements of the 3D array A, namely:

$$k = v\%(m \times n)$$

$$j = (v\%n) + 1$$

$$i = \frac{(k - j + 1)}{n + 1}$$

$$d = \frac{(v - k)}{(m \times n) + 1}$$

Namely, variable k represents the position within a matrix (subarray), and variable i is calculated as $(k - j + 1)/n + 1$, which is the result of the division between $k - j$ and the number of columns $n + 1$ (to take into account the 1-based indexing of arrays in MATLAB). Thus, it calculates the row index within the current matrix. Variable j is calculated as the remainder of v divided by the number of columns $n + 1$. This gives us the column index within the current matrix. Variable d is calculated as $(v - k)/(m \times n) + 1$, which is the result of integer division between $(v - k)$ and the total number of elements in a matrix $(m \times n) + 1$ (again, to account the 1-based indexing). Thus, it calculates the index of the current matrix in the 3D array. Thus, this arithmetic approach allows for iterating through a 3D array with a single loop while calculating the correct indices for each dimension, showcasing a method to linearize multi-dimensional array access.

Matrix Operations

7

Matrix operations are fundamental mathematical operations used in various fields such as linear algebra, physics, computer science, engineering, and more [1]. Matrices are structured arrangements of numbers or symbols in rows and columns, and these operations allow us to manipulate and analyze data efficiently. They play a crucial role in solving systems of linear equations, transformations, and data analysis. In this context, matrix operations encompass a wide range of mathematical processes, including addition, subtraction, multiplication, inversion, and transposition. Each operation serves a specific purpose and can be applied to matrices of different sizes and dimensions. These operations are not only essential in theoretical mathematics but also have practical applications in computer graphics, machine learning, quantum mechanics, and many other areas. Understanding matrix operations is essential for anyone working with data, whether in scientific research, engineering, or data science. These operations provide powerful tools to manipulate, transform, and extract valuable insights from structured data, making them a cornerstone of modern mathematics and science. In the following examples, we will explore various matrix operations and their applications in more detail.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_7

7.1.1 Ex. (95) – How many 1's in matrix

```
% how many 1's in matrix.

a = [
    1, 1, 0, 0, 0;
    0, 1, 0, 0, 1;
    1, 0, 0, 1, 1;
    0, 0, 0, 0, 0;
    1, 0, 1, 0, 1
    ];

n = size(a, 1);
m = size(a, 2);
k = 0;

for i = 1:n
    for j = 1:m
        if a(i, j) == 1
            k = k + 1;
        end
    end
end

fprintf('k = %d\n', k);
```

Output:

```
k = 10
```

This example is designed to count how many times the value “1” appears in a given matrix. The code begins by defining a matrix *a* using a two-dimensional array. Arrays are defined with square brackets [], and rows are separated by semicolons “;”. The matrix from the example consists of rows and columns and contains various values, including 1’s and 0’s. Next, the code initializes three variables: *n*, *m*, and *k*. The *size* function is used to determine the dimensions of the array *a*. Thus, variable *n* is set to the number of rows in the matrix *a* (in this case, there are 5 rows). Variable *m* is set to the number of columns in the matrix *a* (in this case, there are 5 columns). Also, variable *k* is initialized to zero and will be used to count the number of 1’s in the matrix. The code then enters a nested loop structure, using *for* loops to iterate through each element of the matrix. The

outer loop, controlled using variable *i*, iterates through the rows from 1 to *n*. The inner loop, is controlled by variable *j*, that iterates through the columns from 1 to *m*. Inside the innermost part of the loop, the code checks if the current element of the matrix, accessed as *a*(*i*,*j*), is equal to 1. If it is, the *k* variable is incremented by 1. Once both loops complete their iterations, the *k* variable contains the count of 1's in the matrix. The code then prints the value of *k* to the output, which represents the total count of 1's in the matrix. Thus, the primary purpose of this code is to determine the total count of 1's in the given matrix *a*.

7.1.2 Ex. (96) – Sum all values from matrix elements

```
m = [
    [2, 4, 6];
    [3, 5, 6];
    [3, 5, 4]
    ];

r = 0;

for i = 1:size(m, 1)
    for j = 1:size(m, 2)
        r = r + m(i, j);
    end
end

disp(r);
```

Output:

```
38
```

This code defines a two-dimensional array *m*, representing a matrix with three rows and three columns. Each element of the matrix contains numeric values. The code then proceeds to calculate the sum of all the elements within this matrix and stores the result in the variable *r*. It uses nested for loops to traverse the entire matrix. The outer loop, controlled by the variable *i*, iterates through the rows of the matrix. In this specific code, it will go through three rows since *size*(*m*, 1) is 3. The inner loop, controlled by the variable *j*, iterates through the elements within each row. In this case, it iterates through the three elements within each row using *size*(*m*, 2). For each element in the matrix, the value of that element is added to the running total stored in the variable *r*. This accumulation of values continues as the loops iterate through all the elements in the matrix. Thus, the script prints out the sum *r* after the loops have processed the entire matrix. This code essentially calculates the sum of all elements in the 3×3 matrix and then displays that total.

7.1.3 Ex. (97) – Show matrix content

```
m = [
    [2, 4, 6];
    [3, 5, 6];
    [3, 5, 4]
    ];

r = "";
for i = 1:size(m, 1)
    % Use newline to add a new line.
    r = [r newline];

    for j = 1:size(m, 2)
        r = [r num2str(m(i, j)) " "];
    end
end

fprintf('%s', r);
```

Output:

```
2 4 6
3 5 6
3 5 4
```

The souce code example from above defines a two-dimensional array *m* that contains numeric values organized in rows and columns. The array *m* consists of three rows, and each row contains three numerical elements. Next, the code initializes an empty string variable *r* which will be used to accumulate the output. The code then enters a nested loop structure using for loops. The outer loop iterates through the rows of the array *m*, and the inner loop iterates through the elements within each row. For each element in the 2D array, the code appends its value to the string *r*, followed by a space, to format the output. After each row is processed in the inner loop, a *newline* character is added to the string *r* to start a new line for the next row. At the end, the *fprintf* function is called, which typically sends the string *r* to the standard console window, displaying the entire 2D array in a neatly formatted manner.

7.1.4 Ex. (98) – Multiplication involving a scalar and a matrix.

```
m = [
    [2, 4, 6];
    [3, 5, 6];
    [3, 5, 4]
    ];

s = 3; % scalar.

for i = 1:size(m, 1)
    for j = 1:size(m, 2)
        m(i, j) = s * m(i, j);
    end
end

disp(m);
```

Output:

```
6 12 18
9 15 18
9 15 12
```

Here, a two-dimensional array *m* is defined. This array contains three subarrays, and each represents a row of a matrix. The matrix *m* holds integer values arranged in a 3×3 grid. Next, a scalar variable *s* is set to the value 3. This scalar value will be used to perform scalar multiplication on each element of the matrix *m*. The code enters a nested loop structure. The outer loop iterates over the rows of the matrix *m*, and the inner loop iterates over the elements within each row. This loop structure ensures that every element in the 3×3 matrix will be accessed and processed. Within the loop, each element $m(i,j)$ is multiplied by the scalar value *s*, effectively scaling the value of that element. The result of this scalar multiplication overwrites the original value of $m(i,j)$, updating the matrix *m* with the scaled values. Next, the code ends with a *disp* statement, which displays the modified matrix *m* after the scalar multiplication.

7.1.5 Ex. (99) – Sum all values from the rows of the matrix

```
m = [
    [2, 4, 4],
    [3, 5, 6],
    [3, 5, 4]
    ];

r = zeros(1, size(m, 1));

for i = 1:size(m, 1)
    r(i) = 0;

    for j = 1:size(m, 2)
        r(i) = r(i) + m(i, j);
        % r(i) = r(i) * m(i, j);
    end
end

disp(r);
```

Output:

```
10 14 12
```

In this example code, there is an array *m* defined as a 3×3 matrix, where each element contains integer values. Additionally, there is an empty array *r* defined, which will be used to store the sum of elements in each row of the matrix *m*. The code proceeds with two nested loops. The outer loop iterates through the rows of the matrix *m*. Inside the outer loop, a variable $r(i)$ is initialized to 0 for each row *i*. This variable will store the sum of elements in row *I* of the matrix. The inner loop iterates through the elements within each row of the matrix. Within this loop, the code calculates the sum of elements in row *I* by accumulating the values of $m(i,j)$ into the $r(i)$ variable. The result is an array *r* where each element $r(i)$ represents the sum of the elements in the corresponding row of matrix *m*. A comment in the code shows an alternative for a multiplication case. Next, the code concludes with a *disp* statement to display the calculated sums for each row of the matrix *m*. This output will show the sum of elements in each row as an array *r*.

7.1.6 Ex. (100) – Sum all values from the colums of the matrix

```
m = [
    [2, 4, 4],
    [3, 5, 6],
    [3, 5, 4]
    ];

% Initialize c as a
% row vector of zeros.

c = zeros(1, size(m, 2));

for i = 1:size(m, 1)
    for j = 1:size(m, 2)
        c(j) = c(j) + m(i, j);
        % c(j) = c(j) * m(i, j);
    end
end

disp(c);
```

Output:

```
8 14 14
```

In the provided code, a two-dimensional array m is defined, which represents a 3×3 matrix. The matrix m contains integer values in its cells. Additionally, there is an empty array c declared. This array will be used to store the column-wise sum of elements from matrix m. It is worth noting that c is empty initially, and its length is determined based on the number of columns in the matrix m. The code then enters a nested loop structure. The outer loop iterates over the rows of the matrix m, and the inner loop iterates over the elements within each row. This loop structure ensures that every element in the 3×3 matrix will be accessed and processed. Within the loop, variable j is used this time to add the column values. Thus, the code adds the value of $m(i,j)$ to the corresponding element in array c, effectively performing column-wise summation. The result is stored in the c array, accumulating the sum as it iterates through the rows. Next, the above implementation ends with a *disp* statement that allows the contents of variable c to be visually observed in the *Console Window*.

7.1.7 Ex. (101) – Find max and min on columns and rows of a matrix

```
a = zeros(4, 3);

m = [
    [2, 4, 4],
```

Output:

```
C Max = [3 5 6]
```

```
    [3, 5, 6],
    [3, 5, 4]
];

for i = 1:size(m, 1)
    for j = 1:size(m, 2)

        % Initialize elements of a
        % to 10000 where they are falsy.

        a(3, ~a(3, :)) = 10000;
        a(4, ~a(4, :)) = 10000;

        % Check and update the
        % maximum and minimum values.

        if a(1, j) < m(i, j)
            a(1, j) = m(i, j);
        end
        if a(2, i) < m(i, j)
            a(2, i) = m(i, j);
        end
        if a(3, j) > m(i, j)
            a(3, j) = m(i, j);
        end
        if a(4, i) > m(i, j)
            a(4, i) = m(i, j);
        end
    end
end

fprintf('C Max = %s\n', mat2str(a(1, :)));
fprintf('R Max = %s\n', mat2str(a(2, :)));
fprintf('C Min = %s\n', mat2str(a(3, :)));
fprintf('R Min = %s\n', mat2str(a(4, :)));


% or a short version with MATLAB built-in
functions:

a = zeros(4, 3);

m = [
    [2, 4, 4],
    [3, 5, 6],
    [3, 5, 4]
    ];

% Calculate maximum and minimum
% values per column and row.

a(1, :) = max(m, [], 1);
a(2, :) = max(m, [], 2)';
```

```
R Max = [4 6 5]
C Min = [2 4 4]
R Min = [2 3 3]
```

```
a(3, :) = min(m, [], 1);
a(4, :) = min(m, [], 2)';

fprintf('C Max = %s\n', mat2str(a(1, :)));
fprintf('R Max = %s\n', mat2str(a(2, :)));
fprintf('C Min = %s\n', mat2str(a(3, :)));
fprintf('R Min = %s\n', mat2str(a(4, :)));
```

In the provided example, two 2D arrays, *m* and *a*, are defined. The *m* array represents a 3×3 matrix with integer values, and the *a* array represents a 4×3 matrix filled with initial values. The code then enters a nested loop structure using two *for-loops*. The outer loop iterates over the rows of the *m* matrix, while the inner loop iterates over the elements within each row. This nested loop structure ensures that every element in the 3×3 matrix *m* will be accessed and processed. Inside the loop, there are several conditional statements and assignments. The code checks whether specific elements in the matrix *a* are zero. If they are, it assigns the value 10,000 to these elements. This operation effectively initializes matrix *a* with high values. For instance, the outer *a*(3,:) part of the expression selects the third row of the matrix *a*. The colon ":" signifies all columns in that row. The tilde "~" is the logical NOT operator in MATLAB. It converts non-zero elements to 0 (false) and zero elements to 1 (true). Thus, ~*a*(3,:) creates a logical array where the elements are 1 (true) if the corresponding elements in the third row of a are zero, and 0 (false) otherwise. Now, the entire *a*(3,~*a*(3,:)) part of the expression uses logical indexing to select elements in the third row of a where the elements of the third row are zero. In other words, it selects all columns in the third row of a that are zero. And lastly, the *a*(3,~*a*(3,:)) = 10,000 expression assigns the value 10,000 to all those elements in the third row of *a* that were originally zero. Next, there are two sets of conditional statements. The first set (*a*(0,*j*) and *a*(1,*i*)) is designed to find the maximum value along each column and row of the *m* matrix. The second set (*a*(2,*j*) and *a*(3,*i*)) is used to find the minimum value along each column and row of the *m* matrix. The code uses conditional comparisons to update the values in the *a* matrix based on the conditions mentioned above. Next, the code outputs the results using the *fprintf* statement, which displays the following statistics: (1) "C Max" represents the maximum value for each column of the *m* matrix. (2) "R Max" represents the maximum value for each row of matrix *m*. (3) "C Min" represents the minimum value for each column of matrix *m*. (4) "R Min" represents the minimum value for each row of *m*. Note also that a second example is given here, which uses built-in MATLAB functions to find the maximum (*max* function) and minimum (*min* function) values over *a*. In this last example, for instance, *a*(1,:) = *max*(*m*, [], 1) finds the maximum value in each column of matrix *m* and assigns these maximum values to the first row of matrix *a*. The *max* function with the argument [], 1 operates along the first dimension (columns) of *m*. Thus, the second example is shorter when compared to the first, which is more native and applicable to all computer languages.

7.1.8 Ex. (102) – Multiply all values from the columns / rows and store them in array

```
m = [
    2, 4, 4;
    3, 5, 6;
    3, 5, 4
    ];

a = zeros(2, size(m, 2));

for i = 1:size(m, 1)
    for j = 1:size(m, 2)
        if a(1, j) == 0
            a(1, j) = 1;
        end
        if a(2, i) == 0
            a(2, i) = 1;
        end

        a(1, j) = a(1, j) * m(i, j);
        a(2, i) = a(2, i) * m(i, j);
    end
end

disp(['C = ', num2str(a(1,:))]);
disp(['R = ', num2str(a(2,:))]);
```

Output:

```
C = 18 100 96
R = 32 90 60
```

In this code, a two-dimensional array m is defined, just like in the previous example. This array contains three subarrays, representing a 3×3 matrix, with integer values. Additionally, two arrays a are created ($2 \times N$ matrix of zeros, where N is the number of columns in m). These arrays are intended to store the product of elements in the matrix m along rows (R) and columns (C). The code then enters a nested loop structure, similar to the previous example, where the outer loop iterates over the rows of the matrix m, and the inner loop iterates over the elements within each row. Within the loop, there is conditional logic to handle the initialization of elements in the arrays a. If the specific element in $a(1,j)$ or $a(2,i)$ does not exist, it is initialized to 1. This condition ensures that if the element is accessed for the first time, it is set to 1. Subsequently, each element in $a(1,j)$ and $a(2,i)$ is multiplied by the corresponding element from the matrix $m(i,j)$. This step calculates the product of elements along columns and rows, as the code progresses through the matrix m. Next, the code displays the product along columns as "C" and the product along rows as "R", followed by the respective arrays containing these results.

7.1.9 Ex. (103) – Sum all values from the right diagonal of the square matrix

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

[n, m] = size(a);
d = 0;

for i = 1:n
    d = d + a(i, n - i + 1);
    disp(a(i, n - i + 1));
end

disp('---');
disp(d);


% or a short version with MATLAB built-in
functions (i.e. diag and flipud):


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

% Get the second
% diagonal using diag.

s = diag(flipud(a));

% Sum the values
% in the second diagonal.

d = sum(s);

disp(flipud(s));
disp('---');
disp(d);
```

Output:

```
3
5
7
---
15
```

In this case, a two-dimensional array *a* is defined, which represents a 3×3 matrix containing integer values. The code aims to perform specific operations on this matrix and calculate a sum, denoted by the variable *d*. Initially, a variable *d* is declared and initialized with the value 0. This variable will be used to accumulate the sum of specific elements in the matrix. The code then retrieves the dimensions of the matrix *a*. It uses the *size* function to determine the number of rows (*n*) and the number of columns (*m*) in the matrix. Subsequently, the code enters a loop structure. The loop iterates over the rows of the matrix using the variable *i*, ranging from 1 to *n*. Note that *m* is not used in this example. Nonetheless, within the loop, the code calculates the column index to access elements in reverse order from right to left within each row using the expression *n-i*-1. The element *a*(*i*, *n-i* + 1) is accessed, and its value is added to the variable *d* to accumulate the sum. Additionally, the code shows a *disp* statement within the loop, meant to output each value that participates to the summation. Thus, the statement *disp*(*a*(*i*, *n-i* + 1)) displays the values of the matrix elements accessed during the iteration. At the end of the code, there is another *disp* statement that displays a line containing three hyphens "–-" followed by the value of *d*. This is intended to show the sum of the selected elements in the matrix. In the second version of this MATLAB code, *flipud*(*a*) is used to flip the matrix *a* upside down. This is necessary because *diag* extracts the main diagonal of the flipped matrix, which corresponds to the second diagonal of the original matrix. Now, *diag*(*flipud*(*a*)) extracts the values of the second diagonal. Also, *sum*(*s*) calculates the sum of the values in the second diagonal. Therefore, this code will calculate and display the sum of the values in the second diagonal of the matrix *a*.

7.1.10 Ex. (104) – Sum all values from the left diagonal of the square matrix

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

% Get the size of the matrix.
[n, ~] = size(a);

d = 0;

for i = 1:n
    d = d + a(i, i);
    disp(a(i, i));
end

disp('---');
disp(d);


% or a short version with MATLAB built-in
functions (i.e. diag and flipud):


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

% Extract the main diagonal
% elements using diag().

d = diag(a);

% Sum the diagonal
% elements using sum().

s = sum(d);

disp(d)
disp('---')
disp(s);
```

```
Output:

1
5
9
---
15
```

Above, a two-dimensional array a is defined. This array represents a 3×3 matrix, with each subarray corresponding to a row of the matrix. Additionally, a variable d is initialized with the value 0, which will be used to accumulate the sum of all elements in the matrix. The code then proceeds to determine the dimensions of the matrix. Variable n is set to the number of rows (which is 3 in this case), and m is not used because the example involves a square matrix, meaning the rows and columns are equal. Note that in MATLAB, the tilde ("~") is used as a placeholder for an output argument that is not

required (i.e. *m* valiable) or is supposed to be ignored. When a function returns multiple outputs, but only some of them are used, "~" can be used to discard outputs that are not of interest. Next, the loop, controlled by variable *i*, iterates over the columns of the matrix. Within this loop, the code calculates the sum of the elements by adding the value of each element *a*(*i*,*i*) to the accumulator *d*. The value of the element *a*(*i*,*i*) is also printed in the console using the *disp* function. After processing all elements of the matrix, the script ends by printing a separator line ("–-") followed by the accumulated sum *d* which represents the sum of all elements in the left diagonal of the matrix.

7.1.11 Ex. (105) – Sum all values from the left and right diagonal of a square matrix

```
a = [
    1, 2, 3, 4;
    5, 6, 7, 8;
    9, 0, 1, 2;
    3, 4, 5, 6
    ];

ld = 0; % left diagonal.
rd = 0; % right diagonal.

% rows (n), columns (m).
[n, m] = size(a);

for j = 1:m
    ld = ld + a(j, j);
    rd = rd + a(j, m - j + 1);
end

fprintf('L=%d|R=%d\n', ld, rd);
```

Output:

```
L=14|R=14
```

In this Matlab code, we are working with a two-dimensional array *a*, which represents a 4×4 grid. This array contains four subarrays, each of which corresponds to a row of the matrix. Additionally, two variables are defined: *ld* (for left diagonal) and *rd* (for right diagonal), both initially set to 0. These variables will be used to accumulate the sums of the values along the left and right diagonals of the matrix, respectively. The code proceeds with setting the variables *n* and *m*, where *n* represents the number of rows in the matrix (which is 4 in this case), and *m* represents the number of columns (also 4). However, variable *m* is ignored because the example involves a square matrix, meaning the rows and columns are equal. Moreover, a counter variable *j* is also initialized to zero. Next, a *for-loop* structure is used. Within the loop, two actions are performed. The *ld* (left diagonal) variable is updated by adding the value of the current element *a*(*j*,*j*). This step accumulates the sum of values along the left diagonal of the matrix. The *rd* (right diagonal) is updated by adding the value of the element at *a*(*j*,*m*-*j* + 1). This step accumulates the sum of values along the right diagonal of the matrix. The *m*-*j* + 1 expression accesses the elements in reverse order along each row, effectively giving the

values on the right diagonal. The code concludes with a *fprintf* statement that displays the sums of the left and right diagonals, stored in the *ld* and *rd* variables, respectively, and are presented in the following format: "L = *ld*|R = *rd*."

7.1.12 Ex. (106) – Sum all values from the left and right diagonal by using conditions

```
a = [
    1, 2, 3, 4;
    5, 6, 7, 8;
    9, 0, 1, 2;
    3, 4, 5, 6
    ];

% d(1) for principal diagonal.
% d(2) for secondary diagonal.

d = zeros(1, 2);

n = size(a, 1);
m = size(a, 2);

for i = 1:n
    for j = 1:m
        if i == j
            d(1) = d(1) + a(i, j);
        end
        if (i + j) == (n + 1)
            d(2) = d(2) + a(i, j);
        end
    end
end

fprintf('L=%d|R=%d\n', d(1), d(2));
```

Output:

```
L=14|R=14
```

The previous code contained an optimal way to sum the values found on the diagonals of a matrix, without using conditions. However, what if conditional branching is used, what would the example normally look like? In this new version, we are working with a two-dimensional array a, which represents a square matrix with dimensions 4×4. The array d is initialized as an empty array, and it is used to store the sums of the elements along the principal diagonal and the secondary diagonal of the matrix a. The principal diagonal of a square matrix consists of the elements where the row and column indices are the same ($i = j$), and the secondary diagonal consists of the elements where the sum of the row and column indices is equal to one more than the number of rows ($i + j = n + 1$). The variable n is set to the number of rows in the matrix a, and m is set to the number of columns in the matrix a. In this case, both n and m are set to 4 since the matrix a is 4×4. The code enters a nested loop structure. The outer loop iterates over the rows of the matrix a, and the inner loop iterates over the columns of the matrix a. Within this loop, there are conditional statements that check if the current element is part of the principal

diagonal or the secondary diagonal. If i (the current row index) is equal to j (the current column index), the element belongs to the principal diagonal, and its value is added to $d(0)$. This operation accumulates the sum of the elements on the principal diagonal. If $(i + j)$ is equal to $(n + 1)$, the element belongs to the secondary diagonal, and its value is added to $d(1)$. This operation accumulates the sum of the elements on the secondary diagonal. Next, the code prints a string that represents the results. It displays the sum of the elements on the principal diagonal (L) and the sum of the elements on the secondary diagonal (R) in the format "L = value|R = value."

7.1.13 Ex. (107) – Show bottom – left part of the matrix

```
a = [
    1, 2, 3, 4;
    5, 6, 7, 8;
    9, 0, 1, 2;
    3, 4, 5, 6
    ];

n = size(a, 1);  % rows.
m = size(a, 2);  % columns.

d = zeros(n, m);

r = '';

for i = 1:n
    for j = 1:i
        d(i, j) = a(i, j);
        r = [r, num2str(d(i, j)), ' '];
    end
    r = [r, '\n'];
end

fprintf(r);
```

Output:

```
1
5 6
9 0 1
3 4 5 6
```

```
%{
1 - - -
5 6 - -
9 0 1 -
3 4 5 6
%}
```

Other operations may involve selective parts of the matrix. Thus the above MATLAB code defines a 2D array a containing a grid of numbers. It then initializes an empty array d and extracts the dimensions of the array a, storing the number of rows in n and the number of columns in m. Next, a string variable r is also declared and initialized as an empty string. The code proceeds to loop through the rows of the array a by using a *for-loop*, with the index variable i. Within this loop, another nested loop runs with index variable j to iterate through the columns ($j = 1{:}i$). During each iteration, the value from the array a at the i row and j column is stored in the d array at the same position. Additionally, the value is appended to the r string, separated by a space. After each row of d is processed, a newline character is appended to the r string to separate the rows. Next, the code prints the contents of the r string using the *fprintf* function. In short, this

code generates a new 2D array *d* that contains a triangular subset of the original array *a*, and *r* holds a string representation of this subset.

7.1.14 Ex. (108) – Show bottom – left part of the matrix and flip horizontal

```
a = [
    1, 2, 3, 4;
    5, 6, 7, 8;
    9, 0, 1, 2;
    3, 4, 5, 6
    ];

n = size(a, 1);
m = size(a, 2);

d = zeros(n, m);

r = '';

for i = 1:n
    for j = 1:i
        d(i, j) = a(i, j);
        r = [r, num2str(d(i, i-j+1)), ' '];
    end
    r = [r, '\n'];
end

fprintf(r);
```

Output:

```
1
6 5
1 0 9
6 5 4 3
```

```
%{
1 - - -
5 6 - -
9 0 1 -
3 4 5 6
%}
```

In this example, we start with a 2D array *a* that contains a grid of numbers. The code then initializes an empty array *d* and extracts the dimensions of the array *a*, storing the number of rows in *n* and the number of columns in *m*. The variable *r* is declared and initialized as an empty string. The code uses a *for-loop* to iterate through the rows of the *a* array using the index variable *i*. Within this loop, there is a nested loop that iterates through the columns. During each iteration, the value from the array *a* at the *i* row and (*i-j + 1*) column is stored in the *d* array at the same position. The value is also appended to the *r* string, separated by a space. After each row of *d* is processed, a newline character is appended to the *r* string to separate the rows. As in the previous example, the code prints the contents of the *r* string using the *fprintf* function. Please note that this source code example generates a new 2D array *d* that contains a triangular subset of the original array *a*, but this time the values are selected from the opposite diagonal of *a*, and *r* holds a string representation of this subset.

7.1.15 Ex. (109) – Show top – right part of the matrix

```
a = [1, 2, 3, 4;
     5, 6, 7, 8;
     9, 0, 1, 2;
     3, 4, 5, 6];

n = size(a, 1);
m = size(a, 2);

r = '';

for i = 1:n
    for j = i:m
        if j == i
            d(i, j) = a(i, j);
        end
        r = [r, num2str(a(i, j)), ' '];
    end
    r = [r, '\n'];
end

fprintf(r);
```

Output:

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

```
%{
1 2 3 4
- 6 7 8
- - 1 2
- - - 6
}%
```

This MATLAB example operates on a 2D array *a* and produces a different result compared to the previous code. The code begins by defining a 2D array *a* with four rows and four columns. It also initializes an empty array *d*, and it determines the number of rows *n* and columns *m* in the array *a*. Just as before, a string variable *r* is declared and initialized as an empty string. The code then uses a nested loop structure. The outer loop, controlled by the index variable *i*, iterates through the rows of the array *a* from 1 to *n*. Within the outer loop, a nested loop (controlled by the index variable *j*) starts from *i* and goes up to the last column *m*. During each iteration, the value from the array *a* at the *i* row and *j* column is stored in the *d* array at the same position. Additionally, the value is appended to the *r* string, separated by a space. After each column is processed within a row, a newline character is added to the *r* string to separate the rows. In the next stage, the code prints the contents of the *r* string. Thus, this code generates a new 2D array *d* containing a right triangular subset of the original array *a*, and *r* holds a string representation of this subset.

7.1.16 Ex. (110) – Show top – right part of the matrix and flip 90 degrees left

```
a = [
    1, 2, 3, 4;
    5, 6, 7, 8;
    9, 0, 1, 2;
    3, 4, 5, 6
    ];

[n, m] = size(a);
d = cell(n, 1);
r = '';

for i = 1:n
    d{i} = zeros(1, i);
    for j = 1:i
        d{i}(j) = a(i-j+1, i);
        r = [r, num2str(d{i}(j)), ' '];
    end
    r = [r, '\n'];
end

fprintf(r);
```

Output:

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

```
%{
1 2 3 4
- 6 7 8
- - 1 2
- - - 6
}%
```

In this code example all variables are declared just as in the previous example. The code proceeds to loop through the rows of the array using a *for-loop*, just as done previously. However, the key difference is in how the values are obtained. In this case, the values from the array *a* are taken from different positions. Specifically, the value from *a* at the row *i-j + 1* and column *i*, is stored in array *d* at the same position. The rest of the code is similar to the previous example. The values are appended to the *r* string with a space between them, and a newline character is added after each row is processed. Just like the previous example, the code prints the contents of the *r* string using the *fprintf* function. As perhaps it can be observed, this code generates a new 2D array *d* that contains a triangular subset of the original array *a*, but this time the values are selected differently, resulting in a different subset, stored in variable *r*.

7.1.17 Ex. (111) – Show top – right, flip 90 degrees left and flip horizontally

```
a = [
    1, 2, 3, 4;
    5, 6, 7, 8;
    9, 0, 1, 2;
    3, 4, 5, 6
    ];

d = zeros(n, m);
[n, m] = size(a);

% Iterate to create the
% lower triangular matrix.

for i = 1:n
    for j = 1:i
        d(i, j) = a(j, i);
    end
end

for i = 1:n
    for j = 1:i
        fprintf('%d ', d(i, j));
    end
    fprintf('\n');
end
```

```
Output:

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

```
%{
1 2 3 4
- 6 7 8
- - 1 2
- - - 6

}%
```

This MATLAB code is similar to the previous one, but it has a different purpose. It starts by defining a 2D array a that contains a grid of numbers. Then, it initializes an array d and retrieves the number of rows in n and the number of columns in m from the array a. In order to change the scenery a little bit, variable r is not used in this implementation. The code proceeds to loop through the rows of the array a using a *for-loop* with the index variable i (i.e. $i = 1{:}n$). Within this loop, there is another nested loop that iterates through the columns with the index variable j (i.e. $j = 1{:}i$). During each iteration, the value from the array a at the j row and i column (note again the reversed indices) is stored in array d. Just like in the previous code, the *fprintf* function is used to display the contents of variable d by using a second nested *for-loop*. In conclusion, this code generates a new 2D array d that contains a transposed version of the original array a, and r holds a string representation of this transposed array.

7.1.18 Ex. (112) – Secondary diagonal scan (right)

```
a = [1, 2, 3, 4;
     5, 6, 7, 8;
     9, 0, 1, 2;
     3, 4, 5, 6];

n = size(a, 1);
m = size(a, 2);

% Using cell array to hold
% the varying length vectors.

d = cell(n, 1);
r = '';

for i = 1:n
    d{i} = zeros(1, i);
    for j = 1:i
        d{i}(j) = a(j, i-j+1);
        r = [r, num2str(d{i}(j)), ' '];
    end
    r = [r, '\n'];
end

fprintf(r);
```

Output:

```
1
2 5
3 6 9
4 7 0 3
```

```
%{
1 2 3 4
5 6 7 -
9 0 - -
3 - - -
}%
```

This code generates a new 2D array d that contains a transformed subset of the original a array, and r holds a string representation of this transformed subset. It operates on a 2D array a and performs a different transformation compared to the previous code. The key difference here is that the value from a is not extracted directly based on row and column indices as in the previous code. Instead, it uses $a(j, i\text{-}j + 1)$ to obtain values from a by a different pattern. For each iteration, the value from a obtained using $a(j, i\text{-}j + 1)$ is stored in the d array at the same position ($d\{i\}(j)$), and this value is appended to the r string, separated by a space. A newline character is also appended to r after each row of d is processed. Next, the *fprintf* prints variable r. One important note taken from this example is that in MATLAB, accessing elements within arrays or cell arrays is dictated by the type of brackets used. The "{}" is specifically for cell arrays, allowing access to the content within a cell. For instance, if d is a cell array, using $d\{1\}$ accesses the content of the first cell. In contrast, "()" is used for accessing elements in a standard array or matrix. For example, $d(1,2)$ retrieves the element in the first row and second column of a numeric array. When combining these two, as in $d\{\}()$, it implies a two-step access where the first access is made to the content of a cell array using "{}", and then one can index into the content, which is itself an array, using "()". This is useful when the cells of a cell array contain arrays.

7.1.19 Ex. (113) – Secondary diagonal scan (left)

```
a = [
    1, 2, 3, 4;
    5, 6, 7, 8;
    9, 0, 1, 2;
    3, 4, 5, 6
    ];

n = size(a, 1);
m = size(a, 2);
d = zeros(n, n);

r = '';
for i = 1:n
    for j = 1:i
        d(i, j) = a(i-j+1, n-j+1);
        r = [r, num2str(d(i, j)), ' '];
    end
    r = [r, '\n'];
end

fprintf(r);
```

Output:

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

```
%{
1 2 3 4
- 6 7 8
- - 1 2
- - - 6
}%
```

Here, the example generates a new matrix *d* that contains a triangular subset of the original array *a*, following a different pattern compared to the previous code. The code extracts the dimensions of the matrix *a*, storing the number of rows and columns in *n* and *m*, respectively. Also, the code initializes a matrix *d* with zero values. An empty string *r* acting as an accumulator is declared and initialized. The code then enters a loop that iterates through the rows of array *a* using a *for-loop* with the index variable *i*. Within this loop, there is another nested loop with the index variable *j* that iterates through the columns. During each iteration, the code accesses elements from array *a* in a pattern where the row index is based on *i* and *j*, and the column index is derived from *n* and *j*. The value from array *a* at this calculated position is then stored in array *d* at the corresponding *i* and *j* position. The value is also appended to the content of the *r* variable with a space separator. Once the processing of each row of *d* is made, a newline character is added to the *r* string to separate the rows.

7.1.20 Ex. (114) – Show bottom – right and flip horizontally

```
a = [
    1, 2, 3, 4;
    5, 6, 7, 8;
    9, 0, 1, 2;
    3, 4, 5, 6
    ];

d = [];
r = '';
[n, m] = size(a);

for i = 1:n
    for j = 1:i
        d(i, j) = a(i, m-j+1);
        r = [r, num2str(d(i, j)), ' '];
    end
    r = [r, '\n'];
end

fprintf(r);
```

Output:

```
4
8 7
2 1 0
6 5 4 3
```

```
%{
- - - 4
- - 7 8
- 0 1 2
3 4 5 6
}%
```

This code generates a new 2D array *d* that contains another type of a triangular subset of the original array *a* with the elements reversed within each row. The MATLAB example from above is similar to the previous one, but with a slight modification in the way it populates the array *d* from array *a*. The code then enters a nested loop structure using a *for-loop* to iterate through the rows and columns. During each iteration, the value from the array *a* at the *i* row and *m-j* + 1 column is stored in array *d* at the same position. The *m-j* + 1 index is used to select elements in reverse order within each row of array *a*, effectively reversing the order of elements in each row. Please notice that variable *m* is used, however, since the matrix *a* is a square matrix, both *m* and *n* are interchangeable (i.e. *n-j* + 1 would have the same effect). Therefore, the implementation essentially extracts a left flip of the triangular subset represented by the lower right part of the matrix *a*. Moreover, the purpose of these examples of triangular subsets was to show the mechanical interaction that can exist between the *for-loop* counters and their upper bounds.

7.1.21 Ex. (115) – Matrix flip vertical

```
% flip vertical

a = [
    1, 2, 3, 4;
    5, 6, 7, 8;
    9, 0, 1, 2;
    3, 4, 5, 6
    ];

r = '';
[n, m] = size(a);
d = zeros(n, m);

for i = 1:n
    for j = 1:m
        d(i, j) = a(n-i+1, j);
        r = [r, num2str(d(i, j)), ' '];
    end
    r = [r, '\n'];
end

fprintf(r);
```

Output:

```
3 4 5 6
9 0 1 2
5 6 7 8
1 2 3 4
```

The above code flips the original array a vertically to create a new 2D array d, and r holds a string representation of this flipped array. The implementation, like before, begins with a 2D array a containing a grid of numbers. It determines the dimensions of the array a and then initializes an array d. The number of rows is stored in n, and the number of columns is stored in m. The code proceeds to loop through the rows of array a using a *for-loop*, with the index variable i. Within this loop, there is another nested loop that iterates through the columns using the index variable j. During each iteration, the code assigns a value from array a to array d, effectively flipping the rows vertically. It takes the i-th row from a in reverse order (n-i + 1) and assigns it to the i-th row of d. Additionally, the value is appended to the r string, separated by a space. After processing each row of d, a newline character is appended to the r string to separate the rows. The code then shows the contents of the r variable.

7.1.22 Ex. (116) – Matrix flip horizontal

```
% flip horisontal.

a = [
    1, 2, 3, 4;
    5, 6, 7, 8;
    9, 0, 1, 2;
    3, 4, 5, 6
    ];

[n, m] = size(a);
d = zeros(n, m);

for i = 1:n
    for j = 1:m
        d(i, j) = a(i, m-j+1);
    end
end

disp(d);
```

Output:

```
4 3 2 1
8 7 6 5
2 1 0 9
6 5 4 3
```

This code flips an array a horizontally and stores the flipped version in the array d, with r containing a string representation of this flipped array. The code starts with the definition of a 2D array a, which contains a grid of numbers. It initializes an array d and extracts the number of rows (n) and columns (m) of the array a. The code then proceeds to iterate through the rows of the array a using a *for-loop* with the index variable i. Within this loop, there is another nested loop that iterates through the columns using the index variable j. During each iteration, the value from the a array at the i row and m-j + 1 column is stored in the d array at the same position. This effectively flips the columns of array a horizontally. Additionally, the value is appended to the r string, separated by a space. After processing each row of d, a newline character is appended to the r string to separate the rows. The content of the variable is then printed in the output by using the *disp* function.

7.1.23 Ex. (117) – Sum values from elements of a matrix based on a map

```
a = [2, 4, 6;
     3, 5, 6;
     3, 5, 4];

b = [0, 0, 1;
     1, 1, 0;
     0, 0, 1];

[n, m] = size(a);
r = 0;

for i = 1:n
    for j = 1:m
        if b(i,j) == 1
            r = r + a(i,j);
        end
    end
end

disp(r);
```

Output:

```
18
```

This source code example calculates the sum of the values in matrix *a* where the corresponding elements in the *b* matrix are equal to 1 and stores the result in the *r* variable. Above, the code defines two 2D arrays, *a* and *b*, which represent matrices of numbers. It then calculates the dimensions of the array *a*, storing the number of rows in *n* and the number of columns in *m*. A variable *r* is declared and initialized to zero. The code proceeds to loop through the rows of the array *a* using a *for-loop* with the index variable *i*. Inside this loop, there is a nested loop that iterates through the columns using the index variable *j*. During each iteration, the code checks if the value of *b* at the *I* row and *j* column is equal to 1. If it is, the corresponding value from the array *a* at the same position is added to the *r* variable. At the end of it, the code prints the value of *r* using the *disp* function. Note that an alternative case is also presented in the comments to show the possibility of using different types of operations.

7.1.24 Ex. (118) – Add two matrices into a third

```
a = [2, 4, 6; 3, 5, 6; 3, 5, 4];
b = [2, 4, 6; 3, 5, 6; 3, 5, 4];

c = zeros(size(a));

for i = 1:size(a,1)
    for j = 1:size(a,2)
        c(i, j) = a(i, j) + b(i, j);

        % Alternatively for subtraction:
        % c(i, j) = a(i, j) - b(i, j);

    end
end

disp(c);
```

Output:

```
4 8 12
6 10 12
6 10 8
```

```
%{ for substraction:
0 0 0
0 0 0
0 0 0
%}
```

Here, the code begins by defining two 2D arrays, *a* and *b*, in a linear manner, each containing a grid of numbers. It then initializes an array *c* with zero values. The code uses a nested loop to iterate through the rows and columns of *a* and *b*. The outer loop iterates through the rows of *a* and *b* using the index variable *i*. The inner loop iterates through the columns of *a* and *b* using the index variable *j*. In each iteration, the code calculates the sum of the corresponding elements in *a* and *b* and stores the result in the *c* array at the same position. The code then prints the contents of the *c* array by using the *disp* function. Thus, this code calculates the element-wise sum of the *a* and *b* arrays and stores the result in the *c* array.

7.1.25 Ex. (119) – Matrix multiplication with three for loops

```
a = [2, 4, 6; 3, 5, 6; 3, 5, 4];
b = [2, 4, 6; 3, 5, 6; 3, 5, 4];

c = zeros(size(a));
r = "";

for i = 1:size(a, 1)
    r = r + newline;
    for j = 1:size(a, 2)
        for k = 1:size(a, 2)
            c(i, j) = c(i, j) + ...
                      a(k, j) * b(i, k);
        end
        r = r + c(i, j) + " ";
    end
end

disp(r);

% Alternative multiplication
% specific to MATLAB:

%{

a = [2, 4, 6; 3, 5, 6; 3, 5, 4];
b = [2, 4, 6; 3, 5, 6; 3, 5, 4];

c = a * b

%}
```

Output:

```
34 58 60
39 67 72
33 57 64
```

In this code implementation, there are three 2D arrays: *a*, *b*, and *c*. The code performs matrix multiplication between *a* and *b* and stores the result in *c*. A string variable *r* is initialized and will be used to store a string representation of the *c* matrix. The code uses nested *for-loops* to iterate through the rows and columns of the *a* and *b* matrices. The outer loop, controlled by the variable *i*, iterates over the rows of *a* and *b*. The innermost loop, controlled by variable *k*, is responsible for performing the matrix multiplication. It calculates the value of *c*(*i*,*j*) by summing up the products of elements from *a* and *b* matrices. The result is stored in the *c* matrix at the corresponding position, and the calculated value is appended to the *r* string followed by a space. Once all the iterations are completed, the *r* string contains a string representation of the resulting *c* matrix, which is essentially the product of matrices *a* and *b*. Note that, MATLAB is a matrix specific language and an alternative multiplication is c = a * b.

7.1.26 Ex. (120) – Matrix multiplication with two for loops

```
a = [2, 4, 6;
     3, 5, 6;
     3, 5, 4];

b = [2, 4, 6;
     3, 5, 6;
     3, 5, 4];

i = 1; j = 1; r = "";
c = cell(1, 1);

n1 = size(a, 1);
n2 = size(a, 2);
q = n1 * n2;

c{1} = [];

for v = 1:q

    j = mod(v - 1, n2) + 1;
    if (j == 1 && v ~= 1 && i <= n1 && v ~= q)
        i = i + 1;
        c{i} = [];
        r = r + newline;
    end

    c{i}(j) = 0;

    for k = 1:size(a, 2)
        c{i}(j) = c{i}(j) + a(k, j) * b(i, k);
    end

    r = r + c{i}(j) + " ";

end

disp(r);


% or a short version with
% MATLAB built-in functions:


a = [
    2, 4, 6;
    3, 5, 6;
    3, 5, 4
    ];

b = [
    2, 4, 6;
    3, 5, 6;
    3, 5, 4
    ];

c = a * b';

disp(c);
```

```
Output:

34 58 60
39 67 72
33 57 64
```

```
     3, 5, 4];

i = 1; j = 1; r = "";
c = cell(1, 1);

n1 = size(a, 1);
n2 = size(a, 2);
q = n1 * n2;

c{1} = [];

for v = 1:q

    j = mod(v - 1, n2) + 1;
    if (j == 1 && v ~= 1 && i <= n1 && v ~= q)
        i = i + 1;
        c{i} = [];
        r = r + newline;
    end

    c{i}(j) = 0;

    for k = 1:size(a, 2)
        c{i}(j) = c{i}(j) + a(k, j) * b(i, k);
    end

    r = r + c{i}(j) + " ";

end

disp(r);


% or a short version with
% MATLAB built-in functions:


a = [
    2, 4, 6;
    3, 5, 6;
    3, 5, 4
    ];

b = [
    2, 4, 6;
    3, 5, 6;
    3, 5, 4
    ];

c = a * b';

disp(c);
```

The example above performs matrix multiplication between two 2D arrays *a* and *b*. It calculates the product and stores the result in the *c* array. The *r* string is used to format

and print the result. Two 2D arrays, *a* and *b*, are defined, each containing three rows and three columns. Index variables *i* and *j* are initialized to 1. The *r* string is initialized as an empty string, and an array *c* is defined to store the main result. The dimensions of *a* are determined with *n1* representing the number of rows and *n2* representing the number of columns. A loop is used to iterate from 1 to the total number of elements in *a* (*n1* * *n2*), denoted as *q*. The loop variable is *v*. Within the loop, the code calculates the current column index *j* as *mod*(v-1, *n2*) + 1. If *j* is 1, a new row is started by incrementing *i*, and a new empty sub-array is added to *c*. A newline character is also appended to the *r* string to separate rows. The code sets *c*(*i*,*j*) to 0 to prepare it for storing the matrix multiplication result. A nested loop iterates through the elements of *a* and calculates the dot product of the *j*-th column of *a* and the *i*-th row of *b*. The result is stored in *c*(*i*,*j*). The calculated result, *c*(*i*,*j*), is appended to the *r* string followed by a space to separate values. After the loop completes, the *r* string contains the formatted result of the matrix multiplication. The code prints the contents of the *r* string in the console by using the *disp* function. As in the previous code, a MATLAB-specific multiplication is also shown in a second example.

7.1.27 Ex. (121) – Multiply specific elements of two matrices based on a map

```
a = [2, 4, 6;
     3, 5, 6;
     3, 5, 4];

b = [0, 1, 0;
     1, 1, 1;
     0, 1, 0];

c = [2, 4, 6;
     3, 5, 6;
     3, 5, 4];

[n, m] = size(a);
r = '';

for i = 1:n
    r = [r '\n'];
    for j = 1:m
        if b(i,j) == 1
            c(i,j) = a(i,j) * c(i,j);
        end
        r = [r num2str(c(i,j)) ' '];
    end
end

fprintf(r);
```

Output:

```
2 16 6
9 25 36
3 25 4
```

```
% or a short version with
% MATLAB built-in functions:


a = [2, 4, 6; 3, 5, 6; 3, 5, 4];
b = [0, 1, 0; 1, 1, 1; 0, 1, 0];
c = [2, 4, 6; 3, 5, 6; 3, 5, 4];

% Perform element-wise
% multiplication where b is 1

c(b == 1) = a(b == 1) .* c(b == 1);

disp(c);
```

This code performs element-wise multiplication on the *c* array using values from the array *a* based on the *b* array values, and it generates a string *r* containing the updated *c* array with spaces and newlines to format the output. Initially, three 2D arrays *a*, *b*, and *c* are defined. The *a* and *b* arrays contain numeric values, while the *c* array is initially identical to the *a* array. The code initializes variables *n* and *m* with the number of rows and columns in the array *a*. A string variable *r* is also initialized as an empty string. The code then enters a nested loop structure, iterating over the rows (index *i*) and columns (index *j*) of the *a* and *b* arrays. Inside the loop, it checks if the value of *b* at the current position (*i*, *j*) is equal to 1. If it is, it multiplies the corresponding value in the *c* array by the value in the array *a* at the same position. Regardless of whether the multiplication occurs or not, the value from the *c* array at the current position is appended to the *r* string, separated by a space. After all columns in a row are processed, a newline character is added to the *r* string to separate the rows. At the end, the code prints the contents of the *r* variable. Also note a MATLAB-specific shortcut for implementing the native code above. In this new version, the element-wise multiplication ".*" is directly applied to the elements of *c* where *b* equals 1. Thus, there is no need for loops or manual string construction, because MATLAB can directly show the modified matrix c.

7.1.28 Ex. (122) – Different operations based on maps

```
% perform different operations between the values of
% the homologous elements of two arrays based on
% a map/model (third array).

a = [
    2, 2, 2, 2, 2;
    2, 2, 2, 2, 2;
    2, 2, 2, 2, 2;
    2, 2, 2, 2, 2;
    2, 2, 2, 2, 2
];
```

Output:

```
5 5 6 6 6
1 5 6 6 5
5 1 6 5 5
6 6 6 # 6
1 6 2 6 2
```

```
b = [
    1, 1, 0, 0, 0;
    3, 1, 0, 0, 1;
    1, 3, 0, 1, 1;
    0, 0, 0, 7, 0;
    3, 0, 4, 0, 9
];

c = [
    3, 3, 3, 3, 3;
    3, 3, 3, 3, 3;
    3, 3, 3, 3, 3;
    3, 3, 3, 3, 3;
    3, 3, 3, 3, 1
];

% Convert c to a cell
% array to handle strings.

c = num2cell(c);

[n, m] = size(a);

for i = 1:n
    for j = 1:m

        if b(i, j) == 0
            c{i, j} = a(i, j) * c{i, j};
        elseif b(i, j) == 1
            c{i, j} = a(i, j) + c{i, j};
        elseif b(i, j) == 2
            c{i, j} = a(i, j) - c{i, j};
        elseif b(i, j) == 3
            c{i, j} = c{i, j} - a(i, j);
        elseif b(i, j) == 4
            c{i, j} = mod(a(i, j), c{i, j});
        elseif b(i, j) == 5
            c{i, j} = a(i, j) / c{i, j};
        elseif b(i, j) == 6
            c{i, j} = a(i, j);
        elseif b(i, j) == 7
            c{i, j} = '#';
        elseif b(i, j) == 8
            % do stuff
        elseif b(i, j) == 9
            if c{i, j} <= a(i, j)
                c{i, j} = a(i, j);
            end
        end
    end
end

disp(c);
```

This code operates on two matrices a and c based on a specified map/pattern found in matrix b, and then stores the results in matrix c. It then prints the resulting c matrix as

a string representation. The code begins by defining three matrices *a*, *b*, and *c*. Variables *a* and *b* contain numeric values, while *c* contains initial values, which are all 3 s except for the bottom-right element, which is 1. The source code determines the dimensions of the matrices, with *n* representing the number of rows and *m* representing the number of columns. Like many times in the examples from above, it also initializes an empty string *r* for storing the final output. The code enters a nested loop with two for loops, one for iterating over rows (indexed by *i*) and the other for columns (indexed by *j*). Within the nested loop, it checks the value of *b*(*i*,*j*) to determine which operation to perform on the corresponding elements of *a* and *c* matrices. The possible operations are: (1) If *b*(*i*,*j*) is 0, it multiplies the elements of *a* and *c*. (2) If *b*(*i*,*j*) is 1, it adds the elements of *a* and *c*. (3) If *b*(*i*,*j*) is 2, it subtracts the elements of *c* from *a*. (4) If *b*(*i*,*j*) is 3, it subtracts the elements of *a* from *c*. (5) If *b*(*i*,*j*) is 4, it takes the modulus of the elements of *a* and *c*. (6) If *b*(*i*,*j*) is 5, it performs division on the elements of *a* and *c*. (7) If *b*(*i*,*j*) is 6, it sets the element of *c* to the corresponding element of *a*. (8) If *b*(*i*,*j*) is 7, it sets the element of *c* to the character "#". Notice that a cell array (i.e., *c*{*i*, *j*}) is used to handle the "#" character. This approach allows to mix characters and numbers in the same array. (9) If *b*(*i*,*j*) is 9, it checks if the element of *c* is less than or equal to the element of *a* and, if so, sets the element of *c* to the element of *a*. After each operation, the resulting element in the *c* matrix is appended to the *r* string with a space. Also, a newline character is added after each row is processed. At the end, the code prints the content of variable *r*.

7.1.29 Ex. (123) – Different operations based on maps (SMC)

```
a = [2, 2, 2, 2, 2;
     2, 2, 2, 2, 2;
     2, 2, 2, 2, 2;
     2, 2, 2, 2, 2;
     2, 2, 2, 2, 2];

b = [1, 1, 0, 0, 0;
     3, 1, 0, 0, 1;
     1, 3, 0, 1, 1;
     0, 0, 0, 7, 0;
     3, 0, 4, 0, 9];

c = [3, 3, 3, 3, 3;
     3, 3, 3, 3, 3;
     3, 3, 3, 3, 3;
     3, 3, 3, 3, 3;
     3, 3, 3, 3, 1];

[n, m] = size(a);
```

Output:

```
5 5 6 6 6
1 5 6 6 5
5 1 6 5 5
6 6 6 # 6
1 6 2 6 2
```

```
for i = 1:n
    for j = 1:m

        switch b(i,j)
            case 0
                c(i,j) = a(i,j) * c(i,j);
            case 1
                c(i,j) = a(i,j) + c(i,j);
            case 2
                c(i,j) = a(i,j) - c(i,j);
            case 3
                c(i,j) = c(i,j) - a(i,j);
            case 4
                c(i,j) = mod(a(i,j), c(i,j));
            case 5
                c(i,j) = a(i,j) / c(i,j);
            case 6
                c(i,j) = a(i,j);
            case 7
                c(i,j) = NaN;
            case 9
                if c(i,j) <= a(i,j)
                    c(i,j) = a(i,j);
                end
        end
    end
end

SMC(c);

function SMC(m)
    [rows, cols] = size(m);
    for i = 1:rows
        for j = 1:cols
            if isnan(m(i,j))
                fprintf(' # ');
            else
                fprintf(' %d ', m(i,j));
            end
        end
        fprintf('\n');
    end
end
```

This code example is basically the same as the previous one. It involves operations between the values of two matrices *a* and *c* based on a pattern specified by the *b* matrix. However, the result is stored in the *c* matrix, and the *SMC* function is defined to convert and display the *c* matrix. That is, unlike the previous case when the values from the elements of the *c* matrix were formatted as strings and then accumulated in the *r* variable. In order to better understand functions please read the next chapter. To simplify the process, this example also uses numeric arrays. In the case of the "#"character (i.e., case 8 of the previous example), the array *c* is assigned with a *NaN* value, and later in the *SMC* function this *NaN* value is printed as the "#" character.

7.1.30 Ex. (124) – Nested arrays

```
A = {'a', 'b', 'c'};
B = {'d', 'e', 'f'};
C = {'g', 'h', 'i'};

D = {A, B, C};
E = {B, C, A};
F = {C, B, A};

G = {D, E, F};

disp(A{1});
disp(D{1});
disp([G{1}{1},G{1}{2},G{1}{3}]);
```

Output:

```
a
a b c
a b c d e f g h i
```

This code involves the creation of several arrays and the use of nested arrays. Three arrays *A*, *B*, and *C* are declared, each containing three string values. Then, three arrays *D*, *E*, and *F* are defined. These arrays are comprised of references to the previous arrays *A*, *B*, and *C*. Thus, *D* contains {*A*, *B*, *C*}, *E* contains {*B*, *C*, *A*}, and *F* contains {*C*, *B*, *A*}. At last, an array *G* is created, which contains references to the arrays *D*, *E*, and *F*. Then, the code prints elements from these arrays using the *disp* function: (1) *disp*(*A*{1}) will print the first element of array *A*, which is "a". (2) *disp*(*D*{1}) will print the first element of array *D*, which is the reference to array *A*. (3) *disp*([*G*{1}{1}, *G*{1}{2}, *G*{1}{3}]) will print three values from the first element of array *G*, which is the reference to array *D*. Namely, *G*{1} accesses the first element of *G*, which is *D*, whereas *G*{1}{1} accesses the first element of *D*, which is *A*, *G*{1}{2} accesses the second element of *D*, which is *B*, and *G*{1}{3} accesses the third element of *D*, which is *C*. Since *A*, *B*, and *C* are cell arrays of strings, the expression [*G*{1}{1}, *G*{1}{2}, *G*{1}{3}] concatenates the contents of these cell arrays into a single cell array. Therefore, it combines the elements of *A*({'a', 'b', 'c'}), *B*({'d', 'e', 'f'}), and *C*({'g', 'h', 'i'}) into one.

Functions

8

Functions are a fundamental concept in computer programming and serve as a building block for organizing and reusing code [1]. At their core, functions are self-contained blocks of code that can perform specific tasks or operations when called [1]. They encapsulate a set of instructions, and one can think of them as named boxes that take some input, process it, and produce an output. Functions are a powerful tool for breaking down complex problems into smaller, manageable pieces, making code more modular, readable, and maintainable. In most programming languages, including Matlab, Python, C++, and many others, functions are a vital part of the syntax of the language and are used extensively to structure and control the flow of a program. They allow developers to write code that can be reused in various parts of a program, improving code organization and reducing redundancy. Functions come in various flavors, including simple functions that perform a single task, more complex functions that take multiple parameters, and even functions that return other functions. They play a crucial role in making code more efficient, modular, and easier to understand. The following examples are able to show various functions and their applications in more detail.

8.1 Built-In Functions/Methods

When it comes to Matlab programming, a robust collection of built-in functions and methods is at the readers disposal, allowing the reader to perform a wide array of tasks efficiently. In this set of examples (Built-In Functions/Methods), we will explore the practical application of these pre-defined functionalities to handle various common programming tasks [1]. These functions and methods serve as the building blocks that enable

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_8

developers to manipulate data, work with strings, arrays, and more, ultimately empowering them to create dynamic and feature-rich MATLAB applications. Let us explore some illustrative examples to showcase the versatility and power of MATLAB built-in functions and methods.

8.1.1 Ex. (125) – Built-in *Sin, Exp*

```
a = 3.1415;
b = sin(a);
c = exp(sin(a));
disp(b);
disp(c);
```

```
Output:

0.00009265358966049026
1.000092657882137
```

This example calculates the sine of the value approximately equal to *pi* and then calculates the exponential value of that sine. Several operations are being performed. First, a variable *a* is assigned the value 3.1415, which is an approximation of the mathematical constant π (*pi*). This value is then used in subsequent calculations. Next, a variable *b* is assigned the result of applying the *sin*() function to *a*. The *sin*() function calculates the sine of an angle in radians, thus, *b* will hold the sine of the angle approximately equal to 3.1415. Following that, a variable *c* is assigned the result of applying the *exp*() function to the sine of *a*. The *exp*() function calculates the exponential value of its argument, and in this case, it is applied to the sine of *a*. Therefore, *c* will contain the exponential value of the sine of the angle approximately equal to 3.1415. Next, the code shows the values of *b* and *c*.

8.1.2 Ex. (126) – Max between two integer variables

```
maxA = 6;
maxB = 10;
maxValue = max(maxA, maxB);

disp(maxValue);
```

```
Output:

10
```

This code example sets two variables, *maxA* and *maxB*, assigns them specific values, finds the maximum value between them, and shows the result to the console. Initially, the code begins by declaring two variables, *maxA* and *maxB*, and assigning them with numeric values namely 6 and 10, respectively. Next, the code calculates the maximum value between *maxA* and *maxB* using the *max*() function. The *max*() function is a built-in MATLAB function that takes multiple arguments and returns the largest value among them. In this specific case, it is used to find the maximum value between *maxA* and *maxB*. The result is then stored in a variable named *max*. Next, the content of variable *max* is then shown in the console for user inspection.

8.1.3 Ex. (127) – Max between two specific elements of an array

```
a = [6, 7, 1, 9];

maxValue = max(a([4, 2]));

disp(maxValue);
```

Output:

```
9
```

The snippet begins by declaring an array *a* containing four numerical elements: 6, 7, 1, and 9. This array is defined using square brackets and elements separated by commas. Next, the code declares a variable *maxValue* (because *max* is a keyword reserved for the function name). Next, the code uses the *max* function to find the maximum value between two elements of the array *a*. Specifically, it compares *a*(3) (which is the fourth element in the array, with an index of 4, holding the value 9) and *a*(1) (the second element in the array, with an index of 2, holding the value 7). In other words, *a*([4, 2]) creates a new array consisting of the 4th and 2nd elements of *a*. Therefore, *a*([4, 2]) is equivalent to [9, 7]. The *max* function returns the greater of the two values, in this case, 9. Therefore, this example creates an array and finds the maximum value between two elements of the array, then it shows the result.

8.1.4 Ex. (128) – Max over the values from an array

```
a = [6, 7, 1, 9];
b = [2, 5, 1, 1];
maxA = max(a);
maxB = max(b);
disp(maxA);
disp(maxB);
```

Output:

```
9
5
```

This code begins by defining two arrays, *a* and *b*, each containing a set of numerical values. Variable *a* holds the values 6, 7, 1, and 9, while *b* contains the values 2, 5, 1, and 1. The subsequent lines of code calculate the maximum value within each of these arrays. The *max*(*a*) function determines the maximum value in array *a*, and this result is stored in the variable *maxA*. Similarly, the *max* (*b*) function finds the maximum value in array *b* and assigns it to the *maxB* variable. Next, the code prints the values of *maxA* and *maxB* to the output.

8.1.5 Ex. (129) – Max over two array variables

```
a = [6, 7, 1, 9];
b = [2, 5, 1, 1];
maxA = max(a);
maxB = max(b);
maxValue = max(maxA, maxB);

disp(maxValue);
```

Output:

```
9
```

In this snippet, two arrays, *a* and *b*, are defined. The array a contains the elements 6, 7, 1, and 9, while the array *b* holds the values 2, 5, 1, and 1. Next, the *max* method is used to find the maximum value within each array. This method is called on the *max* function with the apply method used to pass the array as arguments. Essentially, it is a way to find the maximum value within an array, which is stored in the variables *maxA* and *maxB*. Then, another variable named *max* is defined and assigned the result of the *max* function, which takes *maxA* and *maxB* as its arguments. This will give us the maximum value among both arrays.

8.1.6 Ex. (130) – Random integers from 0 to 99 in an array

```
n = 10;
a = zeros(1, n);

for k = 1:n
    a(k) = randi(100) - 1;
end

disp(a);
```

Output:

```
41,72,71,20,2,
8,40,0,99,38
```

This code generates an array *a* with 10 random integers between 0 and 99, and prints it to the console. The example initializes an array with zero values called *a* and a variable *n* with a value of 10. It then enters a *for-loop* that will execute 10 times, starting with *k* as 1 and incrementing it in each iteration until it reaches 10 (since *n* is 10). Within the *for-loop*, the code generates a random number between 0 and 99 (inclusive) using the *randi* function (i.e., *randi*(100)–1). The *randi* function generates values from 1 to *N*, we must subtract 1 to get the values in the range 0 to 99. The resulting random integer is assigned to the *k*th element of the array *a*. In other words, it populates the array *a* with 10 random integers. Once the *for-loop* completes, the code shows the content of array *a*.

8.1.7 Ex. (131) – Insert random values in the elements of a matrix.

```
n = 3;
m = 3;
p = zeros(n+1, m+1);

r = '';
for i = 1:n+1
    for j = 1:m+1
        p(i, j) = floor(rand() * 10);
        r = [r, num2str(p(i, j)), ' '];
    end
    r = [r, '\n'];
end

fprintf(r);
```

```
Output:

3 1 0 7
9 3 0 2
4 3 4 2
5 7 8 3
```

This code from above generates a 3×3 matrix of random numbers, storing it in the *p* array, and builds a string representation of the matrix in the variable *r*, where each row is separated by a newline character. The example initializes several variables and uses nested loops to generate a matrix of random numbers. The code begins by declaring a few variables: (1) An empty array *p*, which will be used to store a matrix of random numbers. (2) A variable *n* set to the value 3, representing the number of rows in the matrix. (3) Another variable *m* that is set to 3, indicating the number of columns in the matrix. (4) An empty string *r* that will be used to accumulate the matrix elements as strings, with spaces and line breaks. The code then enters a nested loop structure to fill the *p* array with random numbers and build a string representation of the matrix. It uses two for loops. The outer loop, controlled by the variable *i*, iterates from 1 to $n + 1$ (inclusive), creating an array $p(i)$ for each row. Inside the outer loop, there is an inner loop controlled by the variable *j*, which iterates from 1 to $m + 1$ (inclusive). This loop populates each row ($p(i)$) with random numbers using the *rand* (i.e., *rand*() * 10) and stores them in the 2D array *p*. The code appends each of these random numbers to the string *r*, followed by a space (‘ ’), effectively building a string representation of the matrix row. Once the inner loop completes for each row, the code appends a newline character (‘\n’) to the string *r*, creating a new line in the matrix. Next, the content of variable r is shown in to output. Note that *rand* (used here) and *randi* (used in the previous example) are both MATLAB functions for generating random numbers, but they serve different purposes. The *rand* function produces uniformly distributed random floating-point numbers between 0 and 1, useful for when random fractions are needed. In contrast, *randi* is used to generate uniformly distributed random integers within a specified range, such as [1, 10], making it suitable for scenarios where discrete, whole numbers are required. The choice between them depends on whether the need is for random floating-point numbers or random integers within a certain interval.

8.1.8 Ex. (132) – Split string to integers by using a delimiter symbol

```
a = '2#5#7#1#1#2';
b = strsplit(a, '#');

disp(['b = ', strjoin(b, ' ')]);
```

Output:

```
b = 2 5 7 1 1 2
```

In this snippet, there are two main variables being used. The second variable is *a*, that is declared and assigned a string value, namely "2#5#7#1#1#2", which is a sequence of numbers separated by hash "#" symbols. The crucial operation in this code is "*strsplit(a, '#')*". Here, the string *a* is split into an array of substrings using the "#" character as the delimiter. The resulting substrings are stored in the *b* array. As a result, variable *b* becomes an array with the following elements: ['2', '5', '7', '1', '1', '2']. Lastly, the source code displays the contents of array *b* to the console using the *strjoin* complementary function, which concatenates all the elements in the array into a string. Thus, *strsplit* and *strjoin* functions are converters between data types.

8.1.9 Ex. (133) – Split string to array by using the "|" symbol

```
n = {};
m = {};

c = 'AAAAA|BBBBB|CCCCC';
n = strsplit(c, '|');

disp(n{3});
```

Output:

```
CCCCC
```

Here, the source code example defines two cell arrays *n* and *m* (cell arrays are used when working with strings). These arrays are initially empty, which means they do not contain any elements at the start. Next, a string variable *c* is defined and initialized with the value "AAAAA|BBBBB|CCCCC". This string contains three segments separated by the "|" character. The *strsplit* function is then applied to the string *c* using the "|" character as the delimiter. This function, as in the previous example, splits the string into substrings at each occurrence of the delimiter and stores them in the *n* array. Thus, after this line of code, the *n* array will contain three elements: "AAAAA","BBBBB", and "CCCCC". Next, the code outputs the element at index 3 of the *n* array, which corresponds to the string "CCCCC".

8.1.10 Ex. (134) – Cascading built-in functions (*strrep*, *length*)

```
a = '----##----------##--------';
q = '##';

b = length(a);
c = length(strrep(a, q, ''));

if c < b
    fprintf('a contains q\n');
end
```

Output:

```
a contains q
```

Here, we are working with two strings, *a* and *q*. The string *a* is initialized with the value "----##----------##--------", while *q* is set to "##". Next, we calculate the length of the string *a* and store it in the variable *b*. In this case, *b* will be 24, which is the total number of characters in the string *a*. Then, a transformation is performed on the string *a*. The *strrep* function is used to remove all occurrences of "##". Note that *strrep* in MATLAB is used for string replacement, allowing the search for a specific substring in a string and replace it with another substring. For example, *strrep*('Hello world', 'world', 'MATLAB') would change the string "Hello world" to "Hello MATLAB" by replacing "world" with "MATLAB". This function is case-sensitive and will replace all occurrences of the specified substring. Back to our code, the resulting string is stored in the variable *c*. Thus, *c* represents the length of the string *a* after removing all instances of "##". Once these calculations are made, we have two variables, namely: *b*, which is the original length of *a*, and *c*, which is the modified length of *a* after removing "##". The code then proceeds to check if *c* is less than *b*. If this condition is met, it means that the length of *a* was reduced by removing "##", indicating that "##" was indeed present in *a*. In this case, it will print the message "a contains q." Therefore, this code is essentially checking whether the string *a* contains the substring "##" and, if so, it outputs a message confirming the presence of "##" within *a*. Note that *strsplit* and *strjoin* functions can also be used to simulate the behavior of the *strrep* function.

8.1.11 Ex. (135) – Built-in *sort* function

```
b = [3, 6, 2, 78, 99, 1, 4];
b = sort(b);

disp(b);
```

Output:

```
1,2,3,4,6,78,99
```

This code snippet begins by declaring an array named *b* and populating it with a series of numerical values enclosed within square brackets. The array consists of the following elements: 3, 6, 2, 78, 99, 1, and 4. Following the array initialization, the code proceeds to invoke the *sort*() function on the array *b*. This method is used to arrange the elements

within the array in ascending order, meaning the numbers will be sorted from the lowest value to the highest. Next, the content of b is printed.

8.2 Making of Functions

The introduction of the concept of user-defined functions is a fundamental aspect of programming and an essential skill for any developer. In this set of examples, we will explore the creation and use of functions in MATLAB. Note that these native examples are functional in most programming languages once the syntax is adapted. Functions are reusable blocks of code that can be customized to perform specific tasks, making the code more organized, modular, and easier to maintain [1]. The definition of functions allows for logical encapsulation, improving code readability, and facilitating code reuse. Here, we explore a series of practical examples that showcase the power and versatility of user-defined functions.

8.2.1 Ex. (136) – Making of a function

```
a = 10;
b = compute(a);

disp(b);

function y = compute(x)
    y = x + x / x - x * x;
end
```

Output:

```
-89
```

The code initializes a with the value 10, performs a series of mathematical operations on a inside the *compute* function, and then it prints the result. The mathematical expression in the *compute* function:

$$compute(x) = \frac{x + x}{x - x \times x}$$

The above expression in meant to showcase the order of operations in a more graphical manner. Note, however, that the expression could be simplified as:

$$compute(x) = 1 + x - x \times x$$

which is equivalent to:

$$compute(x) = 1 + x - x^2$$

Nevertheless, the above example begins by declaring a variable a and assigning it the value 10. It then proceeds to declare another variable b and assigns it the result of a

function call, "*compute*(*a*)". The *compute* function is defined in the code and takes a single parameter *x*. Inside the function, a mathematical operation is performed on *x*. It computes $x + x / x - x * x$, which involves addition, division, subtraction, and multiplication operations. In the final step, the value of *b* is printed to the console.

8.2.2 Ex. (137) – Making of a function with more than one parameter

```
disp(mul(2,5));

function result = mul(a, b)
    result = a * b;
end
```

Output:

```
10
```

Essentially, this example computes the product of *a* and *b*. It defines a simple function and then calls it. The code begins with a call to the *disp* function with the argument *mul*(2,5). Next, the code defines a function named *mul* using the *function* keyword. This *mul* function takes two parameters *a* and *b*, to showcase an example with more than one parameter. Inside the body of the function, it performs a simple arithmetic operation, namely the multiplication of *a* by *b*, and then returns the result.

8.2.3 Ex. (138) – Gauss summation - sum all from 0 to *n*

```
disp(gs(100));

function result = gs(n)
    result = n*(n+1)/2;
end
```

Output:

```
5050
```

Here, we have a simple function called *gs* (Gauss summation) which takes a single argument, namely *n*. The purpose of this function is to calculate the sum of the first *n* natural numbers. The formula used to compute this sum is derived from a well-known arithmetic progression formula. The code returns the sum of the natural numbers from 1 to *n* using the formula:

$$gs(n) = \frac{n \times (n-1)}{2}$$

To demonstrate how this function works, there is a call to *gs*(100), which passes the value 100 as an argument to the *gs* function. This function call calculates the sum of the first 100 natural numbers and then prints the result. Thus, if one were to execute this code, it would output the sum of natural numbers from 1 to 100, which is 5050, because *gs*(100) would return 100 * (100 + 1) / 2, which equals 5050.

8.2.4 Ex. (139) – Function calls to other functions

```
function example
    d = main_app(66, 100);
    disp(d);
end

function cc = main_app(x, y)
    cc = sebastian(x, y);
end

function p = sebastian(a, b)
    p = daniela(a, b);
end

function result = daniela(a, b)
    result = a + b;
end
```

Output:

```
166
```

This source code example creates a series of functions that pass arguments and return values to one another. The *main_app* function is the entry point, and it ultimately calculates the sum of the two initial values, 66 and 100, by passing them through the *sebastian* and *daniela* functions, and the result is stored in variable *d*. Next, the value stored in variable *d* is printed to the console for user inspection.

8.2.5 Ex. (140) – A simple scanner to find the output distribution

```
a = distribution(3, 21);
disp(a);

function t = distribution(start, stop)
    t = "";
    for i = start:(stop-1)
        t = t + compute(i) + newline;
    end
end

function result = compute(x)
    result = x + x / x - x * x;
end
```

Output:

```
-5
-11
-19
-29
-41
-55
-71
-89
-109
-131
-155
-181
-209
-239
-271
-305
-341
-379
```

The above program calculates a distribution of values within a specified range and stores the result in a string variable. The distribution is obtained by applying the *compute* function to each value within the given range and concatenating the results with newline characters in between. For a short description, this code generates a string output representing a distribution of values. It begins by declaring a variable *a* and assigns it the result of a function call to *distribution*(3, 21). Then, it proceeds to print the value of *a* using a *disp* function. The *distribution* function is the heart of the program. It takes two parameters, *start* and *stop*, representing the range of values to consider. Inside the function, a variable *t* is initialized as an empty string. A *for-loop* is used to iterate over a range of values from *start* (inclusive) to *stop* (exclusive). During each iteration, the compute function is called with the current value of *i*, and the result is concatenated to the string *t* with a newline character to separate each value. Next, the resulting string *t* is returned. Note that the *compute* function is a simple mathematical operation that takes a single parameter *x*, and it calculates a value based on the formula: $x + x / x - x * x$ (mentioned earlier in this subchapter).

8.2.6 Ex. (141) – Function chaining - nested function calls

```
function main
    a = 3;
    b = c(c(c(c(a))));
    b = -b;

    disp(b);
end

function result = c(x)
    result = x + x / x - x * x;
end
```

Output:

```
756029
```

In this snippet, we have a series of operations and a function defined. First, a variable *a* is assigned the value 3. Then, a variable *b* is assigned the result of calling the function *c* repeatedly with the argument *a*. The function *c* takes a single argument *x* and performs a series of mathematical operations on it, including addition, division, and multiplication. The final value of *b* is negated, making it negative. Next, the value of *b* is shown to the console. Note that repeated calling of a function with its own result is often referred to as "function composition" or "function chaining." In our code, the function *c* is called multiple times with its own result, which effectively chains the function calls together. This can be a useful technique in some scenarios to apply a series of operations or transformations to a value in a concise and readable manner.

8.2.7 Ex. (142) – Function composition

```
a = [1, 2, 3, 4, 5];
t = 0;
b = c1(t, a);

disp(b);

function result = c1(t, a)
    result = 5 + c2(t, a);
end

function result = c2(t, a)
    result = c3(t, a) + 5;
end

function result = c3(t, a)
    s = 1;
    result = s + c4(t, a);
end

function result = c4(t, a)
    result = c5(t, a) + c5(t, a);
end

function result = c5(t, a)
    for i = 1:length(a)
        t = t + a(i);
    end
    result = t;
end
```

```
Output:

41
```

In this code snippet, we have a series of functions and variable assignments that perform calculations based on the input values. We have a constant array *a* containing the elements [1, 2, 3, 4, 5]. A variable *t* is initialized with the value 0. Then, a variable *b* is assigned the result of calling the function *c1* with the arguments *t* and *a*. The function *c1* takes two arguments *t* and *a* and returns the result of calling *c2* with the same arguments and adding 5 to it. The function *c2* takes two arguments *t* and *a* and returns the result of calling *c3* with the same arguments and adding 5 to it. The function *c3* takes two arguments *t* and *a*. Inside this function, a local variable *s* is initialized with the value 1. The function then returns the result of calling *c4* with the same arguments and adding *s* to it. The function *c4* takes two arguments *t* and *a* and returns the result of calling *c5* twice with the same arguments and summing the results. The function *c5* takes two arguments *t* and *a*. It then iterates through the elements of array *a*, adding each element to *t*. Then, it returns the modified value of *t*. The code concludes by printing the value of *b* to the console. The type of function calling demonstrated in the code is often referred to as "function composition" or "function chaining," especially when functions are designed to be composed together in a sequence. In our code, *c1* calls *c2*, which calls *c3*, which calls *c4*, and *c4* calls *c5*. This creates a chain of function calls where each function in the chain

relies on the results of the previous one to calculate its own result. This pattern is often used to break down complex operations into smaller, more manageable pieces.

8.2.8 Ex. (143) – Global vaiables and constants

```
% Constant.
a = 3.1415;

% Global variable.
global b;
b = 11;

b = compute();
fprintf('%d\n%.4f\n', b, a);

function result = compute()
    global b;
    x = b;
    result = x + x / x - x * x;
end
```

Output:

```
-109
3.1415
```

This code snippet demonstrates the use of constants, global variables, a function definition, and a function execution. It calculates a result based on the value of the global variable *b* and displays it along with the constant *a*. Two declarations are made: A constant *a* with the value 3.1415 and a global variable *b* with the value 11. The global variable *b* is later updated with the result of calling the *compute* function. The *compute* function takes the current value of *b*, assigns it to a local variable *x*, and then performs a series of mathematical operations on *x*, including addition, division, and multiplication. The result of these operations is then returned and assigned to *b* outside of the *compute* function. Then, the *fprintf* function is used to display the value of *b* followed by a newline character (“\n”) and the value of the constant *a*. Note that the printing format in *fprintf* is adjusted to print an integer (%d) and a floating-point number with four decimal places (%.4f).

8.2.9 Ex. (144) – Pure and impure functions

```
global a;
a = 10;

b = pure(a);
disp([num2str(b), ' & ', num2str(a)]);

c = inpure(a);
disp([num2str(c), ' & ', num2str(a)]);

d = inpure(a);
disp([num2str(d), ' & ', num2str(a)]);

function y = pure(x)
    y = x + x / x - x * x;
end

function y = inpure(x)
    global a;
    a = 11;
    y = x + x / x - x * x;
end
```

Output:

```
-89 & 10
-89 & 11
-109 & 11
```

The code illustrates the difference between “pure” and “impure” functions. “Pure” functions only depend on their input and do not modify any external state, while “impure” functions can modify external variables or have side effects. First, a variable *a* is assigned the value 10. Then, a variable *b* is assigned the result of calling the *pure* function with the argument *a*. The *pure* function takes a single argument *x* and performs a series of mathematical operations on it, including addition, division, and multiplication. After that, the value of *b* is printed along with the original value of *a*. Next, a variable *c* is assigned the result of calling the *inpure* function with the argument *a*. The *inpure* function also takes a single argument *x* but, in this case, it also modifies a global variable *a* by setting it to 11 before performing the same mathematical operations as the *pure* function. The value of *c* is printed along with the modified value of *a*. Next, a variable *d* is assigned the result of calling the *inpure* function with the argument *a* once more. Again, the *inpure* function modifies the global variable *a* by setting it to 11 and performs the mathematical operations. The value of *d* is printed along with the modified value of *a*.

8.2.10 Ex. (145) – Procedures vs Functions

```
global a b;

a = 16;
b = f(a);
disp(b);
p();
disp(b);

function y = f(x)
    y = x + x / x - x * x;
end

function p()
    global a b;
    x = a - 11;
    b = x + x / x - x * x;
end
```

Output:

```
-239
-19
```

In this code, a distinction between procedures and functions is shown. At the beginning of the code, a variable a is assigned the value 16. Then, a variable b is assigned the result of calling the function f with the argument a, and the value of b is printed to the output. The f function takes a single argument x and performs a series of mathematical operations on it, including addition, division, and multiplication. It returns the result of these operations. Next, there is a procedure named p. Procedures are similar to functions, but they do not return a value explicitly. Inside the p procedure, a new variable x is declared and assigned the result of subtracting 11 from the value of a. Then, b is re-assigned the result of a series of mathematical operations on x, including addition, division, and multiplication. Next, the value of b is printed to the output again, but this time within the p procedure. The key difference between functions and procedures is that functions return values, while procedures do not, and in this code, it is demonstrated how they can be used in different contexts.

8.3 Recursion

Function recursion is a fundamental concept in the world of programming, including MATLAB. It allows a function to call itself in a repetitive and self-referential manner, solving complex problems by breaking them down into smaller, more manageable subproblems. This technique is particularly useful when dealing with tasks that exhibit a recursive structure, such as traversing tree-like data structures, calculating factorials, and implementing various sorting algorithms [1]. Therefore, MATLAB provides a flexible environment for implementing recursive functions. Elegant and/or efficient recursive solutions can be created for various computational problems. Here, the function recursion will

be fully explored, the benefits will be understood and we will see how recursion works through practical examples.

8.3.1 Ex. (146) – Replacement for repeat loops with recursion

```
a = for_loop(0, 7, 0);
disp(a);

function result = for_loop(a, b, r)
    a = a + 1;

    % do stuff from
    r = r + 5;
    % to here.

    if a >= b
        result = r;
    else
        result = for_loop(a, b, r);
    end
end
```

Output:

```
35
```

Here, we have a custom recursive function called *for_loop* that serves as a replacement for traditional repeat loops. The variable *a* is assigned the result of calling the *for_loop* function with the initial parameters 0 for *a*, 7 for *b*, and 0 for *r*. The function is intended to simulate the behavior of a repeat loop. Within the *for_loop* function, variable *a* is incremented by 1 and variable *r* is increased by 5. A comment indicates where the actual processing or "do stuff" part would occur, which is not specified in this code. The function checks if *a* is greater than or equal to *b*. If this condition is met, it returns the value of *r*. Otherwise, it calls itself recursively with the updated values of *a*, *b*, and *r*. This recursive process continues until *a* is greater than or equal to *b*. At the end, the result of the *for_loop* function is stored in the variable *a*, and its value is printed to the console or the output destination for inspection.

8.3.2 Ex. (147) – Repeat string *n* times recursively

```
a = x('#', '', 10);
disp(['Repeat:' newline '[' a ']']);

function result = x(c, s, n)
    s = strcat(s, c);

    if length(s) >= n
        result = s;
    else
        result = x(c, s, n);
    end
end
```

Output:

```
Repeat:
[##########]
```

This code effectively demonstrates string repetition through recursion. First, the variable *a* is assigned the result of calling the function *x* with the arguments ("#", "", 10). The function *x* is designed to repeat the character *c* (in this case, "#") *n* times and initially starts with an empty string *s*. It appends the character to the string *s* in each recursive call until the length of *s* is greater than or equal to *n*. After that, there is a *disp* statement that displays the value of a wrapped inside square brackets preceded by "Repeat:\n." The *x* function essentially implements a form of recursion to repeat a character a specified number of times. It appends the character to the string and recursively calls itself until the desired length is reached.

8.3.3 Ex. (148) – Sum from 0 to *n* recursively

```
b = sum_recursive(23);
fprintf('Sum: [%d]\n', b);

function result = sum_recursive(n)
    if n <= 1
        result = n;
    else
        result = n + sum_recursive(n - 1);
    end
end
```

Output:

```
Sum:[276]
```

This code calculates the sum of numbers from 0 to 23 using a recursive function and then displays the result as part of a message. It starts by defining a variable *b* and assigning it the result of calling the *sum_recursive* function with the argument 23 (note that the *sum* keyword is reserved in MATLAB for the built-in *sum* function). The *sum_recursive* function calculates the sum of numbers from 0 to *n* using recursion. Inside the *sum_recursive* function, there is a base case check. If *n* is less than or equal to 1, it returns *n*, which is the base case of the recursion. Otherwise, it returns *n* added to the result of *sum_recursive*$(n - 1)$, which is the recursive call. After calculating the sum, we use the *disp* function to display a message that includes the computed sum. The message is constructed by concatenating the string "Sum:[" with the value of *b*, and then adding a closing "]".

8.3.4 Ex. (149) – Factorial from 0 to *n*

```
c = factorial_recursive(10);
fprintf('Factorial:\n[%d]\n', c);

function result = factorial_recursive(n)
    if n <= 1
        result = 1; % Factorial of 0 and 1 is 1.
    else
        result = factorial_recursive(n - 1) * n;
    end
end
```

Output:

```
Factorial:
[3628800]
```

Here, this code snippet calculates the factorial of 10 using a recursive function and then prints the result with an informative message. The code starts by calculating the factorial of the number 10 and assigns the result to the variable *c*. The *disp* function shows a message indicating "Factorial" and then displays the value of *c*. The *factorial* function is defined to calculate the factorial of a given number *n*. It uses recursion to perform the calculation. If *n* is less than or equal to 1, it returns 1. Otherwise, it recursively calls itself with $n - 1$ and multiplies the result by *n* to calculate the factorial.

8.3.5 Ex. (150) – Generate a sequence recursively

```
d = sequence(5, zeros(1, 5), 0, 5);
disp(['A sequence:', newline, '[', num2str(d), ']']);

function m = sequence(n, m, i, t)
    m(i + 1) = n;  % MATLAB indexing starts at 1.
    i = i + 1;

    if i >= t
        return
    else
        m = sequence((n - 1) + (n - 2), m, i, t);
    end
end
```

Output:

```
A sequence:
[5,7,11,19,35]
```

This code demonstrates the generation of a sequence of numbers using a recursive function and then displays the result as a string. A variable *d* is assigned the result of calling the *sequence* function with the arguments 5, a zero filled array (*zeros*(1, 5)), 0, and 5. Then, a string is printed to the console, which includes the value of *d* inside a string representation of an array. The *sequence* function takes four parameters: *n*, *m*, *i*, and *t*. It appends the value of *n* to the array *m* at index $i + 1$ and increments *i*. If *i* is greater than or equal to *t*, the function returns the array *m*. Otherwise, it makes a recursive call to *sequence*, updating *n* to be the sum of the previous two values of *n* (i.e., (*n*–1) and

(*n*–2)) and continuing to build the sequence. The result is then returned to variable *d* and printed in the output using the *disp* function.

8.3.6 Ex. (151) – Generate fibonacci recursively

```
e = fibonacci(2, [0, 1, 1], 5);

fprintf('Fibonacci:\n[');
fprintf('%g ', e);
fprintf(']\n');

function result = fibonacci(n, m, t)
    n = n + 1;
    m(n+1) = m(n) + m(n-1);

    if n >= t
        result = m;
    else
        result = fibonacci(n, m, t);
    end
end
```

Output:

```
Fibonacci:
[0,1,1,2,3,5]
```

This code demonstrates the use of a recursive function to generate a *Fibonacci* sequence up to a specified length. First, a variable *e* is assigned the result of calling the *fibonacci* function with the arguments 2, [0, 1, 1], and 5. This function calculates the *Fibonacci* sequence, starting with the initial values [0, 1, 1], and generates the sequence up to the specified length of 5. The *fibonacci* function takes three parameters: *n*, *m*, and *t*. It increments *n* and updates the next value in the *m* array by summing the previous two values in the sequence. It continues to do this recursively until *n* is greater than or equal to *t*, at which point it returns the *m* array containing the *Fibonacci* sequence. Next, the *disp* function displays the *Fibonacci* sequence stored in the *e* variable, preceded by the string "Fibonacci:\n["and followed by"]".

8.3.7 Ex. (152) – Sum all from array recursively

```
q = [1, 3, 3, 4, 5, 9];
f = sum_array(length(q), q, 0);
fprintf('Sum array: [%d]\n', f);

function result = sum_array(n, q, r)
    r = r + q(n);

    if n <= 1
        result = r;
    else
        result = sum_array(n - 1, q, r);
    end
end
```

Output:

```
Sum array:[25]
```

This time, the code demonstrates a recursive function for summing the elements of an array variable. First, an array *q* is defined with some numeric elements. Then, the result of calling the *sum_array* function with three arguments is assigned to a variable *f*: the length of the array *q*, the array *q* itself, and an initial value of 0. The purpose of the *sum_array* function is to recursively sum the elements of the array. The *sum_array* function takes three arguments: *n* represents the current index, *q* is the array, and *r* is the running sum. In order to eliminate any confusion, note here, if not explained before, that arguments are the values passed to the parameters of the functions. In order to continue, the code then adds the element at the current index to the running sum. If the index *n* is less than or equal to 0, the function returns the running sum *r*. Otherwise, it calls itself recursively with a decremented index and the updated running sum. Next, the code prints a message to the console that includes the sum among the elements of the array *q*, which is calculated using the *sum_array* function.

Objects

9

MATLAB is a versatile and widely-used computer language that empowers engineers and scientists to create dynamic and interactive simulations, either on browsers with server-side execution or on local machines with a pre-installed MATLAB package. In MATLAB, objects provide a way to encapsulate data and associated functions, allowing for more structured and modular programming. This concept, stemming from Object-Oriented Programming (OOP), enables MATLAB users to create custom data types with unique behaviors and properties [1]. An object in MATLAB is an instance of a class, where a class is a blueprint defining the properties (data) and methods (functions) that the objects of this class will have. MATLAB classes are defined using *classdef* keyword, and they can include a variety of components such as properties, methods, events, and enumeration. Properties in a class are variables that hold data specific to an object. Methods are functions that can operate on the data of the object. Constructors are special methods used to create instances of the class, and destructors clean up when objects are no longer needed. Also, the object-oriented approach in MATLAB allows for inheritance, where one class can inherit properties and methods from another, enabling code reuse and simplification. It also supports encapsulation, where an data of the object can be protected from outside interference, and polymorphism, where functions can operate on objects of different classes. This approach is particularly useful for complex programs, as it helps organize code into manageable sections and promotes code reuse. In MATLAB, objects and classes can be used in a wide range of applications, from simple data structures to complex GUI development and system modeling. Another important aspect regarding objects is JSON (JavaScript Object Notation). JSON is a lightweight and text-based data

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_9

interchange format that plays a crucial role for data storage and file I/O operations. Therefore, this chapter presents examples related to concepts of MATLAB objects, constructors, methods, and the possibility of working with JSON.

9.1 Constructors and Methods

When the definition of a class in MATLAB is made, the keyword *classdef* is used, followed by the class name. The body of the class includes properties (variables specific to that class) and methods (functions specific to that class). Properties are defined using the properties block, and they can have attributes like access modifiers (public, private, or protected). Methods are defined in the methods block and can include constructors, regular methods, and special methods like setters and getters. The constructor is a special method with the same name as the class. It is automatically called when a new object of the class is created. Constructors can take arguments, which are used to initialize the properties of an object. For example, in a class named *MyClass*, the constructor would be a function named *MyClass* within the methods block. The constructor is responsible for setting up the initial state of a new object. Methods, like constructors, are defined within the methods block. These can be functions that operate on the properties of the objects, modifying them or performing computations. MATLAB supports various method attributes, such as static methods (which do not require an object to be called) and abstract methods (which declare a method signature without an implementation, to be defined in subclasses). Inheritance is another key feature in the OOP capabilities of MATLAB. A class can inherit from another class, gaining its properties and methods. This is done using the "<" symbol in the class definition. For example, classdef *NewClass* "<" *BaseClass* would define *NewClass* as a subclass of *BaseClass*. MATLAB approach to OOP provides a structured way to organize and manage code, particularly for complex mathematical modeling and data analysis tasks. It allows for creating reusable code components, which can be particularly beneficial in large-scale scientific and engineering projects. The encapsulation of data and functions within classes and objects helps in maintaining a clean and understandable code structure, promoting good software design practices. In other words, OOP is meant to lower the entropy that gathers over time for the entire project, whatever that MATLAB project may be.

9.1.1 Ex. (153) – Using an object constructor

```
% file obj.m

classdef obj
    properties
        ax
        bx
        cx
        dx
    end

    methods
        function obj = obj(a, b, c, d)

            % Convert string to
            % cell array of characters.

            obj.ax = cellstr(a');
            obj.bx = length(obj.ax);
            obj.cx = c - b;
            obj.dx = d * c;
        end
    end
end


% script file test.m

o1 = obj('some', 66, 50, 77);
o2 = obj('text', 85, 48, 77);

% Displaying the properties.
disp([strjoin(o1.ax, ''), ' | ', ...
      strjoin(o2.ax, '')]);

disp([num2str(o1.bx), ' | ', num2str(o2.bx)]);
disp([num2str(o1.cx), ' | ', num2str(o2.cx)]);
disp([num2str(o1.dx), ' | ', num2str(o2.dx)]);

% Modification of bx property from o1.
o1.bx = 100;

% Displaying the modified property
disp([num2str(o1.bx), ' | ', num2str(o2.bx)]);
```

Output:

```
s,o,m,e | t,e,x,t
4 | 4
-16 | -37
3850 | 3696
100 | 4
```

The above implementation demonstrates object-oriented programming in MATLAB, with the creation of objects, property assignments, and property value modification. The *obj* function is a constructor function that takes four parameters *a*, *b*, *c*, and *d*. Within the function, *obj.ax* is assigned the result of splitting the string *a* into an array of characters, whereas *obj.bx* is assigned with the length of the array *obj.ax*. Also, *obj.cx* is assigned the result of subtracting *b* from *c*, and *obj.dx* is assigned the result of multiplying *d* by *c*. Once the constructor function is defined, two objects (*o1* and *o2*) are created using the

obj constructor with different parameter values. The *disp* function is then used to display various properties of these objects with some string concatenation. The properties of *o1* and *o2* are printed side by side for comparison. Then, the value of the *bx* property of *o1* is modified to 100, and its new value is printed alongside the unchanged *bx* value of *o2*. Note that the class definition is made in a separate file (file: *obj.m*) than the main script (file: *test.m*).

9.1.2 Ex. (154) – An object with three properties and a method (I)

```
% This example creates an
% object with three properties.
% The cx property is a method.

% file MyObject.m

classdef MyObject
    properties
        ax = 'this';
        bx = 'text';
    end
    methods
        function result = cx(obj)
            result = [obj.ax, ' ', obj.bx];
        end
    end
end

% script file test.m

% Creation of an instance of
% MyObject and calling the
% cx method.

obj = MyObject();
disp(obj.cx());
```

Output:

```
this text
```

In this code snippet, a class, *MyObject*, is defined with properties *ax* and *bx*, and a method *cx*. In this class definition, the properties *ax* and *bx* are initialized to strings “this” and “text”, respectively. The method *cx* when called on an instance of *MyObject*, concatenates these properties with a space, effectively returning the string “this text”. The script *test.m* demonstrates how to create an instance of *MyObject* and use its method. An object *obj* is instantiated from *MyObject*, and the method *cx* is invoked (*obj.cx*). This results in displaying the concatenated string defined in the method *cx*. This encapsulation of data and functions in objects allows for more organized and manageable code, especially in complex programs where entropy of software must be maintained at low levels. It promotes the principles of inheritance, encapsulation, and polymorphism, facilitating code reuse and simplicity in handling various data structures, GUI development, and system

modeling. Also, note again that the class definition is made in *MyObject.m* file and the script is stored in *test.m*.

9.1.3 Ex. (155) – An object with three properties and a method (II)

```
% This example creates an object
% with three properties.
% The cx property is a method.


% Create the object
obj = struct;
obj.ax = 'this';
obj.bx = 'text';
obj.cx = @cx; % Assign the function handle to cx.

% Use the function and access properties.
disp(obj.cx(obj, '-'));
disp(obj.ax);
disp(obj.bx);

% Modify properties
obj.ax = 'super';
obj.bx = 'string';
disp(obj.ax);
disp(obj.bx);
disp(obj.cx(obj, '+'));

% Define the cx function.
function t = cx(obj, g)
    t = [obj.ax, g, num2str(length(obj.bx))];
end


% or a different version:


obj = struct('ax', 'this', 'bx', 'text', 'cx', @cx);

disp(obj.cx(obj, '-'));
disp(obj.ax);
disp(obj.bx);

obj.ax = 'super';
obj.bx = 'string';
disp(obj.ax);
disp(obj.bx);
disp(obj.cx(obj, '+'));

function t = cx(obj, g)
    t = [obj.ax, g, num2str(length(obj.bx))];
end
```

Output:

```
this-4
this
text
super
string
super+6
```

Here, two different versions are presented. In the first version, an object *obj* is created as a *struct* with properties *ax*, *bx*, and a method *cx* represented by a function handle. Initially, *obj.ax* is assigned the value "this" and *obj.bx* the value "text". The *cx* function, defined at the end of the script, takes an object and a character as inputs and concatenates the *ax* property of the object, the input character, and the length of the *bx* property, converted to a string. When *cx* is called with *obj.cx*(*obj*, '−'), it outputs a string combining *obj.ax*, the character "−", and the length of *obj.bx* as a string. The script then displays the values of *obj.ax* and *obj.bx*. The properties *ax* and *bx* of the object *obj* are then modified to "super" and "string", respectively. The script displays these new values and again calls the *cx* function, this time with a "+" character. The *cx* function, therefore, produces a new string based on the updated properties of *obj*. Thus, this script exemplifies how structures in MATLAB can be used to mimic object-oriented programming features, encapsulating data and functions within a single entity. Note that this approach is particularly useful for organizing complex data and functionality in a clear and manageable manner. Also, a secondary implementation is shown below the first. The main difference between the two MATLAB code snippets is in how the *obj* struct is created and initialized. In the first snippet, the structure *obj* is first created as an empty structure using *obj* = *struct*; and then its properties *ax*, *bx*, and *cx* are individually assigned. This approach is more verbose, involving the creation of an empty structure followed by separate assignments for each property. In contrast, the second snippet initializes the *obj* structure with its properties in a single line using struct('ax', 'this', 'bx', 'text', 'cx', @cx). This method is more concise, combining the structure creation and property initialization into one step. Beyond this initial difference in struct creation, the rest of the operations in both snippets are identical, including the modification of properties and the use of the cx function. Functionally, the two snippets are equivalent and will yield the same results, with the difference being a matter of coding style and preference for initialization methods.

9.1.4 Ex. (156) – An object with complex methods

```
% file MyObject.m

classdef MyStatistics
    properties
        AV = 0;
        SD = 0;
        CV = 0;
    end
    methods
        function obj = dx(obj, a)
            n = length(a);
            b = sum(a);
            x = b / n;
            e = sum((a - x).^2);
            s = sqrt(e / (n - 1));
            c = s / x;
            obj.AV = x;
            obj.SD = s;
            obj.CV = c;
        end
    end
end


% script file test.m

a = [5, 6, 2, 9, 44, 200];
statsObj = MyStatistics();

% Reassign the returned object
statsObj = statsObj.dx(a);

fprintf('AV: %f\n', statsObj.AV);
fprintf('SD: %f\n', statsObj.SD);
fprintf('CV: %f\n', statsObj.CV);
```

Output:

```
AV: 44.33
SD: 77.83
CV: 1.75
```

This is a more useful representation of an object and its properties. Namely, an object named *obj* is defined using *struct*. In the given example, a class named *MyStatistics* is defined, which encapsulates the functionality for basic statistical analysis. This class, defined in a file named *MyObject.m*, has three properties: *AV* (Average), *SD* (Standard Deviation), and *CV* (Coefficient of Variation). These properties are initialized to zero. The class also includes a method named *dx*, which takes an array of numbers as an input, calculates their average, standard deviation, and coefficient of variation, and then updates the properties of the object with these calculated values. The *dx* method calculates the average (x), standard deviation (s), and coefficient of variation (c) of the input array a. It then assigns these values to the properties of the object (*AV*, *SD*, *CV*). Noticeably, the method returns the modified object, which is a common practice in object-oriented programming to allow for method chaining or updating the current instance. In a separate script file *test.m*, the class is utilized to perform statistical calculations. An array a is

defined with a set of numbers. An instance of *MyStatistics* is created, and the *dx* method is called with *a* as an argument. Since *dx* returns the modified object, it is essential to reassign this returned object to *statsObj*. Next, the script prints out the calculated average, standard deviation, and coefficient of variation. This example demonstrates how the object-oriented capabilities of MATLAB can be used to create modular, reusable, and well-structured code.

9.1.5 Ex. (157) – Generate multiple objects with methods

```
% file objx.m

classdef objx
    properties
        AV = 0;
        SD = 0;
        CV = 0;
    end
    methods
        function this = dx(this, a)
            n = length(a);
            b = sum(a);
            x = b / n;
            e = sum((a - x).^2);
            s = sqrt(e / (n - 1));
            c = s / x;

            this.AV = x;
            this.SD = s;
            this.CV = c;
        end
    end
end


% script file test.m

a = [5, 6, 2, 9, 44, 200];
b = [7, 4, 6, 8, 6, 4];

box1 = objx();
box2 = objx();

box1 = box1.dx(a);
box2 = box2.dx(b);

fprintf('box 1 - AV: %f\n', box1.AV);
fprintf('box 1 - SD: %f\n', box1.SD);
fprintf('box 1 - CV: %f\n', box1.CV);
fprintf('----------------------\n');
fprintf('box 2 - AV: %f\n', box2.AV);
fprintf('box 2 - SD: %f\n', box2.SD);
fprintf('box 2 - CV: %f\n', box2.CV);
```

Output:

```
box 1 - AV: 44.333333
box 1 - SD: 77.832298
box 1 - CV: 1.7556157
----------------------
box 2 - AV: 5.8333333
box 2 - SD: 1.6020819
box 2 - CV: 0.2746426
```

The provided MATLAB code demonstrates object-oriented programming by defining and using a class named *objx*. This class encapsulates functionality to compute and store statistical metrics such as average (*AV*), standard deviation (*SD*), and coefficient of variation (*CV*) for a set of numbers. In *objx.m*, the *classdef objx* defines the class. It contains three properties: *AV*, *SD*, and *CV*, all initialized to 0. These properties are designed to store the calculated statistical values. The methods block contains a single method named *dx*. This method calculates the average, standard deviation, and coefficient of variation for an array *a* (explained later in this book in great detail). Inside *dx*, the method calculates the mean (*x*), standard deviation (*s*), and coefficient of variation (*c*) using standard statistical formulas. It then assigns these values to the properties of the object: *this.AV*, *this.SD*, and *this.CV*. The method is defined to return the modified object, *this*, allowing the changes to the properties to be saved in the object instance. In *test.m*, the script demonstrates how to use the *objx* class. It first creates two arrays, *a* and *b*, with different sets of numbers. Then, it creates two instances of the *objx* class, named *box1* and *box2*, using the constructor of the class. The method *dx* is called on both *box1* and *box2*, passing in the arrays *a* and *b*, respectively. This updates the properties of each *objx* instance with the calculated statistical values for their respective arrays. Next, the script prints out the average, standard deviation, and coefficient of variation for each of the two *objx* instances, displaying the computed statistics for the data sets *a* and *b*. This code effectively showcases how object-oriented MATLAB features can be used to encapsulate data and related functionality within a class, providing a clear and modular approach to handling complex data and operations.

9.2 JSON

Constructors and methods are fundamental building blocks in the world of software engineering, serving as the essential tools for creating and manipulating data within software applications. Just as constructors are responsible for initializing objects and defining their initial state, methods enable us to interact with and modify those objects during runtime. Now, for data exchange and storage, JSON (Matlab Object Notation) provides a perfect illustration of this synergy between constructors and methods [1]. JSON is a lightweight data interchange format that relies on key-value pairs to represent structured data. Constructors, in the context of JSON, serve as blueprints for creating complex data structures, while methods are used to access and manipulate the data contained within these structures. In this dynamic interplay, constructors and methods play a pivotal role in harnessing the power of JSON to manage and transmit data efficiently in the digital world. Thus, below are a number of examples that explain concepts such as serialization and deserialization.

9.2.1 Ex. (158) – Object to JSON

```
% a Matlab object...
% ...converted into JSON:

% Create a structure
obj = struct('v1', 1, 'v2', 2, 'v3', 3);

% Convert the structure to a JSON string.
txt = jsonencode(obj);

disp(txt);

% send JSON:
Url = 'index.php?obj=';
r = webread([Url, txt]);
```

Output:

```
{"v1":1,"v2":2,"v3":3}
```

Here, we have a simple object manipulation operation where a MATLAB object is converted into JSON format that can potentially be a parameter to a URL (please see the comments). First, an object named *obj* is defined using *struct* notation. This object has three properties: "v1", "v2", and "v3", each assigned a numeric value. Next, the *jsonencode*(*obj*) function is used to convert the *obj* object into a JSON-formatted string. This JSON representation will look like {"v1":1,"v2":2,"v3":3}. Then, the *txt* variable is assigned the JSON-formatted string produced by *jsonencode*(*obj*). Afterwards, there is a comment that mentions sending the JSON data. The commented lines suggest the construction a URL string that includes the JSON data as a query parameter. In order to send the JSON data to a URL, one would typically need to use an *webread* request method. Therefore, in order to summarize this example, this code snippet creates a MATLAB object, converts it to a JSON string, and suggests the intent or ability to send that JSON data to a URL and get a response from the server.

9.2.2 Ex. (159) – JSON to Object

```
% myJSON is text received in JSON format.
% Convert JSON into a Matlab object:

txt = '{"v1":1,"v2":2,"v3":3}';

% Convert JSON string to a MATLAB structure.
obj = jsondecode(txt);

disp(obj.v1);
disp(obj.v2);
disp(obj.v3);
```

Output:

```
1
2
3
```

Here, the variable *txt* is assigned with a string containing JSON data, which represents an object with three key-value pairs: "v1" with a value of 1, "v2" with a value of 2,

and “v3” with a value of 3. The *jsondecode*(*txt*) method is used to convert the JSON-formatted string stored in the *txt* variable into a MATLAB object. This method parses the JSON data and creates a corresponding MATLAB object, which is then stored in the *obj* variable. Three separate *disp* statements are used to log the values of the *v1*, *v2*, and *v3* properties of the *obj* object to the console. Thus, when this code is executed, it will take the JSON string in *txt*, convert it into a MATLAB object named *obj*, and then print the values associated with the properties *v1*, *v2*, and *v3* to the console.

9.2.3 Ex. (160) – Anything to object to string

```
a = {'a', 'b', 'c'};

b = [0, 1, 0;
     1, 1, 1;
     0, 1, 0];

c = struct('c1', {a}, 'c2', {b}, 'c3', 42);

obj = struct('v1', {a}, 'v2', {b}, 'v3', {c});

txt = jsonencode(obj);

disp(txt);
```

Output:

```
{"v1":["a","b","c"],"v2":[[0,1,0],[1,1,1],[0,1,0]],"v3":
{"c1":["a","b","c"],"c2":[[0,1,0],[1,1,1],[0,1,0]],"c3":42}}
```

Several variables and objects are defined and manipulated in this example. Variable *a* is declared as an array containing three string elements: “a,” “b,” and “c.” Variable *b* is declared as a two-dimensional array (a matrix) containing three arrays. Each inner array represents a row in the matrix and consists of integer values. Variable *c* is defined as an object with three properties: “c1,” “c2,” and “c3.” “c1” is assigned the value of the array

a, “c2” is assigned the value of the two-dimensional array *b*, and “c3” is assigned the integer value 42. For some variation of data, variable *obj* is defined as an object with three properties: “v1,” “v2,” and “v3.” Thus, “v1” is assigned the value of the array *a*, “v2” is assigned the value of the two-dimensional array *b*, and “v3” is assigned the value of the object *c*. The *jsonencode*(*obj*) function is called to convert the obj object into a JSON-formatted string. This string represents the structured data in the *obj* object as a text format. The code essentially defines and structures data in the form of arrays, objects, and matrices, and then converts this data into a JSON string using the *jsonencode*(*obj*) function before logging it to the console.

9.2.4 Ex. (161) – Complex string to object (I) - direct nested access

```
txt = ['{"v1":["a","b","c"],"v2":' ...
       '[[0,1,0],[1,1,1],[0,1,0]],' ...
       '"v3":{"c1":["a","b","c"],' ...
       '"c2":[[0,1,0],[1,1,1],[0,' ...
       '1,0]],"c3":42}}'];

obj = jsondecode(txt);

disp(obj.v1{2});
disp(obj.v2(1, 2));
disp(obj.v3.c2);
disp(obj.v3.c2(2, 2));
```

Output:

```
b
1
0,1,0,1,1,1,0,1,0
1
```

A variable *txt* is assigned with a JSON-formatted string. This string represents a JSON object with three key-value pairs. The first key, “v1”, has an array value [“a”, “b”, “c”]. The second key, “v2”, contains a nested array [[0, 1, 0], [1], [0, 1, 0]]. The third key, “v3”, holds another nested JSON object with three key-value pairs: “c1” with an array value [“a”, “b”, “c”], “c2” with a nested array value [[0, 1, 0], [1], [0, 1, 0]], and “c3” with a numeric value 42. Following the JSON object creation, the code proceeds to parse this JSON string into a MATLAB object using *jsondecode*(*txt*). After parsing, it demonstrates the use of the object by printing specific values. It prints the second element of the “v1” array, the value at the first index of the second array within “v2”, the entire “c2” object within “v3”, and finally, the value at the second index of the second array within “c2” (second row and second column). The main purpose of this example is to access the nested values found in the main properties of the object.

9.2.5 Ex. (162) – Complex string to object (II) - nested access by reference

```
txt = ['{"v1":["a","b","c"],"v2":' ...
       '[[0,1,0],[1,1,1],[0,1,0]],' ...
       '"v3":{"c1":["a","b","c"],' ...
       '"c2":[[0,1,0],[1,1,1],[0,' ...
       '1,0]],"c3":42}}'];

obj = jsondecode(txt);

% Extract values from
% the parsed object.

a = obj.v1;
b = obj.v2;
c = obj.v3;

disp(a);
disp(b);
disp(c.c1);
disp(c.c2);
disp(c.c3);
```

Output:

```
a b c
0 1 0 1 1 1 0 1 0
a b c
0 1 0 1 1 1 0 1 0
42
```

A JSON string named *txt* is defined, which contains three key-value pairs. The first key, “v1”, maps to an array of three string elements: “a,” “b,” and “c.” The second key, “v2”, maps to a nested array of numeric values. This array has three sub-arrays, each containing numeric values. The third key, “v3,” maps to an object with three key-value pairs, namely: (i) “c1” is associated with an array that contains three string elements, similar to the array in “v1.” (ii) “c2” is associated with a nested array that is structurally similar to “v2,” containing numeric values. (iii) “c3” corresponds to a numeric value, which is specifically 42. After defining the JSON string, the code uses *jsondecode*(*txt*) to convert it into an object, which is then stored in the variable *obj*. Subsequently, the code extracts specific components from the parsed object: Variable *a* holds the value of *obj.v1*, which is the first array. Variable *b* holds the value of *obj.v2*, which represents the nested array. Also, *c* holds the value of *obj.v3*, which is an object with sub-properties. Next, the code prints these extracted values for user inspection. The main purpose of this example is to assign parts of one object (i.e. *obj*) into a new object (i.e. *c*) and access the nested values found in its properties (i.e. *c1*, *c2*, and *c3*).

9.2.6 Ex. (163) – Make 1D array from parts of an object

```
txt = ['{"v1":["a","b","c"],"v2":' ...
       '[[0,1,0],[1,1,1],[0,1,0]],' ...
       '"v3":{"c1":["a","b","c"],' ...
       '"c2":[[0,1,0],[1,1,1],[0,1,0]],"c3":42}}'];

obj = jsondecode(txt);

% Initialize empty cell array.
a = cell(size(obj.v3.c1));

% Extract elements from
% the nested structure.

for i = 1:numel(obj.v3.c1)
    a{i} = obj.v3.c1{i};
end

disp(a);
```

Output:

```
a b c
```

Here, a one-dimensional array *a* is made based on values found inside a complex object (*obj*). This code begins by defining a variable *txt*, which is a string containing JSON data. As before, the JSON data within *txt* includes three key-value pairs: "v1", "v2", and "v3": (i) The "v1" key has an array as its value containing the elements "a," "b," and "c." (ii) The "v2" key has a nested array as its value, which represents a 3×3 matrix with binary values. (iii) The "v3" key has an object as its value with three key-value pairs: "c1", "c2", and "c3". (iv) The "c1" key has an array as its value, similar to the "v1" array. (v) The "c2" key has a nested array as its value, representing another 3×3 matrix with binary values. (vi) The "c3" key has a numerical value, which is 42. The code then proceeds to parse the JSON string stored in *txt* into an object and assigns it to the variable *obj*. It initializes an empty array *a* and then uses a *for-loop* to iterate over the elements of the *c1* array within the *v3* object. For each element, it assigns the value to the corresponding index in array *a*. The purpose of this code is to iterate over a subset of elements inside the main object.

9.2.7 Ex. (164) – Make a matrix from parts of an object

```
txt = ['{"v1":["a","b","c"],"v2":' ...
       '[[0,1,0],[1,1,1],[0,1,0]],' ...
       '"v3":{"c1":["a","b","c"],' ...
       '"c2":[[0,1,0],[1,1,1],[0,1,0]],"c3":42}}'];

obj = jsondecode(txt);

% Initialize an empty array. Since
% obj.v3.c2 is a numeric array,
% cell is not needed here.

a = [];

% Copy the 2D array
% from obj.v3.c2 to a.

for i = 1:size(obj.v3.c2, 1)
    a(i,:) = obj.v3.c2(i,:);
end

% Convert and display the
% 2D array as a string.

fprintf(SMC(a));

% Function to convert matrix to
% string representation. This
% function must be defined at
% the end of the script.

function r = SMC(m)
    r = '';
    for i = 1:size(m,1)
        for j = 1:size(m,2)
            r = [r, num2str(m(i,j)), ' '];
        end
        r = [r, '\n'];
    end
end
```

Output:

```
0 1 0
1 1 1
0 1 0
```

In this last implementation of this subchapter, a matrix *a* is made based on values found inside a complex object (*obj*). Just as before, the code starts by defining a JSON string that contains nested arrays and objects. It then uses *jsondecode* MATLAB function to parse this JSON string into a MATLAB structure named *obj*. Once the JSON string is parsed, an empty array *a* is initialized to prepare for data storage. The script then proceeds to copy a 2D numeric array from *obj.v3.c2* into the previously initialized empty array a using a *for-loop* that iterates over the rows of the *obj.v3.c2* array. Once the 2D array has been successfully copied, the script calls the well-known function named *SMC*, which is defined later in the script, passing the array *a* as an argument. The *SMC* function is designed to convert the numeric matrix it receives into a string representation, where each element

is separated by a space and each row of the matrix is separated by a newline character. The resulting string is then output to the MATLAB command window using the *fprintf* function. The *SMC* function itself is composed of nested for loops that iterate over each element of the matrix, convert each element to a string using *num2str*, and concatenate them with spaces and newline characters to form the final string. The function is defined at the end of the script, which is a requirement in MATLAB for scripts that contain function definitions. This organization ensures that the main script code is executed first, followed by the local function definitions, which can only be called from within the script itself.

Moderate Examples

10

A spectrum of coding examples exists, that range from the very basic to the highly advanced. Nestled comfortably in between these extremes are moderate examples. These moderate examples strike a balance between simplicity and complexity, making them valuable teaching tools for both beginners looking to expand their knowledge and experienced scientists and engineers for seeking practical insights. Moderate examples often explore intermediate concepts and techniques, offering real-world relevance without overwhelming programmers with intricate (at times useless) complexities. They bridge the gap between introductory code snippets and complex, production-ready applications, making them an ideal starting point for those eager to deepen their understanding of MATLAB. Therefore, building useful applications, working with data, or optimizing our code for performance, moderate examples provide a stepping stone towards becoming a proficient Matlab developer.

10.1 Load Arrays from Strings

Loading arrays from strings is a common task in programming, often used to convert data in string format into a structured array for further processing. In various computer languages, including MATLAB, Python, C++, JavaScript and others, this operation plays a crucial role in tasks such as data parsing, data import/export, and matrix manipulation. In the examples that follow, we will explore how to load arrays from strings, demonstrating practical use cases and illustrating the process of converting raw data into a more manageable format. These examples will showcase how to parse strings and transform

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_10

them into structured arrays, unlocking the potential for a wide range of data processing applications.

10.1.1 Ex. (165) – Strings to 1D arrays (I)

```
function main_app
    a = "10|13|55|56|1|3|123";
    b = "45|33|55|0|1|22|127";

    aa = strsplit(a, '|');
    bb = strsplit(b, '|');
    cc = zeros(1, length(aa));

    for i = 1:length(aa)
        cc(i) = daniela(i, aa, bb);
    end

    disp(cc);
end

function result = daniela(i, aa, bb)
    result = str2double(aa{i}) * ...
             str2double(bb{end - i + 1});
end
```

Output:

```
1270 286 55 0 55 99 5535
```

This above program defines a series of functions and variables that perform a specific task. It begins by declaring a variable *d* and initializing it by calling the *main_app* function. The *main_app* function is defined, and it sets up two strings, *a* and *b*, which contain a series of numeric values separated by the pipe character "|". Then, it splits these strings into arrays *aa* and *bb* by using the *strsplit* function. After that, a zero filled array *cc* is declared to store the results of the calculations. The code enters a loop that iterates from *i* equal to 1 up to the length of the *aa* array. During each iteration, the *daniela* function is called with the current index *i*, the *aa* array, and the *bb* array as arguments. The result of this function call is stored in the *cc* array at the same index *i*. The *daniela* function then calculates a value by multiplying the numeric values at the current index *i* of the *aa* array with the corresponding value at the reversed index of the *bb* array (the last value of *bb* corresponds to the first value of *aa*, and so on). It then returns this calculated value. Next, and last, the *disp* function is called with the *cc* array as an argument in order to show its content to the user.

10.1.2 Ex. (166) – Strings to 1D arrays (II)

```
function main_app
    a = "10|13|55|56|1|3|123";
    b = "45|33|55|0|1|22|127";

    % Split the string by '|' to
    % create a cell array.

    aa = str2double(split(a, '|'));
    bb = str2double(split(b, '|'));
    cc = zeros(1, length(aa));

    for i = 1:length(aa)
        cc(i) = sebastian(i, aa, bb);
    end

    disp(cc);
end

function result = sebastian(i, aa, bb)
    result = aa(i) * bb(length(bb) - i + 1);
end
```

Output:

```
1270 286 55 0 55 99 5535
```

Just like before, this code example splits two strings of numbers, performs a series of multiplications on corresponding elements from these arrays, and stores the results in a new array, which is then returned by the *main_app* function and printed as *d*. However, this time some small changes can be observed. First, it defines a series of functions to perform a specific task. It starts with a *main_app* function, which is the entry point for the program. The code begins by declaring a variable *d* and assigns the result of calling the *main_app* function to it. The *main_app* function is defined next. Inside the *main_app* function, two strings *a* and *b* are defined. These strings contain a series of numbers separated by the "|" character. The *strsplit* function is used to split these strings into arrays, *aa* and *bb*, respectively. A new array *cc* is also initialized, which will be used to store the results of a computation. The code then enters a *for-loop*, iterating from 1 to the length of the *aa* array. Inside the loop, the values from the *aa* and *bb* arrays are converted from string to integers via the *sebastian* function. The *sebastian* function is called with three arguments: *i*, *aa*, and *bb*. The result of this function call is stored in the *cc* array at index *i*. Note that the *sebastian* function takes three parameters: *i*, *aa*, and *bb*, which means the passing of variables to it is made by reference. Inside, the function retrieves elements from the *aa* and *bb* arrays at specific indices, multiplies them as numbers, and returns the result. Then, the *main_app* function returns the *cc* array, which contains the results of the computations. After calling the *main_app* function, the code prints the value of *d* into the console.

10.1.3 Ex. (167) – A 2D array loaded from string

```
% Cell arrays are used when
% working with strings.

n = {};
m = {};

c = 'AAAAA|BBBBB|CCCCC|DDDDD';

n = strsplit(c, '|');

% Convert the cell array into
% a matrix of characters.

for i = 1:length(n)
    m{i} = char(n{i});
end

disp(SMC(m));

function r = SMC(m)
    r = "";
    for i = 1:length(m)
        for j = 1:length(m{i})
            r = r + m{i}(j) + " ";
        end
        r = r + newline;
    end
end
```

Output:

```
A A A A A
B B B B B
C C C C C
D D D D D
```

Here, this code splits a string into a 2D matrix and prints the matrix, where elements are separated by spaces, and rows are separated by newline characters. The code begins by declaring two empty cell arrays, *n* and *m*, where *n* is meant to hold a string split into parts, and *m* will represent a matrix. The string *c* is defined as:

$$\text{c} = \text{“AAAAA|BBBBB|CCCCC|DDDDD".}$$

Then *c* is split into parts using the *strsplit* (‘l’) function, with the results stored in the *n* array. The code proceeds with a *for-loop* to iterate through the elements in the array *n*. During each iteration, the elements in *n* are further split into individual characters via the *char* function, and stored in array *m*, effectively creating a 2D array or matrix. After setting up the *m* matrix, the code calls a function named *SMC* with *m* as an argument and prints the result. The *SMC* function is defined below. This function iterates through the rows and columns of the *m* matrix, building a string *r* that represents the contents of the matrix with space-separated elements in rows and newline characters separating rows. The resulting string is returned from the *SMC* function, and this final string is printed for user inspection.

10.1.4 Ex. (168) – Load a matrix from a string by using two delimiters

```
function main
    c = '1,2,4,1,0|3,5,6,7,8|1,2,3,4,5|5,4,3,2,1';
    m = Bahdir(c);
    fprintf(SMC(m));
end

function m = Bahdir(c)
    n = split(c, '|');

    % Preallocate m for efficiency.
    % MATLAB creates warnings if
    % preallocation is not made for
    % variables with a growing content.

    m = zeros(length(n), length(split(n{1}, ',')));

    for i = 1:length(n)
        temp = split(n{i}, ',');
        for j = 1:length(temp)
            m(i, j) = str2double(temp{j});
        end
    end
end

function r = SMC(m)
    r = "\n";
    for i = 1:size(m, 1)
        for j = 1:size(m, 2)
            r = r + " " + m(i, j) + " ";
        end
        r = r + "\n";
    end
end
```

```
Output:

 1  2  4  1  0
 3  5  6  7  8
 1  2  3  4  5
 5  4  3  2  1
```

This implementation defines a series of variables and functions for manipulating and printing a matrix represented as a string of numbers separated by commas and pipe symbols. Note that the use of the previous example for splitting numbers found in string format, may create some issues. That is, two or three digit numbers would be counted as separate columns and an error may occur. Here, the goal is to be able to store multi-digit numbers in a string if necessary. The code starts by declaring two empty arrays, *n* and *m*, which will be used to store the matrix data. The string *c* is initialized with a specific set of numbers separated by commas and pipe symbols, which represents a matrix. The numbers are organized in rows and columns, with rows separated by pipe symbols and columns separated by commas. The *Bahdir* function is defined, which takes the *c* string as its parameter. This function is responsible for parsing the string and converting it into a two-dimensional array, which represents the matrix. It does this by first

splitting the input string *c* into an array of rows using the *split*(*c*, '|') function. Then, it iterates through each row and further splits each row into an array of numbers by using the *split*(*n*{*i*}, ',') function. Next, it converts the numbers from strings to numbers using the *str2double*() function and stores the resulting two-dimensional array in the *m* variable. The *m* array is returned by the function. Next, the *SMC* function is defined, which takes a two-dimensional array *m* as its parameter (just like in the previous example). Thus, this function is responsible for formatting the matrix and returning it as a string. It initializes a string variable *r* with a newline character to create a new line at the beginning of the output. It then iterates through the rows and columns of array *m* using nested *for-loops*, and for each element in the matrix, it appends the element to the *r* string, surrounded by spaces. After each row is processed, a newline character is added to the *r* string to start a new line. The final formatted matrix is stored in the *r* variable, and it is returned as a string. Thus, the code calls the *Bahdir* function to parse the *c* string and convert it into a two-dimensional array and then passes the result to the *SMC* function to format and print the matrix to the console. One can see the use of the *strsplit* function in the previous example and the *split* function used here. Of course, one may ask: what is the difference between the two? The *strsplit* and *split* are both functions in MATLAB used for dividing strings into parts, but they have some differences in their usage and behavior. The *strsplit* function is designed to split a character vector or string scalar based on a delimiter, which can be specified. It works well with both character arrays and string arrays, and it is particularly useful when dealing with data where the delimiter is consistent and a split of the entire string is needed. On the other hand, function *split* is more recent and is specifically tailored for working with string arrays introduced in MATLAB R2016b. It splits strings at whitespace by default (just like in *JavaScript*, for instance), but delimiters can also be specified. Function *split* is more flexible when dealing with string arrays and allows for more sophisticated text manipulation, like splitting at multiple different delimiters or using regular expressions. While both functions are used for splitting strings, *strsplit* is a more general-purpose function suitable for a wide range of applications, and *split* is more specialized for advanced string manipulation, particularly with the newer string array data type.

10.1.5 Ex. (169) – A function to correctly display a matrix

```
function main
    m = [
        20, 4, 60;
        39, 5, 60;
        3, 50, 40
    ];

    fprintf(SMC(m));
end

function r = SMC(m)
    r = '\n';
    for i = 1:size(m, 1)
        for j = 1:size(m, 2)

            % Append the matrix element with
            % padding using square brackets.

            r = [r, int2str(m(i,j)), ps(m(i,j), 3)];
        end
        r = [r, '\n'];
    end
end

function t = ps(a, s)

    % Calculate the padding. Use
    % space character for padding.

    b = s - mod(length(int2str(a)), s);
    t = repmat(' ', 1, b);
end
```

```
Output:

20 4  60
39 5  60
3  50 40
```

The above example answers the following question: How can an array be elegantly displayed if the values in the array elements contain a different number of digits? Thus, the purpose of this code is to format the matrix *m* and return a string representation of it with each element properly spaced, ensuring a uniform appearance across all columns. The *ps* function aids in achieving this formatting by adding the necessary spaces. But how? At this point it is clear from the last two examples that the *SMC* function is arranging the matrix. However, this arrangement alone, does not take into account the presence of more than two digits in the array elements, which may lead to an erroneous display. For a more proper display, the *ps* function is called from inside *SMC*. Function *ps* takes two arguments: *a* (a number) and *s* (the desired width). It calculates the number of spaces required to make the number *a* fit within the specified width *s*. The implementation does this by finding the difference between *s* and the number of characters in the string representation of *a* and then generating a string consisting of that number of spaces. Therefore, the function returns this space-filled string, which is properly added to the content of the *r* variable, that in turn is printed to the console for user inspection.

10.1.6 Ex. (170) – A function to load and display matrices

```
c1 = ['12,2,44,1,0|34,5,6,7,8|' ...
      '1,2,3,4,5|5,4,3,2,1'];

c2 = ['66,5,45,10,10|37,50,60,17,18|' ...
      '10,25,37,4,5|5,4,3,2,1'];

c3 = ['66,5,45,10,10|37,50,60,17,18|' ...
      '10,25,37,4,5|5,4,3,2,1'];

disp(SMC(loadData(c1)));
disp(SMC(loadData(c2)));
disp(SMC(loadData(c3)));

function m = loadData(c)
    n = split(c, '|');
    m = cell(size(n,1), 1);
    for i = 1:length(n)
        m{i} = cellfun(@str2num, split(n{i}, ','));
    end
end

function r = SMC(m)
    r = "";
    for i = 1:length(m)
        for j = 1:length(m{i})
            r = r + m{i}(j) + padSpace(m{i}(j), 3);
        end
        r = r + newline;
    end
end

function t = padSpace(a, s)
    b = mod(s - length(num2str(a)), s);
    t = repmat(' ', 1, b);
end
```

Output:

```
12 2  44 1  0
34 5  6  7  8
1  2  3  4  5
5  4  3  2  1

66 5  45 10 10
37 50 60 17 18
10 25 37 4  5
5  4  3  2  1

66 5  45 10 10
37 50 60 17 18
10 25 37 4  5
5  4  3  2  1
```

This MATLAB code is designed to load and process numerical data represented as comma-separated values in a specific format, and then format the data as a string with specific spacing between the values, ready for display. In other words, this is a combination between the previous example and the loading of matrices from strings. The code starts by declaring three variables, *c1*, *c2*, and *c3*, each containing a string of comma-separated values arranged in rows separated by vertical bars/pipes ("|"). These strings are essentially representing numerical matrices, and there are three such matrices. First, the *load* function is defined. This function takes a string as input and splits it into a two-dimensional array where the values are separated by commas and rows are separated by vertical bars (i.e., pipes;"|"). Each element in the array is then converted to a numeric value. The function returns this two-dimensional array. The *SMC* function is also defined. It takes a two-dimensional array as input and processes it to format the data in a specific way. Next, it iterates through each element of the array, converting the values to strings and padding them with spaces to ensure each value is a fixed length of 3 characters. The new values are then concatenated into a string with each row separated by a newline character. The formatted string is returned. Also, the *ps* function is defined, which is called from inside the *SCM* function. It takes a numeric value and a desired string length (*s*) as input. It calculates the number of spaces required to pad the value to the specified length and returns a string containing those spaces. After defining these functions, the code proceeds to call the *load* function on each of the *c1*, *c2*, and *c3* strings. The resulting arrays are then passed to the *SMC* function, and the output is printed to the console window.

10.1.7 Ex. (171) – Load two matrices from strings and make the addition

```
c1 = ['12,2,44,1,0|34,5,6,7,8|' ...
      '1,2,3,4,5|5,4,3,2,1'];

c2 = ['66,5,45,10,10|37,50,60,17,18|' ...
      '10,25,37,4,5|5,4,3,2,1'];

m1 = loadData(c1);
m2 = loadData(c2);

disp(SMC(m1));
disp(SMC(m2));

sm = cell(size(m1));
for i = 1:length(m1)
    sm{i} = zeros(size(m1{i}));
    for j = 1:length(m1{i})
        sm{i}(j) = m1{i}(j) + m2{i}(j);
    end
end

disp(SMC(sm));

function m = loadData(c)
    n = split(c, '|');
    m = cell(size(n,1), 1);
    for i = 1:length(n)
        m{i} = cellfun(@str2num, split(n{i}, ','));
    end
end

function r = SMC(m)
    r = "";
    for i = 1:length(m)
        for j = 1:length(m{i})
            r = r + m{i}(j) + padSpace(m{i}(j), 3);
        end
        r = r + newline;
    end
end

function t = padSpace(a, s)
    b = mod(s - length(num2str(a)), s);
    t = repmat(' ', 1, b);
end
```

Output:

```
12 2  44 1  0
34 5  6  7  8
1  2  3  4  5
5  4  3  2  1

66 5  45 10 10
37 50 60 17 18
10 25 37 4  5
5  4  3  2  1

78 7  89 11 10
71 55 66 24 26
11 27 40 8  10
10 8  6  4  2
```

This code example makes use of the previous examples, namely it loads two matrices from strings, calculates their sum element-wise, and prints the original matrices and their sum as formatted strings. The code begins by declaring two strings, *c1* and *c2*, which represent two matrices with rows and columns separated by pipe characters ("|"). Then, two empty arrays, *m1* and *m2*, are declared and populated with the matrices loaded from the respective strings using the *load* function. Next, a zero filled array *sm* is declared to store the sum of the two matrices. The *disp* function is used to print the string representation of the matrices *m1* and *m2* after they are loaded using the *SMC* function, which formats the matrices into strings. A nested *for-loop* is used to iterate over the elements of *m1* and *m2* and calculate the sum of corresponding elements, storing the result in the *sm* matrix. The *SMC* function is defined to format a matrix as a string. It iterates over the elements of the matrix, converts them to strings, and pads them with spaces to ensure consistent column alignment. As before, the *ps* function is called from inside *SMC* function and it represents a utility function that pads a number with spaces to make it a specific length.

10.2 Some Matrix Operations

As specified before, matrix operations play a fundamental role in various fields of mathematics, science, and computer science specifically. In order to streamline and modularize the process of performing these operations, it is common practice to store them within functions. These functions serve as reusable building blocks that simplify code, improve readability, and enhance the maintainability of programs that involve matrices. In this context, we will explore the concept of matrix operations stored in functions and their significance in solving complex mathematical and computational problems efficiently. This approach not only promotes code organization but also facilitates the reuse of these operations in different parts of a program, making it a valuable practice in both algorithm development and software engineering.

10.2.1 Ex. (172) – Function to swap diagonal of matrix

```
% Function to swap diagonal of matrix

a = [
    3, 1, 2;
    1, 0, 1;
    2, 1, 3
    ];

a = swapDiagonal(a);
fprintf(SMC(a));

function a = swapDiagonal(a)
    n = size(a, 1);
    for i = 1:n
        t = a(i, i);
        a(i, i) = a(i, n - i + 1);
        a(i, n - i + 1) = t;
    end
end

function r = SMC(m)
    r = "\n";
    [rows, cols] = size(m);
    for i = 1:rows
        for j = 1:cols
            r = r + " " + m(i, j) + " ";
        end
        r = r + "\n";
    end
end
```

Output:

```
2  1  3
1  0  1
3  1  2
```

This code swaps the diagonals of a matrix and then prints the modified matrix using the custom *SMC* function. First, the *swapDiagonal* function is defined. It takes a matrix a as an argument. Within this function, it calculates the dimension n of the matrix and then loops through the matrix rows using a *for-loop* with the index variable i. During each iteration, it swaps the element at position (i, i) with the element at position $(i, n–i + 1)$. This effectively swaps the diagonal elements of the matrix. Next, the function called *SMC*

is defined. It takes a matrix *m* as an argument and is responsible for generating a string representation of the matrix. The function initializes an empty string *r* with a newline character. It then uses nested for-loops to iterate through the rows and columns of the matrix, building a string representation of the matrix with spaces and newline characters to separate the rows. The code then calls the *swapDiagonal* function to swap the diagonals of the *a* matrix. After the swap, it prints the matrix using the *SMC* function and the *disp* function, similar to the previous code.

10.2.2 Ex. (173) – Function to transpose a matrix

```
% Native function to transpose a matrix

a = [
    [1, 2, 3, 4];
    [5, 6, 7, 8];
    [9, 0, 2, 3];
    [4, 5, 6, 7]
    ];

a = transpose(a);
fprintf(SMC(a));

function a = transpose(a)
    [n, m] = size(a);
    for i = 1:n
        for j = i+1:m
            t = a(j, i);
            a(j, i) = a(i, j);  % swap
            a(i, j) = t;
        end
    end
end

function r = SMC(a)
    r = "\n";
    [rows, cols] = size(a);
    for i = 1:rows
        for j = 1:cols
            r = r + " " + num2str(a(i, j)) + " ";
        end
        r = r + "\n";
    end
end
```

Output:

```
1  5  9  4
2  6  0  5
3  7  2  6
4  8  3  7
```

This version of the code defines a matrix *a*, transposes it using the *transpose* function, and then prints the result of the matrix operation using the *SMC* function. The *SMC* function constructs a string representation of the matrix for display. It starts by defining a 2D array *a*, representing a matrix. The *transpose* function is then called with *a* as its argument, followed by a call to the *SMC* function with *a* as its argument, and the result is printed using a *fprintf* function. The *transpose* function accepts a 2D array *a* and performs the transpose operation on it. It calculates the number of rows *n* and columns *m* in the matrix *a*. Next, it uses two nested loops to iterate through the matrix and swap elements along the main diagonal, effectively transposing the matrix.

10.2.3 Ex. (174) – Function for rotation of a matrix by 90 degre

```
% left rotation of a matrix
% by 90 degree without using
% any extra space

a = [
    [1, 1, 1, 1];
    [2, 6, 7, 4];
    [2, 0, 2, 4];
    [2, 3, 3, 3]
];

fprintf(SMC(a));

a = transpose(a);
fprintf(SMC(a));

a = revColumn(a);
fprintf(SMC(a));

function a = transpose(a)
    [n, m] = size(a);
    for i = 1:n
        for j = i+1:m
            t = a(j, i);
            a(j, i) = a(i, j);
            a(i, j) = t;
        end
    end
end
```

```
Output:

 1  1  1  1
 2  6  7  4
 2  0  2  4
 2  3  3  3


 1  2  2  2
 1  6  0  3
 1  7  2  3
 1  4  4  3


 1  4  4  3
 1  7  2  3
 1  6  0  3
 1  2  2  2
```

```
function a = revColumn(a)
    [n, m] = size(a);

    % Loop over columns
    for j = 1:m

        % Loop over the
        % first half of rows

        for i = 1:floor(n/2)

            % Temporarily store
            % the top element.

            t = a(i, j);

            % Swap the top element
            % with the bottom element

            a(i, j) = a(n-i+1, j);

            % Replace the bottom element
            % with the temporary element

            a(n-i+1, j) = t;

        end
    end
end

function r = SMC(a)
    r = "\n";
    [n, m] = size(a);
    for i = 1:n
        for j = 1:m
            r = r + " " + num2str(a(i, j)) + " ";
        end
        r = r + "\n";
    end
end
```

The overall effect of this code is to rotate the *a* matrix by 90 degrees counterclockwise without using any extra space, and then it reverses the columns of the rotated matrix to complete the 90-degree left rotation. The code defines utility functions *transpose* and

revColumn for this purpose. The matrix *a* is defined as a 4×4 grid of numbers. The *transpose* function takes the *a* matrix and transposes it in place. It swaps elements across the main diagonal of the matrix. This is done by looping through the rows and columns, exchanging $a(i, j)$ with $a(j, i)$ for each element where i is the row index and j is the column index. This effectively transposes the matrix. After transposing the *a* matrix, the *SMC* function is called to print the matrix, showing the result of the transposition. Next, the *revColumn* function is defined to reverse the columns of the *a* matrix in place. It does this by iterating through the rows and using two pointers, j and k, to swap elements from the leftmost and rightmost columns within each row. In a final step, after reversing the columns of matrix *a*, the *SMC* function is called again to print the matrix in the output, showing the result of the column reversal.

10.3 Logical Operations

Logical operations play a fundamental role in computer science and programming, enabling the manipulation and evaluation of data through the use of binary logic. These operations form the building blocks for decision-making, data filtering, and conditional control in various software applications and systems. Common logical operations include AND, OR, NOT, XOR, and more, each serving a distinct purpose in processing and analyzing data. Understanding how to apply these operations is essential for both computer scientists, software engineers and other computer programmers. One powerful way to comprehend and implement logical operations is through simulation using functions. Functions are modular units of source code that encapsulate a specific set of tasks that can be reused. This approach promotes code reusability, readability, and simplifies debugging, making it an essential technique in the world of software development. In these implementation examples, we will explore the fundamentals of logical operations, their importance, and how they can be simulated and applied using functions. Therefore, this is an exploration of the concept of truth tables, the role of *Boolean* algebra, and of the practical examples of how functions can perform logical operations. Moreover, examples are presented and discussed with reference to the built-in functions for logical operations.

10.3.1 Ex. (175) – Logical NOT

```
%{

NOT
--------------
Input   Output

A   Q
-----
0   1
1   0

%}


disp(['1 -> ' num2str(f_not(1))]);
disp(['0 -> ' num2str(f_not(0))]);

function result = f_not(a)
    result = 1 - a;
end

%{

function result = f_not(a)
    result = mod(a + 1, 2);
end

function result = f_not(a)
    if a == 1
        result = 0;
    else
        result = 1;
    end
end

%}
```

Output:

```
1 -> 0
0 -> 1
```

The above code demonstrates a simple implementation of a NOT gate function. It begins by calling this function with two different input values (or arguments), 1 and 0, and then prints the results. The function *f_not* accepts a single argument *a*, which is the input value. It calculates the NOT operation by subtracting the input *a* from 1 and returns the result. This operation effectively inverts the input, turning 1 into 0 and 0 into 1. There are of course alternative implementations of a NOT gate that have been commented out in the code, but the primary function being used here is the one described above. Nonetheless, the alternatives include using the *modulo* operation or conditional statements to achieve the same logical NOT functionality.

10.3.2 Ex. (176) – Logical AND

```
%{

AND
----------------
Input   Output

A   B   Q
---------
0   0   0
0   1   0
1   0   0
1   1   1

%}

disp(['[1, 0] -> ' num2str(f_and(1, 0))]);

function result = f_and(a, b)
    result = a * b;
end
```

Output:

```
[1, 0] -> 0
```

The provided implementation from above defines a simple function named *f_and* that implements the logical AND operation. This function takes two input arguments, *a* and *b*, representing binary values (0 or 1). It then computes the logical AND operation between these two input values and returns the result as the output. The logical AND operation returns 1 only when both of its operands are 1; otherwise, it returns 0. This behavior is represented in a truth table that specifies the output (Q) for all possible combinations of inputs (A and B): (i) When A and B are both 0, the output Q is 0. (ii) When A is 0 and B is 1, the output Q is 0. (iii) When A is 1 and B is 0, the output Q is 0. (iv) When both A and B are 1, the output Q is 1. The code then provides an example usage of the *f_and* function with the input [1, 0], and it prints the result as "[1, 0] – > 0", which corresponds to the logical AND operation of 1 and 0, resulting in 0 (output).

10.3.3 Ex. (177) – Logical OR

```
%{

OR
----------------
Input   Output

A   B   Q
---------
0   0   0
0   1   1
1   0   1
1   1   1

%}

disp(['[1, 0] -> ', num2str(f_or(1, 0))]);

function result = f_or(a, b)
    result = (a + b) - (a * b);
end
```

Output:

```
[1, 0] -> 1
```

This code starts with a comment section that contains a truth table illustrating the logical OR operation for two binary inputs A and B. It shows the input combinations (0 and 1 for A and B) and the corresponding output Q of the logical OR operation. The comment provides a clear representation of the expected behavior of the *f_or* function that follows. The code then proceeds to define a function named *f_or*(a, b) which takes two arguments, a and b. Inside the function, it calculates the logical OR operation for the input values a and b using a mathematical expression $(a + b) - (a \times b)$. This expression effectively computes the OR operation and returns the result. Lastly, the code prints the result of calling the *f_or* function with the input values [1, 0].

10.3.4 Ex. (178) – Logical NAND (NOT AND)

```
%{

NAND
----------------
Input   Output

A   B   Q
---------
0   0   1
0   1   1
1   0   1
1   1   0

%}

disp(['[1, 1] -> ', num2str(f_nand(1, 1))]);

function result = f_nand(a, b)
    result = f_not(f_and(a, b));
end

function result = f_not(a)
    result = 1 - a;
end

function result = f_and(a, b)
    result = a * b;
end
```

Output:

```
[1, 1] -> 0
```

This implementation from above defines a set of functions to implement the NAND logic gate and other related logic gates, specifically NOT and AND gates. The code begins with a comment block that describes the truth table for the NAND gate, showing its inputs A and B and the corresponding output Q. The truth table specifies that the output Q is 1 when A and B are both 0 or when either A or B is 1. Otherwise, when both A and B are 1, the output Q is 0. The code then proceeds to define the following functions: *f_nand*(*a*, *b*), *f_not*(*a*) and *f_and*(*a*, *b*). The *f_nand*(*a*, *b*) function implements the NAND gate and takes two arguments , *a* and *b*. It returns the result of applying the NAND operation on

the inputs *a* and *b*. It does this by first calling the *f_and* function to perform the AND operation on *a* and *b*, and then passing the result to the *f_not* function to invert the result, effectively implementing the NAND operation. Described in the previous examples, the *f_not*(*a*) function implements the NOT gate and takes one argument, *a*. It returns the complement of the input *a*. If *a* is 0, it returns 1, and if *a* is 1, it returns 0. Also, the *f_and*(*a*, *b*) function implements the AND gate and takes two arguments, *a* and *b*. It returns the result of applying the AND operation on the inputs *a* and *b*. It multiplies the values of *a* and *b*, and the result is 1 only if both *a* and *b* are 1; otherwise, it is 0. The code concludes by using the defined functions to demonstrate the functionality of the NAND gate by calling *f_nand*(1, 1).

10.3.5 Ex. (179) – Logical NOR (NOT OR)

```
%{

NOR
----------------
Input   Output

A   B   Q
---------
0   0   1
0   1   0
1   0   0
1   1   0

%}

disp(['[0, 0] -> ' num2str(f_nor(0, 0))]);

function result = f_nor(a, b)
    result = f_not(f_or(a, b));
end

function result = f_not(a)
    result = 1 - a;
end

function result = f_or(a, b)
    result = (a + b) - (a * b);
end
```

Output:

```
[0, 0] -> 1
```

This code defines a set of functions that implement the NOR (NOT OR) logic gate, a basic digital logic gate with two input variables (A and B) and one output variable (Q). The NOR gate returns a true (1) output only when both of its input variables are false (0). The code begins with a comment block, providing a truth table for the NOR gate, which

lists the possible input combinations of A and B along with the resulting output Q. The *disp* function is used to display the result of the NOR gate for a specific input combination of A and B. For example, [0, 0] – > 1 indicates that when both A and B are 0, the NOR gate outputs 1. The code defines several functions to implement the NOR gate. The novel function called *f_nor*(*a*, *b*) takes two input arguments, *a* and *b*, and calculates the NOR operation by first applying the OR operation using the *f_or* function and then negating the result using the *f_not* function. The function *f_not*(*a*), showcased prior to this example, takes one input argument *a* and negates it by subtracting it from 1, effectively converting 1 to 0 and 0 to 1. Also, the function *f_or*(*a*, *b*) presented before this current example, takes two input arguments, *a* and *b*, and calculates the OR operation. It does so by adding *a* and *b* and then subtracting their product ($a \times b$). The result is 1 only if at least one of the input values (*a* or *b*) is 1; otherwise, it is 0. Thus, this code defines functions to implement the NOR gate, utilizing the concepts of NOT and OR operations to achieve the desired logic. It also provides a specific example of using the *f_nor* function to evaluate the NOR gate output for the input combination [0, 0], which naturally evaluates to 1 ([0, 0] – > 1).

10.3.6 Ex. (180) – Logical XOR

```
%{

XOR
----------------
Input   Output

A   B   Q
---------
0   0   0
0   1   1
1   0   1
1   1   0

%}

disp('[0, 0] -> ' + string(f_xor(0, 0)));

function result = f_xor(a, b)
    result = (a + b) - 2 * (a * b);

    % Alternatively, one can use either
    % of the commented out returns for
    % the same XOR operation result:

    % result = (a - b) * (a - b);
    % result = mod((a + b) * (a + b), 2);

End
```

Output:

```
[0, 0] -> 0
```

This MATLAB code defines a function that implements the XOR (exclusive OR) logic operation. XOR takes two binary inputs, A and B, and returns 1 if exactly one of them is 1, and 0 if both are the same (0 or 1). The code starts with a comment section that describes the truth table for XOR, showing the input values A and B and their corresponding output Q. It lists all possible combinations and the expected result for each combination. The code then calls the function *f_xor* to demonstrate the XOR operation on the input [0, 0] and prints the result to the console. The *f_xor* function itself implements the XOR logic using simple arithmetic operations. It takes two arguments, a and b, representing the binary inputs. The function calculates the XOR result by adding a and b together ($a + b$), which can be 0, 1, or 2. Then, subtracting twice the product of a and b from the sum. This effectively handles the XOR logic, ensuring that the result is 1 when only one of a or b is 1, and 0 when both are 0 or both are 1. Please notice that the code provides two alternative implementations as comments, which use different mathematical expressions to achieve the same XOR logic. However, the main implementation with addition and subtraction is active.

10.3.7 Ex. (181) – Logical XNOR

```
%{

XNOR
----------------
Input   Output

A   B   Q
---------
0   0   1
0   1   0
1   0   0
1   1   1

%}

disp(['[0, 0] -> ', num2str(f_xnor(0, 0))]);

function result = f_xnor(a, b)

    % Using NOT and XOR
    result = f_not(f_xor(a, b));

    % Using NOT, AND, and NOT
    % ----------------------------------------
    % result = f_not(f_and(f_not(a), b) + ...
                f_and(a, f_not(b)));
```

Output:

```
[0, 0] -> 1
```

```
    % Using NOT, OR, and NOT
    % ----------------------------------------
    % result = f_not(f_or(f_not(a), b) + ...
    %           f_or(a, f_not(b)));

    % Using NOT, OR, and multiplication
    % ----------------------------------------
    % result = f_not(f_or(a, b)) + (a * b);

    % Using NOT and algebraic manipulation
    % ----------------------------------------
    % result = f_not((a + b) - (a * b)) + (a * b);

    % Using NOT and algebraic
    % manipulation with simplification
    % ----------------------------------------
    % result = f_not((a + b) - (a * b) + (a * b));

    % Using NOT and algebraic manipulation
    % with further simplification
    % ----------------------------------------
    % result = f_not((a + b) - 2 * (a * b));
end

function result = f_xor(a, b)
    result = (a + b) - 2 * (a * b);
end

function result = f_not(a)
    result = 1 - a;
end

% The functions f_or and f_and would only be
% needed if one uncomments one of the
% alternative implementations that use them.

function result = f_or(a, b)
    result = (a + b) - (a * b);
end

function result = f_and(a, b)
    result = a * b;
end
```

This code defines functions for XNOR, XOR, and NOT logical operations and provides a simple way to calculate the XNOR of two input values. The code could be extended to include other logical operations by uncommenting and modifying the relevant functions. First, the implementation defines a set of functions to implement the XNOR (exclusive NOR) logical operation, and it also includes some related functions for other logical operations like XOR, NOT, AND, and OR. It begins with a comment section that provides a table representing the XNOR truth table, with input values A and B, and their corresponding output value Q. The XNOR operation returns 1 (true) when both A and B are the same (either both 0 or both 1), and it returns 0 (false) when A and B are different. The code then prints the result of applying the XNOR operation to the input [0, 0], which outputs 1. Next, there are several functions defined. The *f_xnor*(*a*, *b*) function is the main XNOR function. It takes two arguments, *a* and *b*, representing the input values. It computes the XNOR operation by first calling the *f_xor* function to get the XOR result and then negating it using the *f_not* function to get the final XNOR result. There are some commented-out alternative implementations that I wish the reader to inspect, namely for XNOR using AND, OR, and other logical operations. This demonstrates the myriad of possibilities of achieving the same result, and definitely worth a look. To continue, *f_xor*(*a*, *b*) function calculates the XOR operation between *a* and *b*. XOR returns 1 when the inputs are different and 0 when they are the same. It is implemented by subtracting twice the product of *a* and *b* from the sum of *a* and *b*. Also, the *f_not*(*a*) function implements the NOT operation, which negates the input *a*. It returns 1 if *a* is 0 and returns 0 if *a* is 1. As mentioned above, there are also commented functions for OR and AND, which are not used in the XNOR implementation, but can be valuable teaching examples for different lectures.

10.4 Miscellaneous

Where the lines between disciplines blur, miscellaneous codes are at the forefront, embracing ambiguity, and the ever-expanding possibilities of software engineering. They are not constrained by labels or boundaries, but rather driven by the limitless potential of technology and the boundless horizons of their own capabilities. In short, miscellaneous codes are those implementations that do not fit clearly in any category, these being considered the useful outliers of the field of science (usually). Some of the codes in this subchapter are not necessarily true outliers in any of the scientific or engineering fields, however, they are outliers for the structure of this book.

10.4.1 Ex. (182) – Logaritm of *b* in base *a*

```
a = 10; % base.
b = 2;  % value.

disp(log_custom(a, b));

function result = log_custom(n, v)
    result = log(v) / log(n);
end
```

Output:

```
0.30102999566398114
```

This code calculates and prints the logarithm of *b* to the base *a* in the console. In this Matlab code snippet, two variables are defined: *a* and *b*. Variable *a* is assigned the value 10, and it serves as the base, while variable *b* is assigned the value 2, which is the value used in the subsequent mathematical calculation. The code then invokes a function called *log_custom* and passes two arguments to it: *a* and *b*. The *log_custom* function calculates the logarithm of *b* to the base *a* using the built-in *log* function for natural logarithms. It then uses the formula for logarithm conversion by dividing the natural logarithm of *b* by the natural logarithm of *a*. The result of this calculation is then returned by the function and is printed in the output.

10.4.2 Ex. (183) – Smooth signal

```
a = [5, 1, 8, 4, 6, 2, 9, 8];

disp(smooth(a));

function b = smooth(a)
    n = length(a);
    b = a;
    for i=2:n-1
        b(i) = (a(i-1) + a(i+1))/2;
    end
end
```

Output:

```
5,6.5,5.2,5.6,3.8,6.4,7.2,8
```

This implementation uses an array *a* containing a sequence of numbers: 5, 1, 8, 4, 6, 2, 9, and 8. This array is passed as an argument to a function called *smooth*. The *smooth* function takes the array as its parameter and performs a smoothing operation on it. It first determines the length of the array and stores it in variable *n*. Then, it enters a *for-loop* that iterates over the elements of the array. However, the loop starts at the second element (index 1) and ends at the second-to-last element (index $n-1$). Inside the loop, each element at index i is updated by taking the average of the elements at indices $i-1$ and $i+1$, effectively smoothing out the values. The result of this smoothing operation is stored in the array *b*, which is then returned by the function. Thus, the code calls the *smooth* function and prints the returned array to the console.

10.4.3 Ex. (184) – Greatest common divisor (GCD)

```
% greatest common divisor (GCD).

disp(gcd(45, 12));

function result = gcd(a, b)
    if a == 0
        result = b;
        return;
    end

    while b ~= 0
        if a > b
            a = a - b;
        else
            b = b - a;
        end
    end
    result = a;
end
```

Output:

```
3
```

This calculates the greatest common divisor (GCD) of two numbers and then prints the result. It defines a function called *gcd* which takes two parameters, *a* and *b*. The code first checks if *a* is equal to 0, and if so, it returns the value of *b* as the GCD. If *a* is not zero, it enters a while loop. Inside the loop, it repeatedly subtracts the smaller of the two numbers from the larger one. This process continues until one of the numbers becomes zero. Once that happens, the GCD is found, and the result is returned. Thus, the code calls the *gcd* function with the values 45 and 12 and prints the result.

10.4.4 Ex. (185) – Pseudo random generator

```
x = 3; % seed.
disp(prandom(x));

function r = prandom(x)
    a = 11;
    m = 25;
    c = 17;
    r = "";

    for i = 1:10

      x = mod(a * x + c, m);
      r = [r, num2str(x), ","];

    end
end
```

Output:

```
0,17,4,11,13,10,2,14,21,23,
```

In this code, there is a function named *prandom* that generates a sequence of pseudo-random numbers based on a mathematical formula. The function takes an initial value x as a parameter, which is used as a seed for the random number generation. The variables a, m, and c are constants used in the formula. The function initializes an empty string r to store the generated numbers. It then enters a loop that runs 10 times. In each iteration, it updates the value of x using the formula:

$$x = (a \times x + c)\%m$$

It then appends the new value of x to the string r, separated by commas. Once all iterations are completed, the function returns the string r, which contains the sequence of generated pseudo-random numbers. The initial value of x and the constants a, m, and c determine the pattern of the generated numbers, making it pseudo-random in nature. This code essentially demonstrates a simple pseudo-random number generator using a linear congruential generator (LCG) algorithm.

10.4.5 Ex. (186) – Double brute force algorithm (DBFA)

```
% Double Brute Force Algorithm (DBFA)

x = Block_Alocation(133);
disp(x);

function m = Block_Alocation(L)

    a = 1;
    b = 1;
    t = 5; % min block length.
    m = 8; % max block length.

    while true
        a = a + 1;
        t = mod(L, a);
        r = (L - t);
        v = mod(r, 2);
        t = t + 1;
        if ~(t > 3 && v == 0)
            break;
        end
    end

    while true
        m = m + 1;
        b = mod(r, m);
        if ~(b == 0 || m > 1000)
            break;
        end
    end

    return;
end
```

Output:

```
9
```

The code implements a block allocation algorithm that involves two nested brute force loops to determine the value of m based on the input L, and the result is stored in the variable x. The example implements a *Double Brute Force Algorithm* (DBFA) for block allocation [1]. The point of the algorithm is the calculation of text chunks (blocks) of length m that divides a sequence of length L, such that in the last chunk there will be a minimum of t characters. First, this code defines a function called *Block_Alocation* that takes an input parameter L, and it also demonstrates the usage of this function. The *Block_Alocation* function begins by declaring several variables: v, r, a, b, t, and m, and initializes them with specific values. Within the first *while-loop*, it iteratively calculates values for a, t, r, and v until the condition ($t > 3$ && $v == 0$) is no longer satisfied (please notice the negation character "~" in front of the parentheses). The loop performs a series of mathematical operations on these variables based on the input value L. After the first loop exits, the second *while-loop* is initiated. This loop calculates values for m and b iteratively based on the calculated value of r from the previous loop. It continues until the condition ($b == 0 \,||\, m > 1000$) is no longer met. The final result is the value of m, which is returned as the output of the *Block_Alocation* function. The main part of the code then initializes a variable x with the result of calling the *Block_Alocation* function with the argument 133, and it prints the value of x to the console window for user inspection.

10.4.6 Ex. (187) – Alphabet detection

```
disp(alpha('uiuhd87wqsaidhsad'));

function a = alpha(c)

    % Convert string
    % to char array.
    t = char(c);

    % Empty array.
    a = [];

    % Get the length of
    % the char array.
    k = length(t);

    for i = 1:k

        % flag variable.
        q = 1;

        % Check against
        % the length of a.

        for j = 1:length(a)
            if t(i) == a(j)

                % Set flag to 0 if
                % duplicate is found.
                q = 0;

                % Exit inner loop
                % if a match is found.
                break;
            end
        end
        if q == 1

           % Append unique
           % character to a.
            a = [a t(i)];
        end
    end
 end
```

Output:

```
u,i,h,d,8,7,w,q,s,a
```

Alphabet detection is an algorithm that identifies the unique characters from a sequence of text (ex. input: “ABBBABBACABBA”; output: “ABC”). The code defines a function named *alpha* that takes a single argument *c*, which is expected to be a string. Thus, the purpose of this function is to detect unique characters in the input string and return them as an array. Within the function an empty array *a* is initialized. This array will be used to store unique characters from the input string. The input string *c* is split into an array of individual characters and stored in the variable *t*. The length of the array *t* is stored in the

variable *k*. Next, there are two nested loops used to identify and store unique characters in the array *a*. The outer loop iterates over the indices of array *t*, from 1 to *k* (inclusive). Inside the outer loop, a variable *q* is initialized to 1 (flag variable). The inner loop iterates over the indices of the array *a*, from 1 to the current length of *a*. Within the inner loop, the code checks if the current character in *t* at index *i* is equal to any character in the array *a*. If it is, *q* is set to 0, indicating that the character is not unique. If *q* remains 1 after the inner loop, it means that the character is unique, and it is pushed to the array *a*. After both loops have completed, the function returns the array *a*, which contains all the unique characters from the input string. The *alpha* function is invoked with the argument sequence "uiuhd87wqsaidhsad", and the result is printed to the console window for inspection.

10.4.7 Ex. (188) – Alphabet detection on matrices

```
c = [
    [ 1, 1, 1, 1, 1, 1, 0, 1, 1, 1 ];
    [ 1, 0, 1, 0, 1, 1, 1, 0, 1, 1 ];
    [ 1, 1, 1, 0, 1, 1, 0, 1, 0, 1 ];
    [ 0, 1, 0, 0, 1, 1, 1, 0, 0, 1 ];
    [ 1, 1, 1, 0, 1, 1, 1, 0, 1, 0 ];
    [ 1, 0, 1, 1, 1, 1, 0, 1, 0, 0 ];
    [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 ];
    [ 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 ];
    [ 1, 1, 0, 0, 0, 0, 1, 0, 0, 1 ]
];

unique_elements = matrix_alphabet(c);
disp(unique_elements);

function a = matrix_alphabet(t)
    a = [];         % Holds unique elements.
    n = size(t, 1); % Number of rows.
    m = size(t, 2); % Number of columns.

    for i = 1:n
        for j = 1:m
            q = 1;

            for k = 1:numel(a)
                if t(i,j) == a(k)
                    q = 0;
                    break;
                end
            end
            if q == 1
                a = [a t(i,j)];
            end
        end
    end
end
```

Output:

```
1,0
```

The provided code defines a two-dimensional array *c* representing a binary matrix and a function named *matrix_alphabet* that extracts unique elements from this matrix (mainly just like in the previous example but this time for a 2D structure). The implementation calls the function *matrix_alphabet* with the matrix *c* as an argument and prints the result. The *c* array is a 2D array with 9 rows and 10 columns, containing binary values (i.e., 0 or 1) that likely can represent some sort of pattern or alphabet. The *matrix_alphabet* function takes a 2D array *t* as an argument and aims to find the unique elements from it. It initializes an empty array *a* to store these unique elements and calculates the dimensions of the input matrix *t* with *n* representing the number of rows and *m* representing the number of columns. The function then iterates through each element of *t* and checks if it is already in the array *a*. If not, it adds the element to *a*. The function returns the array *a*, which contains the unique elements from the input matrix. The code contains a *disp* statement at the end to display the result of calling the *matrix_alphabet* function with the *c* matrix as an argument. Note that this time, the detection of unique characters was made over a matrix (2D), instead of a sequence (1D), as it was done in the previous example.

10.5 Sorting

Sorting is a fundamental concept in computer science and plays a crucial role in a wide range of applications, from data organization to optimization and search algorithms. At its core, sorting involves the arrangement of elements in a specific order, typically in ascending or descending order. This seemingly simple task has far-reaching implications, as efficiently organized data allows for faster search and retrieval, facilitates data analysis, and enhances the overall performance of various algorithms and systems. The need for sorting arises in diverse fields, from databases and information retrieval systems to scientific computing and everyday tasks like organizing files and lists. Sorting is also a key component in numerous computational problems, such as searching for a specific item in a dataset, identifying duplicates, or solving optimization problems. In this subchapter, we will examine different sorting algorithms, their characteristics, and their real-world applications.

10.5.1 Ex. (189) – Low level native sort and eliminate duplicates (I)

```
b = [3, 6, 2, 78, 99, 1, 4];
n = length(b);

% Initialize array a with
% zeros up to the maximum
% value in b.

a = zeros(1, max(b));

% Fill a using values
% from b as indices.

for i = 1:n
    a(b(i)) = b(i);
end

% Compact a back into b,
% effectively sorting it r = 1;

for j = 1:length(a)
    if a(j) ~= 0
        b(r) = a(j);
        r = r + 1;
    end
end

% Truncate b to the new
% size r - 1 to remove
% trailing zeros.

b = b(1:r-1);

disp(b);
```

```
Output:

1,2,3,4,6,78,99
```

The example initializes two arrays, a and b, with a initially being an empty array and b containing some numerical values. It then defines variable n. Variable n is assigned the length of array b. The code proceeds to enter a loop that iterates through the elements of array b using a *for-loop*, where i serves as the loop counter. Inside the loop, it assigns the value of $b(i)$ to $a(b(i))$. This effectively creates a new array a where the indices correspond to the values of b, and the values in a are the same as the corresponding values in b. A second *for-loop* begins, with j as the loop counter. This loop iterates through the indices of the array a, starting from 1 up to the length of array a. Inside this loop, it checks if there is a non-falsy value at index j in array a. If a non-falsy value is found, it assigns that value to $b(r)$ and increments the value of r. Next, the code prints the contents of array b. Note that this native sorting method works well for number sequences of short ranges and is written for this book. To my knowledge it is not published anywhere but here. Note that the time spent by this method to sort the values of an array is $n + m$,

where m represents the maximum value among the values found in the elements of the array.

10.5.2 Ex. (190) – Low level native sort and eliminate duplicates (II)

```
b = [3, 6, 2, 78, 99, 1, 4];
n = length(b);
r = n;

% Fill a with zeros.
a = zeros(1, max(b));

% Populate array a
% with elements from b.

for i = 1:n
    a(b(i)) = b(i);
end

m = length(a);

% Iterate backwards
% through a to sort b.

for j = 1:m
    if a(j) ~= 0

        % Place the non-zero
        % values in b.

        b(r) = a(j);
        r = r - 1;
    end
end

disp(b);
```

Output:

```
99,78,6,4,3,2,1
```

How about sorting the values from maximum to minimum? Well, we can simply reverse the output order of the previous result. But, let us write something more elegant than a simple inversion. Like before, this example initializes two arrays, a and b. Array b is assigned a set of numeric values. The code also initializes two variables, r and n, with n as the length of array b and with r set to the value of n. The first part of the method is the same as in the previous example. Namely the code then enters a *for-loop* that iterates from 1 to n, and in each iteration, it assigns the value of $b(i)$ to the corresponding index in array a. This operation effectively populates array a with values from array b at the same indices. Next, the code calculates the length of array a and stores it in the variable m, representing the maximum value in the array. In the second part, the code then enters another *for-loop*, this time iterating from 1 to m. Inside this loop, it checks if the value at index j in array a exists (i.e., is not falsy, or in other words the element is

not empty), and if it does, it assigns that value to the corresponding index in array *b* at index *r* and increments the value of *r*. This operation essentially filters out falsy values from array *a* and stores them in array *b*. Next, the code prints the modified array *b* to the console window. Note that this native sorting method works well for number sequences containing small maximum values (ex. 100). Again, to my knowledge this method is not published anywhere but here. Both examples (the current and the previous one) showcase how indexing and iterating strategies can be manipulated to achieve different sorting behaviors, demonstrating the flexibility and power of MATLAB for array manipulations.

10.5.3 Ex. (191) – An optimized version of *Bubble Sort*

```
a = [4, 5, 8, 1, 1, 5, 2, 9];
sorted_a = bs(a);
disp(sorted_a);

function a = bs(a)
    n = length(a);
    for i = 1:n-1
        for j = 1:n-i
            if a(j) > a(j+1)  % Swap
                t = a(j);
                a(j) = a(j+1);
                a(j+1) = t;
            end
        end
    end
end
```

Output:

```
1,1,2,4,5,5,8,9
```

There are a couple of sorting algorithms. However, this code demonstrates a simple implementation of the *Bubble Sort* algorithm to sort an array *a* in ascending order. The array *a* is defined with a set of unsorted numeric values. The code defines a function *bs*(*a*) that takes an array *a* as a parameter and performs the *Bubble Sort* algorithm. Inside the function it initializes variables *i*, *j*, *n*, and *t*. Variable *n* is set to the length of the input array *a*. It uses two nested loops to iterate through the array to compare adjacent elements. If the element at index *j* is greater than the element at index $j + 1$, it swaps the elements to sort them in ascending order. The *t* variable is used for temporary storage during the swap. Also, the outer loop (*i*) iterates *n*–1 times, and the inner loop (*j*) iterates *n*–*i* times, as the largest elements have already bubbled to the end of the array during each pass of the outer loop. Overall, the sorted array *a* is returned by the function. Thus, the code calls the *bs*(*a*) function with the array *a* and prints the resulting sorted array to the console for user inspection.

10.6 Permutations

Permutations are a fundamental concept in mathematics and combinatorics. They are at the heart of numerous real-world problems and are essential for understanding the possibilities that exist within various sets of objects. In short, permutations refer to the arrangement of all or part of a set of objects, with the order being important. For instance, the permutations of the letters A, B, C are ABC, ACB, BAC, BCA, CAB, and CBA. In each permutation, the order of the items is significant. Combinations, on the other hand, are selections of items from a set where the order does not matter. For example, the combinations of two letters chosen from A, B, C are AB, AC, and BC. Here, AB and BA are considered the same combination, as the order is irrelevant. Thus, permutations and orderings focus on sequences where the arrangement is key, while combinations are about selecting items from a set without a regard to order.

10.6.1 Ex. (192) – Get all permutations of a given string (I)

```
function permuteAll()
    s = 'ACTG';
    n = length(s);
    a = permute(s, n, 1, {});
    disp(a);
end

function a = permute(s, r, l, a)
    if l == r

        % Append new permutation.
        a{end+1} = s;

    else
        for i = l:r
            s = swap(s, l, i);

            % Recursively call permute.
            a = permute(s, r, l+1, a);

            % Swap back, backtrack.
            s = swap(s, l, i);

        end
    end
end

function s = swap(s, i, j)
    c = s(i);
    s(i) = s(j);
    s(j) = c;
end
```

Output:

```
ACTG,
ACGT,
ATCG,
ATGC,
AGTC,
AGCT,
CATG,
CAGT,
CTAG,
CTGA,
CGTA,
CGAT,
TCAG,
TCGA,
TACG,
TAGC,
TGAC,
TGCA,
GCTA,
GCAT,
GTCA,
GTAC,
GATC,
GACT
```

This code generates all possible permutations of a given string and stores them in an array *a*. It follows a recursive approach to generate these permutations. The code starts by initializing an empty array *a* to store the permutations. Next, there is a function *permute* that takes four arguments: *s* (the string to be permuted), *r* (the right index), *l* (the left index), and array *a*. Next, it checks if *l* is equal to *r*, which means that the string is fully permuted. If so, it adds the permuted string *s* to the array *a*. If *l* is not equal to *r*, the function enters a loop that iterates from *l* to *r*. Inside the loop, it swaps characters in the string *s* and recursively calls *permute* with the updated string to permute the remaining characters. After the recursive call, it swaps the characters back to their original positions. There is also a swap function that takes a string *s* and two indices *i* and *j*. It converts the string to an array, swaps the characters at indices *i* and *j*, and then converts the array back to a string. The code defines a string *s* with the initial value "ACTG" and calculates its length *n*. Overall the implementation calls the *permute* function with the string *s*, the right index *n*, the left index 1 to start the permutation process, and an empty array. Once all permutations are generated the content of array *a* is shown to the console window for user inspection.

10.6.2 Ex. (193) – Get all permutations of a given string (II)

```
s = 'ABC';
a = '';
b = {};
b = permute(s, a, b);
disp(b);

function b = permute(s, a, b)
    if isempty(s)
        b{end+1} = a;
    else
        for i = 1:length(s)
            c = s(i);
            l = s(1:i-1);   % left part
            r = s(i+1:end); % right part
            q = [l, r];
            b = permute(q, [a, c], b);
        end
    end
end
```

Output:

```
ABC,
ACB,
BAC,
BCA,
CAB,
CBA
```

Here, there is an array *b* initialized as an empty array. The code defines a function named *permute*, which takes three arguments, *s*, *a* and *b*. The purpose of this function is to generate all permutations of a given string *s*. It does this by recursively permuting the characters of the string and collecting the permutations in the array *b*. Within the permute function a few statements allow for the permutations. If the input string *s* is empty, it means a permutation has been successfully formed, and it pushes the current permutation *a* into the *b* array. Otherwise, it iterates over the characters in the string *s*. For each

character *c* at index *i*, it splits the string into two parts, *l* (the characters to the left of *c*) and *r* (the characters to the right of *c*). Also, it constructs a new string *q* by combining *l* and *r*, effectively removing *c* from the string. Also, it calls the *permute* function recursively with the modified string *q* and the current permutation: *a* concatenated with *c*. After defining the *permute* function, the code initializes a string *s* with the value "ABC" and an empty string *a*. Then, it calls the *permute* function with these values, effectively generating all permutations of the string "ABC" and collecting them in the *b* array. Next, the code prints the array *b* to the console for the user inspection, which contains all the permutations from "ABC".

10.7 Statistics

Statistics is a discipline that lies at the heart of understanding and interpreting data. It is the science of collecting, organizing, analyzing, interpreting, and presenting data to gain insights and make informed decisions. In a world inundated with information, statistics serves as a crucial tool for both scientists and decision-makers, allowing us to extract meaningful patterns and knowledge from the vast array of data that surrounds us. Statistics is not just a collection of mathematical techniques; it is a powerful way of thinking, one that permeates various fields, from science and social sciences to economics and business. Statistics is everywhere, from mathematics to biology [2–4]. Whether it is predicting trends, testing hypotheses in science, or making informed policy decisions, statistics provides the framework for evidence-based reasoning. It helps us answer questions, validate assumptions, and draw meaningful conclusions from raw data, contributing to the advancement of knowledge and informed decision-making. Understanding statistics is not only beneficial for researchers but also for everyday individuals looking to navigate the increasingly data-driven world effectively, which the modern man is bound to do.

10.7.1 Ex. (194) – Return an array with proportions (relative frequencies)

```
a = [5, 1, 8, 4, 6, 2, 9, 8];

disp(p(a));

function t = p(a)

    max_val = max(a);
    n = length(a);
    m = 100;          % Set the scale factor
    t = zeros(1, n); % Fill array with zeros

    for i = 1:n

        % Scale each element
        t(i) = (m / max_val) * a(i);
        t(i) = round(t(i));

    end

    t = cellstr(num2str(t')) + "%";
end
```

```
Output:

56%,11%,89%,44%,
67%,22%,100%,89%
```

This code starts by defining an array named *a*, which contains a list of numeric values. Next, it calls a function named *p* with *a* as an argument and prints the result. In other words, the *p* function takes an array *a* as its parameter and performs a number of steps. First, it calculates the maximum value in the input array *a* using the *max*(*a*) function and stores it in a variable named *max_val*. Next, it determines the length of the input array *a* and stores it in a variable named *n*. Also, it initializes a variable *m* with the value 100. Next, it creates a zero filled array *t* to store the transformed values. The function enters a loop that iterates over each element in the input array *a*. Inside the loop, it calculates a new value for each element in *t*. The new value is calculated by scaling the original value (*a*(*i*)) by the ratio of *m* to *max_val*. Then it rounds the result to the nearest integer and appends the "%" sign to it before storing it in the array *t*. The function returns the array *t*, which contains the transformed values of the input array *a*, expressed as percentages. Variable *t* is then printed in the console.

10.7.2 Ex. (195) – Average, standard deviation and coeficient of variation

```
a = [5, 1, 8, 4, 6, 2, 8, 9];

b = stat(a);
disp(b);

function r = stat(a)
    n = length(a);
    b = 0;
    e = 0;

    % Initialize the results array
    r = zeros(1, 3); % AV, SD, CV

    % Calculate the sum of
    % the array elements.

    for j=1:n
        b = b + a(j);
    end

    % Calculate the arithmetic
    % mean (AV).

    r(1) = b/n;

    % Calculate the standard
    % deviation (SD).

    for j=1:n
        e = e + (a(j) - r(1))^2;
    end

    r(2) = sqrt(e/(n-1));

    % Calculate the coefficient
    % of variation (CV).

    r(3) = r(2)/r(1);
end
```

```
Output:

5.375,
2.9246489410818914,
0.5441207332245379
```

The above example performs statistical calculations on an array *a* and then calls the *stat* function with the array. The code starts by defining an array *a* with a list of numerical values. Then, the code initializes a variable *b* to 0 and another variable *e* to 0. Additionally, it creates an array *r* with three elements for storing statistical results, namely for the average (AV; mathematically denoted as $\overline{x}$), standard deviation (SD; mathematically denoted as σ), and the coefficient of variation (CV; mathematically denoted as C_v):

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{\sum_{i=1}^{n} a(i)}{n} = \frac{b}{n} = r(1)$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}} = \sqrt{\frac{\sum_{i=1}^{n} (a(i) - r(1))^2}{n-1}} = \sqrt{\frac{e}{n-1}} = r(2)$$

$$C_v = \frac{\sigma}{\overline{x}} = \frac{r(2)}{r(1)} = r(3)$$

For the expressions shown above, please observe the progressive replacement and correlation with the variables from the code. The *for-loop* iterates through each element in the array *a* to calculate the sum of all values, which is stored in variable *b*. Next, the average (AV) is calculated by dividing the sum *b* by the total number of elements in the array (*n*), and the result is stored in *r*(1). The code then proceeds to calculate the sum of squared differences from the average (*e*) for each element in the array. This step is essential for calculating the standard deviation (SD). The standard deviation (SD) is calculated as the square root of the sum of squared differences from the average, divided by (*n*–1). The result is stored in *r*(2). The coefficient of variation (CV) is calculated as the ratio of the standard deviation to the average (*r*(1)/*r*(2)). The *r* array, which now contains the calculated statistical values, is returned by the *stat* function. Thus, the code prints the result, which is the *r* array returned by the *stat* function to the console window by using the *disp* function. Note that the *stat* function is essentially used to compute and return statistical information about the input array *a*, including the average, standard deviation, and coefficient of variation.

10.7.3 Ex. (196) – Pearson correlation coefficient

```
a = [6, 8, 10];
b = [12, 10, 20];

disp(p(a,b));

function r = p(a, b)
    n = length(a);
    m = zeros(1,2);

    for i = 1:n
        m(1) = m(1) + a(i);
        m(2) = m(2) + b(i);
    end

    m(1) = m(1) / n; % mean a.
    m(2) = m(2) / n; % mean b.

    s0 = 0;
    s1 = 0;
    s2 = 0;

    for i = 1:n
        s0 = s0 + (a(i)-m(1)) * (b(i)-m(2));
        s1 = s1 + (a(i)-m(1))^2;
        s2 = s2 + (b(i)-m(2))^2;
    end

    r = s0 / sqrt(s1 * s2);
end


% Also, a MATLAB built-in version would be:

a = [6, 8, 10];
b = [12, 10, 20];

m = corrcoef(a, b);
r = m(1,2);

disp(r);
```

Output:

```
0.7559
```

Here, the code begins by initializing two arrays, a and b, each containing three numeric values. The purpose of this code is to calculate the *Pearson correlation coefficient* between these two arrays, which is a statistical measure of the linear relationship between two datasets. The core of the calculation is performed in the p function. Inside this function, the length of the arrays is stored in the variable n. Variable m is initialized as an array to store the intermediate values during the calculation. First, the means (averages) of arrays a and b are computed. The sum of all values in a is accumulated in $m(1)$, and the sum of all values in b is accumulated in $m(2)$. These sums are then divided by n to calculate the mean of each array. The next step involves calculating the *Pearson correlation coefficient*,

namely r:

$$r = \frac{\sum_{i=1}^{n}[(x_i - \overline{x})(y_i - \overline{y})]}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2} \times \sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}$$

where x_i is a sample from the first data set, y_i is the sample from the second data set, $\overline{x}$ is the mean of the values from the first data set, $\overline{y}$ is the mean of the values from the second data set, and finally n is the total number of samples from either data set (because they are equal). The formula for r looks complicated, however, the code will show the reader a different story. Thus, the above mathematical formula is computed by using three accumulators: *s0*, *s1*, and *s2*. Variable *s0* accumulates the sum of the products of the differences between each element of a and b from their respective means. On the other hand, variable *s1* accumulates the sum of the squared differences of each element in a from its mean, and *s2* accumulates the sum of squared differences for each element in b from its mean. With these intermediate values, the *Pearson correlation coefficient* (r) is computed as the ratio of *s0* divided by the square root of the product of *s1* and *s2*:

$$r = \frac{s0}{s1 \times s2}$$

$$s0 = \sum_{i=1}^{n}[(x_i - \overline{x})(y_i - \overline{y})] = \sum_{i=1}^{n}[(a(i) - m(1))(b(i) - m(2))]$$

$$s1 = \sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2} = \sqrt{\sum_{i=1}^{n}(a(i) - m(1))^2} = \sqrt{\sum_{i=1}^{n}(a(i) - m(1))^2} = sqrt\left(\sum_{i=1}^{n}(a(i) - m(1))^2\right)$$

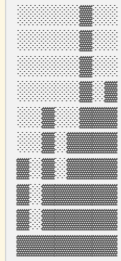
$$s2 = \sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2} = \sqrt{\sum_{i=1}^{n}(b(i) - m(2))^2} = \sqrt{\sum_{i=1}^{n}(b(i) - m(2))^2} = sqrt\left(\sum_{i=1}^{n}(b(i) - m(2))^2\right)$$

Note the progressive replacement in the formulas with MATLAB functions like *sqrt*. This coefficient measures the strength and direction of the linear relationship between the two arrays. A positive value of r indicates a positive correlation, a negative r indicates a negative correlation, and r close to 0 indicates a weak or no linear correlation. In other words, the *Pearson correlation coefficient*, often denoted as "r," is a common method used to determine how closely two variables are linearly related, with the values ranging from -1 (perfect negative correlation) to 1 (perfect positive correlation), and 0 indicating no linear correlation. The result of the *Pearson correlation coefficient* calculation is returned from the *p* function, and then is printed in the output. Note that the *Pearson correlation coefficient* between two vectors can be computed directly in MATLAB using the built-in *corrcoef* function, which returns the matrix of correlation coefficients. For our specific vectors a and b, this can be done in a single line of code: $r = corrcoef(a, b)$; This line will return a 2×2 matrix where $r(1,2)$ (or $r(2,1)$) will be the *correlation coefficient* between a and b.

10.7.4 Ex. (197) – Vertical chart from the array with pre-declared values

```
a = [5, 2, 8, 4, 6, 12, 8, 9];
m = 10;

% Preallocate the cell array.
t = cell(m, length(a));
n = length(a);
max_val = max(a);

for j = 0:(m-1)
    for i = 1:n
        t{m-j, i} = ' ';
        f = floor((m / max_val) * a(i));
        if j < f
            % Dark shade character.
            t{m-j, i} = char(9619);
        else
            % Light shade character.
            t{m-j, i} = char(9617);
        end
    end
end

disp(SMC(t));

function r = SMC(m)
    r = '';
    for i = 1:size(m, 1)
        for j = 1:size(m, 2)
            r = [r m{i, j}];
        end
        r = [r newline];
    end
end
```

Output:



The application from above shows how an ASCII (American Standard Code for Information Interchange)/UTF-8 chart can be made in the console, with no sophisticated graphical interfaces. The code operates on an array *a*, which contains a list of numerical values. It also initializes some variables and a 2D array *t* used for generating a graphical representation of the data. Next, the code calculates the maximum value within array *a* using the *max* function and stores it in the *max_val* variable. A double *for-loop* is used to populate the 2D array *t*. The outer loop iterates from 0 to *m*–1, and the inner loop iterates from 1 to *n*, where *m* is a scalar variable and *n* is the length of array *a*. Inside the inner loop, the code calculates the value *f* using the formula $(m/max\_val) \times a(i)$, and then it assigns the corresponding character ("\u2591" or "\u2593") to the *t* array based on the relationship between *j* and *f*. The character "\u2591" represents a light shade block, and "\u2593" represents a dark shade block, so this code is essentially generating a bar graph where the darkness of the blocks represents the relative magnitude of the values in the array *a*. After the construction of the 2D array *t*, it calls the *SMC* function to convert

the array into a string format that visually represents the data in the console for user inspection.

10.7.5 Ex. (198) – Vertical chart from array with random values at each run

```
% Generate a random array of n
% numbers between 0 and 99.
n = 9;
a = floor(rand(1, n) * 100);

disp(SMC(chart(a)));
disp(newline + mat2str(a));

function t = chart(a)
    m = 9;
    max_val = max(a);
    t = cell(m, length(a));

    for j = 0:(m-1)
        for i = 1:length(a)
            f = floor((m / max_val) * a(i));
            t{m-j, i} = char(9617);
            if j < f
                t{m-j, i} = char(9619);
            end
        end
    end
end

function r = SMC(a)
    [n, m] = size(a);
    r = '';

    for i = 1:(n+2)
        for j = 1:m

            if i <= n
                % Chart characters.
                r = [r a{i, j}];
```

Output:

```
▒▒▒▒▒▓▒▒▒▒_9
▓▒▒▒▒▓▒▒▒▒_8
▓▒▒▒▒▓▓▓▒▒_7
▓▒▒▓▓▓▓▓▒▒_6
▓▒▒▓▓▓▓▓▓▒_5
▓▒▒▓▓▓▓▓▓▒_4
▓▒▒▓▓▓▓▓▓▓_3
▓▒▒▓▓▓▓▓▓▓_2
▓▓▓▓▓▓▓▓▓▓_1
|||||||||
123456789

79,17,61,61,95,
71,70,56,70,30
```

```
                r = [r '|'];
            else
                % Column numbers.
                r = [r num2str(j)];
            end
        end

        if i == (n+1)
            % Newline after the
            % horizontal axis.
            r = [r newline];
        elseif i <= n
            % Row numbers
            % and newline.
            r = [r '_' num2str(n-i+1) newline];
        end
    end
end
```

Compared to the previous example, the new implementation brings the ASCII axis for this chart and the ability to view the chart with different values each time the code is run. Thus, the given code creates a visual representation of data in the form of a bar chart using ASCII characters. The code begins by declaring a variable n with a value of 9, and an array a with a size determined by the value of n. The code line $a = floor(rand(1, n) * 100)$; generates the one-dimensional array a and populates it with random integers. In other words, function *rand* generates an array of random numbers with dimensions 1 by n, where n is a predefined variable. Each number in this array is a floating-point number greater than or equal to 0 and less than 1, uniformly distributed. The multiplication of the array with 100 allows to scale these random numbers to a new range. Thus, each number will be greater than or equal to 0 and less than 100. Next, the *floor* function rounds each element of the array down to the nearest integer. As a result, we get an array of integers where each integer is greater than or equal to 0 and less than 100. Next, the code prints the result of the $SMC(chart(a))$ function, which generates a bar chart from the array a and then prints a visual representation of the chart. The $chart(a)$ function is defined next. It takes an array a as its parameter. Inside the function, it initializes variables m and t, and it calculates the maximum value in the array a by using $max(a)$. A nested *for-loop* constructs the visual representation of the bar chart, using *Unicode* block characters (▓ and ▒) to represent the data. The result is stored in the t array, which is returned at the end. The $SMC(a)$ function is defined to create a string representation of the bar chart. It takes an array a as its parameter, representing the bar chart. It calculates the dimensions of the chart, initializes an empty string r, and then constructs the chart using nested loops. The loops iterate through the array a and construct the chart row by row, with underscores ("_") at the end of each row, and pipe characters "|" at the bottom of the chart. String r represents the bar chart and is then returned. In the end, the code prints the chart using $disp(SMC(chart(a)))$, followed by a new line character and a display of the original data array a.

10.7.6 Ex. (199) – Shannon entropy

```
c = 'uiuhd87wqsaidhsad';
disp(entropy(c));

function e = entropy(c)
    a = alpha(c);
    n = length(a);
    k = length(c);
    e = 0;

    for i=1:n
        r = length(strrep(c, a(i), ''));
        p = (k - r) / k;
        e = e + (p * log2(1/p));
    end
end

% Detect unique characters
% (alphabet) in the string.

function a = alpha(c)
    a = unique(c);
end

%{

Note: The user defined Log function is not needed in MATLAB
as log2() is built-in. However, if one wishes to avoid built-
in functions, the logarithm of b in base a, can be found in
Ex. (182).

Also, note that 'unique(c)' is a built-in function that is
MATLAB specific. The native version of this function is found
in the Ex. (187).

%}


% A second MATLAB version based
% on built in functions is shown
% below:

disp(entropy('uiuhd87wqsaidhsad'));

function e = entropy(c)
    charSet = unique(c);
    freqs = histc(c, charSet) / length(c);
    e = -sum(freqs .* log2(freqs + eps));
end

% Note that 'eps' ensures no log(0).
```

Output:

```
3.21
```

The MATLAB code from above calculates the *Shannon* entropy of a string, which is a measure of the randomness or disorder in the string. The function *entropy* is defined

twice, each implementing the calculation differently but achieving the same goal. In the first implementation, the *entropy* function starts by identifying the unique characters in the string *c* (i.e., "uiuhd87wqsaidhsad") using a helper function *alpha*, which employs the *unique* built-in function from MATLAB. It then calculates the frequency of each character in the string and uses this frequency to compute the entropy. The frequency is determined by replacing each occurrence of *a* character with an empty string and measuring the length difference (a strategy also presented in Ex. (134)). The entropy is calculated using the out of context formula $e = e + (p * \log_2(1/p))$, where p is the probability of each character:

$$e = \sum_{i=1}^{n} p_i \times \log_2\left(\frac{1}{p_i}\right)$$

The second implementation of entropy is more concise, utilizing the built-in functions of MATLAB more extensively. It still begins by identifying the unique characters but then uses *histc* to directly calculate the frequencies of each character in the string. It computes the entropy using the formula $e = -sum(freqs.* log2(freqs + eps))$. Note that in MATLAB, the ".*" operator is used for element-wise multiplication of arrays. It multiplies each element of one array with the corresponding element of another array of the same size. The mathematical formula for the entropy calculation used in the MATLAB code line from above can be expressed as follows:

$$e = -\sum_{i=1}^{n} f_i \times \log_2(f_i + \varepsilon)$$

Were, n is the number of unique characters in the string, f_i represents the frequency of each unique character i in the string, log_2 denotes the logarithm base 2, and ε (epsilon) is a very small number (also known as a pseudocount) added to avoid the logarithm of zero, as the logarithm of zero is undefined. The sum then operates over all unique characters, multiplying the frequency of each character by the logarithm of that frequency (plus epsilon) and then summing these products. Again, the *eps* is added to avoid taking the logarithm of zero, which would result in a mathematical error. In MATLAB, *eps* is a function that returns the machine epsilon, which is the smallest positive number that, when added to 1, yields a result different from 1 due to floating-point precision. Thus, the use of *eps* is common in situations where division by zero or taking a logarithm of zero must be avoided. For instance, when calculating the logarithm of a variable that might be zero, $log(variable + eps)$ is used to ensure that the logarithm function always has a valid input greater than zero, preventing errors or undefined results. However, note that both implementations effectively calculate the *Shannon* entropy, providing a quantitative measure of the amount of information in the string. This can be useful in various fields, including information theory, cryptography, and data compression. The entropy value gives an indication of how much variability there is in the content of a particular string.

10.8 Useful Conversions

Conversions between various numerical and text representations are essential in computer science, programming, and digital communication. These conversions enable us to translate data between different formats, making it more accessible, and adaptable for specific applications. Among the most fundamental and commonly used conversions are those between hexadecimal (hex), text (txt), decimal, and binary representations [5]. In this subchapter, we will explore the significance and utility of these conversions, highlighting their relevance in various aspects of computing and data manipulation. Whether it is about encoding characters into binary for digital storage or translating numeric data into a human-readable format, understanding these conversions is crucial for anyone working with digital information.

10.8.1 Ex. (200) – Text (txt) to hexadecimal (hex)

```
% txt to hex

a = '.~ text';
c = cell(size(a));

for i = 1:length(a)

    % Convert character to
    % decimal and then to hex.

    b = dec2hex(double(a(i)));

    if length(b) < 2

        % Pad with zero
        % if single digit.

        c{i} = ['0', b];
    else
        c{i} = b;
    end
end

disp(c);
```

Output:

```
2e,7e,20,74,65,78,74
```

A conversions of text to hexadecimal representation is shown here. The code takes a string a containing characters and converts it into a hexadecimal representation. It starts by defining three variables: a to store the input string ".~text," b to store temporary values during the conversion, and c to store the hexadecimal value of each character. Then, it enters a *for-loop* that iterates through each character in the string a. Inside the loop, it uses $a(i)$ to get the current character. That is possible in MATLAB because a string in single quotes is a char array by default. The line of code $b = dec2hex(double(a(i)))$;

within the script is responsible for converting each character of the string *a* into its hexadecimal representation. Thus, *a*(*i*) selects the *i*-th character of the string *a*, whereas *double*(*a*(*i*)) function converts this character to its corresponding ASCII value. ASCII (American Standard Code for Information Interchange) is a character encoding standard where each character (like letters, numbers, and symbols) is represented by a numeric value. Next, *dec2hex* converts this ASCII value from decimal to hexadecimal format. The result is a string representing the hexadecimal value of the ASCII code of the character *a*(*i*). For example, if *a*(*i*) is the character "A", *double*(*a*(*i*)) would convert it to 65 (the ASCII value of "A"), and *dec2hex*(*double*(*a*(*i*))) would convert the value 65 to "41", which is the hexadecimal representation of 65. Nevertheless, this line of code, namely *dec2hex*(*double*(*a*(*i*))), is executed in a loop for each character in the string *a*, thereby converting the entire string into a sequence of hexadecimal values. The code checks if the resulting hexadecimal string *b* has a length less than 2. If it does, it adds a leading "0" to ensure that the hexadecimal representation always consists of two characters. The resulting two-character hexadecimal value is then assigned to the corresponding index in the *c* array. Next and last, the code prints the hexadecimal representations from the *c* array.

10.8.2 Ex. (201) – A *txt* to *hex* from array *a* to array *b*

```
% txt to hex from array a to array b element
correspondence

% Convert string to
% a character array.

a = '.~ text';
b = cell(size(a));

for i = 1:length(a)

    b{i} = dec2hex(double(a(i)));

    % Ensure at least 2 characters,
    % padding with '0' if necessary.

    b{i} = pad(b{i}, 2, 'left', '0');
end

disp(b);
```

Output:

```
2e,7e,20,74,65,78,74
```

The code from above is a version of the previous example and it involves arrays instead of strings. The implementation is designed to convert characters from an array *a* into their corresponding hexadecimal representations and store them in an array called *b*. The code starts by initializing an array, implicitly splitting the string ".~text" into individual characters and storing them as array elements. A zero filled array *b* is also initialized. A *for-loop* is used to iterate through the elements of array *a*. Inside the loop two events

take place. The line of code $b = dec2hex(double(a(i)))$; within the script is responsible for converting each character of the string a into its hexadecimal representation. Next, the main result of the conversion is stored in the corresponding index of array b. The *pad* function is used for padding strings. The first argument is the string to be padded, which in this case is $b\{i\}$, the ith element of the cell array b. The second argument, 2, specifies the total length that the string should have after padding. The third argument, "left", indicates that padding should be added to the left side of the string. The fourth argument, "0", specifies that the padding should be done with the character zero. Thus, in this way the conditional statement used in the previous example is removed here. Next, the code prints array b to the console for user inspection, which contains the hexadecimal representations of the characters in the array a. Please note that the transformation was made from matrix a to the matrix b, and the c matrix seen in the previous example was removed.

10.8.3 Ex. (202) – A *txt* to *hex* with in-place replacement

```
% txt to hex by replacing each character with
% the hex code in the same element of the array.
a = char('.~ text');

for i = 1:length(a)

    % Convert character to hexadecimal.
    a(i) = dec2hex(uint8(a(i)));

    % Prepend zero if single digit hex.
    if length(a(i)) < 2
        a(i) = strcat('0', a(i));
    end
end

disp(a);
```

Output:

```
2e,7e,20,74,65,78,74
```

Here, a small optimisation is shown, that is able to use a single array and a conversion in place. As before, the code begins by defining an array a which is initialized by splitting the string ".~text" into individual characters and storing them as separate elements in the array. A *for-loop* is then employed to iterate through the elements of the array a. Inside the *for-loop*, the conversion of $a(i)$ is done in place, i.e. replacing the original value in $a(i)$ with the hexadecimal value. Next, the array a is printed using the *disp* function. Note: This time the conversion was made from matrix a to the same matrix a, and the matrix b seen in the previous example was removed.

10.8.4 Ex. (203) – A *txt* to *hex* in a function that receives an as argument

```
a = char('.~ text');
disp(hex(a));

function hexArray = hex(a)

    % Initialize a cell array
    % to hold hex values.

    hexArray = cell(size(a));
    for i = 1:numel(a)

        % Convert character to
        % decimal and then to hex.

        hexValue = dec2hex(double(a(i)));

        % Prepend zero if
        % single digit hex.

        if length(hexValue) < 2
            hexValue = ['0', hexValue];
        end

        hexArray{i} = hexValue;
    end

    % Convert cell array
    % back to character array.

    hexArray = char(hexArray);
end
```

Output:

```
2e,7e,20,74,65,78,74
```

The current example is the same as the previous one, however, the difference is that a *hex* function with a parameter *a*, embeds the *for-loop*. In the code outside the function, there is an array *a* initialized with the characters of the string ".~text" split into individual characters. Then, it calls the *hex*(*a*) function with this array as an argument and prints the result to the console window for user inspection. Please notice that the above example includes the code from Ex. (200), into a function.

10.8.5 Ex. (204) – Multiple functions for any to any conversion

```
a = '☁È.~ text';
a_cell = num2cell(a); % Convert to cell array

disp(['Array a = ', ...
cellArrayToString(a_cell)]);

disp(['txt_hex = ', ...
cellArrayToString(txt_hex(a_cell))]);

disp(['hex_bin = ', ...
cellArrayToString(hex_bin(txt_hex(a_cell)))]);

disp(['bin_dec = ', ...
cellArrayToString(bin_dec(hex_bin(txt_hex(a_cell))))]);

disp(['dec_txt = ', ...
cellArrayToString(dec_txt(bin_dec(hex_bin(txt_hex(a_cell)))))]);

disp(['txt_bin = ', ...
cellArrayToString(txt_bin(a_cell))]);

disp(['bin_hex = ', ...
cellArrayToString(bin_hex(txt_bin(a_cell)))]);

disp(['hex_dec = ', ...
cellArrayToString(hex_dec(txt_hex(a_cell)))]);

disp(['dec_bin = ', ...
cellArrayToString(dec_bin(bin_dec(hex_bin(txt_hex(a_cell)))))]);

disp(['bin_txt = ', ...
cellArrayToString(bin_txt(txt_bin(a_cell)))]);

disp(['txt_dec = ', ...
cellArrayToString(txt_dec(a_cell))]);
```

```
disp(['dec_hex = ', ...
cellArrayToString(dec_hex(bin_dec(hex_bin(txt_hex(a_cell))))]);

disp(['hex_txt = ', ...
cellArrayToString(hex_txt(txt_hex a_cell))]);

function str = cellArrayToString(cellArray)
    str = strjoin(cellArray, ' ');
end

function a = txt_hex(a)
    a = cellfun(@(c) dec2hex(double(c), 2), a, 'UniformOutput', false);
end

function a = txt_bin(a)
    a = cellfun(@(c) dec2bin(double(c)), a, 'UniformOutput', false);
end

function a = txt_dec(a)
    a = cellfun(@(c) num2str(double(c)), a, 'UniformOutput', false);
end

function a = hex_txt(a)
    a = cellfun(@(c) char(hex2dec(c)), a, 'UniformOutput', false);
end

function a = hex_bin(a)
    a = txt_bin(hex_txt(a));
end

function a = hex_dec(a)
    a = txt_dec(hex_txt(a));
end

function a = bin_hex(a)
    a = txt_hex(bin_txt(a));
end

function a = bin_txt(a)
    a = cellfun(@(c) char(bin2dec(c)), a, 'UniformOutput', false);
end

function a = bin_dec(a)
    a = txt_dec(bin_txt(a));
end

function a = dec_hex(a)
    a = txt_hex(dec_txt(a));
end
```

```
function a = dec_txt(a)
    a = cellfun(@(c) char(str2double(c)), a, 'UniformOutput', false);
end

function a = dec_bin(a)
    a = txt_bin(dec_txt(a));
end
```

```
Output:

Array a = ☁,È,.,~, ,t,e,x,t
txt_hex = 2601,400,2e,7e,20,74,65,78,74
hex_bin = 10011000000001,10000000000,101110,1111110,
          100000,1110100,1100101,1111000,1110100
bin_dec = 9729,1024,46,126,32,116,101,120,116
dec_txt = ☁,È,.,~, ,t,e,x,t
txt_bin = 10011000000001,10000000000,101110,1111110,
          100000,1110100,1100101,1111000,1110100
bin_hex = 2601,400,2e,7e,20,74,65,78,74
hex_dec = 9729,1024,46,126,32,116,101,120,116
dec_bin = 10011000000001,10000000000,101110,1111110,
          100000,1110100,1100101,1111000,1110100
bin_txt = ☁,È,.,~, ,t,e,x,t
txt_dec = 9729,1024,46,126,32,116,101,120,116
dec_hex = 2601,400,2e,7e,20,74,65,78,74
hex_txt = ☁,È,.,~, ,t,e,x,t
```

In the previous examples the narrative of the code reached the point of function formation. Here, this MATLAB code performs a series of data type conversions on a string *a* (i.e., "☁È.~text"), and uses a collection of functions to convert between text, hexadecimal, binary, and decimal formats. The process is done in a sequence of steps, each utilizing different custom functions. The string *a* is first converted to a cell array *a_cell* by using the *num2cell* function, which creates a cell array with each character of the string as an individual cell element. This format is necessary for the subsequent cell-wise operations. Also, the function *cellArrayToString* is used throughout to convert cell arrays back to a string representation for display. It concatenates the elements of the cell array with spaces in between by using the *strjoin* function. The *txt_hex* function converts each character of the text in the cell array to its hexadecimal representation using function *dec2hex* after converting the character to its decimal ASCII value with function *double*. Next, the *txt_bin* converts text to binary by first finding the decimal ASCII value of each character with function *double* and then converting it to binary by using the *dec2bin* function. Next, the *txt_dec* function similarly converts text to decimal. The ASCII value of each character is obtained with function *double* and then converted to a string by using function *num2str*. Also, functions such as *hex_txt*, *hex_bin*, and *hex_dec* first convert hexadecimal to text by using the *hex2dec* built-in function and the *char* function. Then the appropriate functions are called (*txt_bin*, *txt_dec*) for the next conversion step. Next, the functions *bin_hex*, *bin_*

txt, and *bin_dec* perform the reverse operations, converting binary to other formats. The *bin2dec* built-in function is used to convert binary to decimal, and then the appropriate function is called for the next conversion step. Next, the functions *dec_hex*, *dec_txt*, and *dec_bin* convert decimal to other formats. For decimal to text, built-in function *str2double* is used to convert the string to a numeric value, which is then converted to a character with function *char*. At *run-time*, the code finally applies these conversion functions to the array *a*, one by one, and prints the results for each step by using the *disp* function. Therefore, this code demonstrates how to manipulate character data in various formats, showcasing the flexibility of MATLAB as a computer language, in handling different representations of text characters.

10.8.6 Ex. (205) – One function for any to any conversion and input type detection

```
a = '☁È.~ text';
a_cell = num2cell(a);

disp(['Array a = ', cellArrayToString(a_cell)]);

disp(['Converted to bin = ', ...
cellArrayToString(convert_to('bin', a_cell))]);

function str = cellArrayToString(cellArray)
    str = strjoin(cellArray, ' ');
end

function a = convert_to(h, a)

    n = length(a);
    t = 0;

    for i = 1:n
        t = t + length(a{i});
    end

    t = t / n;

    if(t == 1)
        q = 'txt';
    elseif(t >= 2 && t < 3)
        q = 'hex';
    elseif(t >= 3 && t < 4)
        q = 'dec';
    else
        q = 'bin';
    end
```

```
    for i = 1:n

        switch [q, '_', h]
            case 'txt_hex'
                a{i} = dec2hex(double(a{i}), 2);
            case 'dec_hex'
                a{i} = dec2hex(str2double(a{i}), 2);
            case 'txt_bin'
                a{i} = dec2bin(double(a{i}));
            case 'txt_dec'
                a{i} = num2str(double(a{i}));
            case 'hex_dec'
                a{i} = num2str(hex2dec(a{i}));
            case 'bin_dec'
                a{i} = num2str(bin2dec(a{i}));
            case 'hex_txt'
                a{i} = char(hex2dec(a{i}));
            case 'bin_txt'
                a{i} = char(bin2dec(a{i}));
            case 'dec_txt'
                a{i} = char(str2double(a{i}));
            case 'dec_bin'
                a{i} = dec2bin(str2double(a{i}));
            case 'hex_bin'
                a{i} = dec2bin(hex2dec(a{i}));
            case 'bin_hex'
                a{i} = dec2hex(bin2dec(a{i}), 2);
        end

    end
end
```

Output:

```
Array a = ☁,È,.,~, ,t,e,x,t
10011000000001,10000000000,101110,1111110,100000,1110100,1100101,1
111000,1110100
```

Separate functions for one to one conversions are important pieces of code that can be used as they are in different contexts. However, what about only one function that can convert anything into anything? The following code is another implementation that performs character data conversions between different representations, such as hexadecimal, binary, decimal, and plain text, similar to the previous code. The code starts by initializing a string variable *a* with the value "☁È.~text" and then splits it into an array of individual characters, just like in the previous version. Then, it uses the *disp* function to display the original array *a*. This time, instead of defining multiple conversion functions as in the previous version, this code defines a single function called *convert_to*(*h*, *a*). This function takes two arguments: *h*, which represents the target conversion type (e.g., "bin", "hex", "dec", "txt"), and *a*, which is the input array to be converted. Inside the *convert_to* function, the code calculates the average length of characters in the input array *a* and determines the appropriate conversion type *q* based on this average length. Next, it uses the *strcmp* function (returns 1 if *h* and *q* are the same and 0 otherwise) that checks if the target conversion type *h* is the same as the determined *q*. If they are the same, it returns the input array *a* as there is no need to perform a conversion. If *h* is different from *q*, it iterates through the characters in the array and applies various conversion cases based on the combination of *q* and *h*. These cases cover conversions like text to hexadecimal, text to binary, decimal to text, etc. After performing the necessary conversions, the function returns the modified array *a*. This version of the code is more modular and concise than the previous one. It defines a single conversion function that handles all conversion cases dynamically based on the target and source types. The previous version defined multiple conversion functions, each with a specific purpose, making it longer and potentially harder to maintain. Both versions achieve the exact same goal of character data conversions, but this version encapsulates the logic within a single function, making it more versatile and adaptable to different conversion scenarios. Additionally, this version calculates the target conversion type dynamically based on the average character length, which is a feature not present in the previous version. Notice again that the input string type is identified by the first part of the *convert_to* function, which then triggers the conversion case automatically.

10.8.7 Ex. (206) – *Base64* encoding function

```
function example
    s = 'ABC';
    q = encodeBase64(s);
    disp(q);
end

function r = encodeBase64(s)

    a = ['ABCDEFGHIJKLMNOPQRSTUVWXYZ' ...
         'abcdefghijklmnopqrstuvwxyz' ...
         '0123456789+/'];

    b = '';
    for i = 1:length(s)
        c = dec2bin(s(i), 8);
        b = strcat(b, c);
    end

    % Pad the binary string to make
    % its length a multiple of 6.

    while mod(length(b), 6) ~= 0
        b = strcat(b, '0');
    end

    r = '';
    for i = 1:6:length(b)
        x = b(i:i+5);
        d = bin2dec(x);
        r = strcat(r, a(d+1));
    end

    % Add padding if necessary
    while mod(length(r), 4) ~= 0
        r = strcat(r, '=');
    end
end
```

Output:

```
QUJD
```

The code from above defines an encoding function called *encodeBase64* that takes a string *s* as input and returns its Base64 encoding. The code starts by defining a string *s* with the value "ABC." Next, it calls the *encodeBase64* function with *s* as an argument and assigns the result to a variable *q*. The *encodeBase64* function begins by initializing two strings, *a* and *b*. Variable *a* contains the Base64 encoding characters for uppercase letters, lowercase letters, numbers, and two special characters ("+" and "/"), whereas variable *b* is an empty string that will be used to store the binary representation of the characters in the input string. A *for-loop* iterates over each character in the input string *s*. Inside the loop, it converts each character (i.e., *dec2bin*) to its binary representation (8 bits) and ensures that the binary representation is left-padded with zeros to make it exactly 8 bits long. The binary representations are appended to the *b* string by using the *strcat* function.

The *strcat* function in MATLAB is used to concatenate strings. It concatenates strings horizontally, end-to-end, and automatically removes trailing white spaces (spaces, tabs, newlines) from each input string before concatenation. After converting all characters to binary and appending them to *b*, the code checks if the length of *b* is not a multiple of 6. If not, it adds zeros to the end of *b* until its length becomes a multiple of 6. Then, the code initializes an empty string *r*, which will store the final Base64-encoded result. Another *for-loop* iterates over the binary string *b* in chunks of 6 bits at a time. It converts each 6-bit chunk back to decimal and uses the decimal value as an index to look up the corresponding Base64 character from the *a* string. The Base64 characters are appended to the *r* string. Thus, the code checks if the length of the *r* string is not a multiple of 4. If not, it adds "=" padding characters to the end of the *r* string until its length becomes a multiple of 4. The *encodeBase64* function returns the Base64-encoded result *r*. The main part of the code concludes by printing the result *q* to the console window, which is the Base64 encoding of the input string "ABC."

Complex Examples

11

Complex examples in MATLAB represent the intricate and multifaceted aspects of this versatile computer language. As MATLAB has evolved over the years, it has become a powerful tool for scientists and engineers to design computer applications. Whether it is about building interactive applications for simulations, implementing complex algorithms, or integrating with different visualisation techniques, complex MATLAB examples showcase the adaptability of this computer language and its ability to tackle the demands of modern software development. In this exploration of complexity, scientists and engineers can discover innovative solutions and push the boundaries of what is achievable in MATLAB. The following examples demonstrate how MATLAB can be used to create sophisticated and dynamic science and engineering applications and manipulate data, among other advanced functionalities that can be integrated. Note that many of the coding strategies used in the examples of previous chapters, here, will be fully integrated into different contexts.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_11

11.1.1 Ex. (207) – Spectral forecast for signals

```
% Spectral forecast for signals in Matlab.

A = '10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4';
B = '18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4';

tA = str2double(strsplit(A, ','));
tB = str2double(strsplit(B, ','));

% Maximum values in the arrays.
maxA = max(tA);
maxB = max(tB);

% Overall maximum value.
maxVal = max(maxA, maxB);

d = 33;
M = '';

% Calculate the combined signal M.
for i = 1:length(tA)
    v = ((d / maxA) * tA(i)) + (((maxVal - d) / maxB) * tB(i));
    M = [M, sprintf('%.2f', v)]; % two decimal places.
    if i < length(tA)
        M = [M, ','];
    end
end

fprintf('Signal A: %s\n', A);
```

```
fprintf('Max(A[i]): %.2f\n', maxA);
fprintf('Signal M: %s\n', M);
fprintf('Signal B: %s\n', B);
fprintf('Max(B[i]): %.2f\n', maxB);
```

Output:

```
Signal A:10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4
Max(A[i]):63.2
Signal M:15.37,35.12,51.12,57.17,47.89,43.08,60.35,67.91,63.72
Signal B:18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4
Max(B[i]):70.4
```

A spectral forecast implementation for signals is shown above. It processes two input signal arrays, *A* and *B*, and calculates a modified signal *M* based on certain mathematical operations [1]. That is, it creates a signal *M* that resembles signal *A* and *B* in a certain proportion set by variable *d*. The code initially begins by defining three variables: *A*, *B*, and *M*. Variables *A* and *B* are initialized with comma-separated strings of numerical values, representing two input signals. Variable *M* is initialized as an empty string, which will store the modified signal. Two arrays, tA and tB, are used to store the numeric values converted from strings by using the *str2double* function. The code then uses the *strsplit*() function to split the strings *A* and *B* into arrays *tA* and *tB*, respectively, using commas as the delimiter. Next, it calculates the maximum values in the *tA* and *tB* arrays using the *max*() function. The maximum values are stored in *maxA* and *maxB* variables. The overall maximum value between *maxA* and *maxB* is calculated and stored in the *maxVal* variable. Next, a variable *d* is initialized with the value 33. A *for-loop* is used to iterate through the elements of the *tA* array. Inside the loop, a variable *v* is calculated using a formula that involves scaling the elements of *tA* and *tB* by certain factors based on *maxA*, *maxB*, and *d*. The calculated value *v* is then appended to the *M* string with two decimal places using the *sprintf*("%0.2f", *v*) function. A comma is added to separate values in the *M* string only if it is not the last element. Post processing, the code prints the following information in the output: (i) The original signal *A*. (ii) The maximum value found in *A*. (iii) The modified signal *M*. (iv) The original signal *B*. (v) The maximum value found in *B*.

11.1.2 Ex. (208) – Logic gate functions applied to matrix elements

```
a = [
    [1, 1, 1];
    [0, 1, 0];
    [0, 1, 0]
    ];

b = [
    [0, 1, 0];
    [1, 1, 1];
    [0, 1, 0]
```

```
Output:

0 1 0
0 1 0
1 1 1

C = 0 0 0 1
C = 0 1 1 1
B = 1 0
```

```
    ];

c = zeros(3, 3);

[n, m] = size(a);
r = '';

for i = 1:n
    r = strcat(r, '\n');
    for j = 1:m
        c(i, j) = f_xnor(a(i, j), b(i, j));
        r = strcat(r, num2str(c(i, j)), ' ');
    end
end

fprintf(r);

function result = f_not(a)
    result = 1 - a;
end

function result = f_and(a, b)
    result = a * b;
end

function result = f_or(a, b)
    result = (a + b) - (a * b);
end

function result = f_nand(a, b)
    result = f_not(f_and(a, b));
end

function result = f_nor(a, b)
    result = f_not(f_or(a, b));
end

function result = f_xor(a, b)
    result = (a + b) - 2 * (a * b);
end

function result = f_xnor(a, b)
    result = f_not(f_xor(a, b));
end


% MATLAB specific built-in logic operations:


% For two variables A and B, the AND operation can be
performed as C = A & B:

a = [0 1 0 1];
```

```
C = 1 1 1 0
C = 1 0 0 0
C = 0 1 1 0
C = 1 0 0 1
```

```
b = [0 0 1 1];
c = a & b;

% The OR operation can be performed as C = A | B:

a = [0 1 0 1];
b = [0 0 1 1];
c = a | b;

% The NOT operation can be performed as B = ~A:

a = [0 1];
b = ~a;

% This is the NAND (negation of AND) and can be
performed as C = ~(A & B):

a = [0 1 0 1];
b = [0 0 1 1];
c = ~(a & b);

% This is the NOR (negation of OR) and can be
performed as C = ~(A | B):

a = [0 1 0 1];
b = [0 0 1 1];
c = ~(a | b);

% For two variables A and B, the XOR operation can be
performed using C = xor(A, B):

a = [0 1 0 1];
b = [0 0 1 1];
c = xor(a, b);

% This is the XNOR (negation of XOR) and can be
performed as C = ~xor(A, B):

a = [0 1 0 1];
b = [0 0 1 1];
c = ~xor(a, b);
```

This example defines and performs operations on matrices *a*, *b*, and *c* using various logical functions. The code begins by initializing three matrices *a*, *b*, and *c*, where each matrix is represented as an array of arrays, containing numeric values. Variables *n* and *m* are set to the number of rows and columns in matrix *a*, respectively. An empty string *r* is initialized, which will be used to build a string representation of the result. Next, a nested *for-loop* is used to iterate over each element in matrices *a* and *b*. Inside the loop, the *f_xnor* function is called with the corresponding elements from *a* and *b*. The result of the *f_xnor* function is stored in the corresponding position of matrix *c*, and the result is also appended to the string *r* with a space. The *f_xnor* function is defined to calculate

the XNOR (exclusive NOR) operation between two values. It calls the *f_xor* function and then negates the result using the *f_not* function. Below the main code logic, several logical functions are defined. Function *f_not*(*a*) that returns the logical NOT operation of a value. Next, function *f_and*(*a*, *b*) is defined, that returns the logical AND operation between two values. Also, function *f_or*(*a*, *b*) is defined, that is able to return the logical OR operation between two values. Next, function *f_nand*(*a*, *b*) returns the logical NAND operation between two values by combining the *f_and* and *f_not* functions. Next the *f_nor*(*a*, *b*) function returns the logical NOR operation between two values by combining the *f_or* and *f_not* functions. Also, function *f_xor*(*a*, *b*) is defined, that returns the logical XOR operation between two values. Lastly, the *f_xnor*(*a*, *b*) function is defined, that is able to return the logical XNOR operation between two values by combining the *f_xor* and *f_not* functions. The main example uses fundamentals to construct the logical gates functions, however, a second example is also shown, in which built-in functions from MATLAB can be used directly.

11.1.3 Ex. (209) – The general logic gate based on a map

```
a = [
    [1, 1, 1];
    [0, 1, 0];
    [0, 1, 0]
    ];

b = [
    [0, 1, 0];
    [1, 1, 1];
    [0, 1, 0]
    ];

c = zeros(size(a));
n = size(a, 1);
m = size(a, 2);
r = '';

for i=1:n
    r = strcat(r, '\n');
    for j=1:m
        c(i,j) = g(a(i,j), b(i,j), 6);
        r = strcat(r, int2str(c(i,j)), " ");
    end
end

fprintf(r);

% A B
% t = 1 = AND
% t = 2 = NAND
% t = 3 = OR
% t = 4 = NOR
```

```
Output:

0 1 0
0 1 0
1 1 1
```

```
% t = 5 = XOR
% t = 6 = XNOR

function result = g(a, b, t)
    % Truth table
    h = [ % A  B  1  2  3  4  5  6
            [0, 0, 0, 1, 0, 1, 0, 1];
            [0, 1, 0, 1, 1, 0, 1, 0];
            [1, 0, 0, 1, 1, 0, 1, 0];
            [1, 1, 1, 0, 1, 0, 0, 1]
        ];

    n = size(h, 1);

    for i=1:n
        if a == h(i,1) && b == h(i,2)
            result = h(i,t+2);
            return;
        end
    end
end

%{
_____________________________________
|       |   | N |   |   |   | X |
| Input | A | A |   | N | X | N |
|       | N | N | O | O | O | O |
| a | b | D | D | R | R | R | R |
-------------------------------------
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
-------------------------------------
%}
```

The previous example showed independent scattered functions for the logic gate functions. However, how about an optimization that makes a shortcut that melts all the previously described functions into one? Well, this code defines two 3×3 matrices a and b as well as an empty matrix c. It then performs bitwise logical operations between the corresponding elements of matrices a and b based on a specified operation code t using a function called g. The result of these operations is stored in matrix c. Next, it prints the contents of matrix c in a human-readable format. In more detail, the matrices a and b are 3×3 arrays of binary values. The code initializes an array matrix c filled with zero values, and two variables n and m to store the dimensions of matrix a. It also initializes an empty string r to store the formatted result. A nested *for-loop* is used to iterate through the elements of matrices a and b, perform the logical operation using the g function, and store the result in the matrix c. It also constructs a string r that represents the elements of matrix c separated by spaces and newlines. The g function takes three arguments: a, b, and t. It uses a predefined matrix h to perform bitwise logical operations

based on the value of t and returns the result. The matrix h is a lookup table that specifies the results of different logical operations (AND, NAND, OR, NOR, XOR, XNOR) for different combinations of a and b. The code concludes by printing the string r, which represents the resulting matrix c. Additionally, there are comments in the source code that can explain the meaning of the values from t and provide a visual representation of the logical operations and their results in a tabular format.

11.1.4 Ex. (210) – Decompose a matrix into multiple matrices based on unique values

```
c = {
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1];
    [1, 2, 1, 0, 1, 3, 1, 0, 1, 1];
    [1, 1, 2, 0, 1, 3, 0, 1, 0, 1];
    [0, 1, 0, 2, 1, 3, 1, 0, 0, 1];
    [1, 1, 1, 0, 2, 3, 1, 0, 1, 0];
    [1, 0, 1, 1, 1, 3, 0, 0, 0, 0];
    [1, 0, 3, 3, 3, 3, 0, 0, 0, 1];
    [1, 0, 1, 1, 1, 0, 0, 9, 9, 9];
    [1, 1, 0, 0, 0, 0, 1, 9, 0, 9]
    };

b = matrix_alphabet(c);
t = decompose(c, b);

for k = 1:numel(b)
    disp(SMC(t, k));
end

disp(b);

function c = replace_with_nan(c, r)
    for i = 1:numel(c)
        for j = 1:numel(c{i})
            if ischar(c{i}(j)) && c{i}(j) == r
                c{i}(j) = NaN;
            end
        end
    end
end

function a = matrix_alphabet(c)
    a = [];
    for i = 1:numel(c)
        for j = 1:numel(c{i})
            if all(~ismember(a, c{i}(j)))
                a = [a, c{i}(j)];
            end
        end
    end
    a(isnan(a)) = [];
```

Output:

```
M1
 ----------
|111111 111|
|1 1 1 1 11|
|11  1  1 1|
| 1  1 1  1|
|111   1 1 |
|1 111     |
|1        1|
|1 111     |
|11    1   |
 ----------
M2
 ----------
|      0   |
|   0   0  |
|   0  0 0 |
|0 0    00 |
|   0   0 0|
| 0    0 00|
| 0     00 |
| 0    0   |
|  0000  0 |
 ----------
M3
 ----------
|          |
| 2        |
|  2       |
|   2      |
|    2     |
|          |
|          |
|          |
|          |
 ----------
M4
```

```
end

function d = decompose(c, a)
    [n, ~] = size(c);
    d = cell(n, numel(a));
    for i = 1:n
        for k = 1:numel(a)
            d{i, k} = '';
            for j = 1:numel(c{i})
                if isnumeric(c{i}(j)) && ...
                            c{i}(j) == a(k)
                    d{i, k} = [d{i, k}, ...
                               num2str(c{i}(j))];
                else
                    d{i, k} = [d{i, k}, ' '];
                end

            end
        end
    end
end


function r = SMC(m, k)
    r = sprintf('M%d\n', k);
    [n, ~] = size(m);
    for i = 1:n
        r = sprintf('%s|', r);
        for j = 1:numel(m{i, k})
            r = sprintf('%s%s', r, m{i, k}(j));
        end
        r = sprintf('%s|\n', r);
    end
end
```

```
 ----------
|          |
|     3    |
|     3    |
|     3    |
|     3    |
|     3    |
|  3333    |
|          |
|          |
 ----------
M5
 ----------
|          |
|          |
|          |
|          |
|          |
|          |
|          |
|       999|
|       9 9|
 ----------
1,0,2,3,9
```

This code implementation from above defines a series of variables and functions to manipulate and decompose a matrix c based on an alphabet of unique values found in that matrix. The code also prints the decomposed matrices and the alphabet. First, the code defines a 2D array c, representing a matrix, where each element is a numerical value. This matrix has dimensions 9×10. The code then calls the *matrix_alphabet* function to extract unique values from the matrix c and stores them in an array b. It also calls the *decompose* function to decompose the matrix c into a 3D array t. This decomposition is based on the alphabet found in b. A *for-loop* is used to iterate through the elements of the alphabet array b, and for each unique value in b, it prints a decomposed version of the matrix t using the *SMC* function. The *decompose* function takes the matrix c and the alphabet array a as arguments. It creates a new 3D array d and populates it by iterating through the elements of c. For each element in c, it creates a sub-array in d, and for each unique value in a, it assigns the corresponding value from c to the sub-array in d. The *matrix_alphabet* function extracts unique values from the matrix t and stores them in an array a. It iterates through the elements of t, checks if each value is already in a, and if

not, adds it to *a*. Also, the *SMC* function is used to create a string representation of a decomposed matrix. It takes a matrix *m* and an index *k* representing the alphabet element to consider. It constructs a string *r* that displays the decomposed matrix with the index *k*. The decomposed matrix is surrounded by lines and separators for better visualization. The implementation concludes by printing the alphabet array *b* and the decomposed matrices for each unique element.

11.1.5 Ex. (211) – Count islands over the matrix and show their location

```
a = [
    [ 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 ];
    [ 1, 0, 1, 0, 1, 1, 0, 0, 1, 1 ];
    [ 1, 1, 1, 0, 1, 1, 0, 0, 0, 1 ];
    [ 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 ];
    [ 1, 1, 1, 0, 1, 1, 1, 0, 1, 0 ];
    [ 1, 0, 1, 1, 1, 1, 0, 0, 0, 0 ];
    [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 ];
    [ 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 ];
    [ 1, 1, 0, 0, 0, 0, 0, 0, 0, 1 ]
    ];

b = [
    [+1, 0]; % right side element
    [-1, 0]; % left side element
    [ 0,+1]; % upward side element
    [ 0,-1]; % downward side element
    [+1,+1]; % upward-right side element
    [-1,-1]; % downward-left side element
    [+1,-1]; % downward-right side element
    [-1,+1]  % upward-left side element
    ];

% Pass b to the function
% and receive the modified a.

[a, c] = SCAN(a, b);
fprintf("Islands = %d\n", c);
fprintf(SMC(a));

function [a, c] = SCAN(a, b)
    n = size(a, 1); % row
    m = size(a, 2); % col
    c = 0;          % islands count

    for i = 1:n
        for j = 1:m
            if a(i,j) == 1
                c = c + 1;

                % Pass b and receive
```

Output:

```
Islands = 3

3 0 3 3 3 3 0 4 4 4
3 0 3 0 3 3 0 0 4 4
3 3 3 0 3 3 0 0 0 4
0 0 0 0 3 0 0 0 0 4
3 3 3 0 3 3 3 0 4 0
3 0 3 3 3 3 0 0 0 0
3 0 0 0 0 0 0 0 0 5
3 0 3 3 3 3 0 5 5 5
3 3 0 0 0 0 0 0 0 5
```

```
                % the modified a.

                [a, ~]=d(a, i, j, n, m, c, b);
            end
        end
    end
end

function [a, didChange] = d(a,i,j,n,m,c,b)
    didChange = false;
    if i<1 || j<1 || i>n || j>m || a(i,j)~=1
        return;
    end

    % Mark the cell as visited
    % by changing it to c+2.

    a(i,j) = c + 2;
    didChange = true;

    for k = 1:size(b,1)
        [a, changed] = ...
        d(a, i+b(k,1), j+b(k,2), n, m, c, b);
        didChange = didChange || changed;
    end
end

function r = SMC(m)
    r = "\n";
    for i = 1:size(m,1)
        for j = 1:size(m,2)
            r = r + num2str(m(i,j)) + " ";
        end
        r = r + "\n";
    end
end
```

This code defines a series of functions and uses them to perform operations on a 2D array called *a*. The array *a* represents a grid or map, containing information about islands and their connectivity. The *a* array is a 2D grid consisting of 10 rows and 10 columns, where each element is either 0 or 1, representing land (1) and water (0). The *b* array is an array of 2-element arrays, each representing a direction (right, left, upward, downward, etc.) for navigating neighboring elements in the grid. The code then proceeds with the following key functions: *d*, *SCAN*, and *SMC*. Function *d*(*a*, *i*, *j*, *n*, *m*, *c*, *b*) is a recursive function that is used to traverse the grid (*a*) starting from a given position (*i*, *j*). It explores neighboring elements and marks connected landmasses with a unique value *c*. This function recursively explores landmasses and increments the *c* value for each new landmass found. Function *SCAN*(*a*) scans the entire grid (*a*) for landmasses (regions of connected land elements). It initializes *c* to zero and, for each land element encountered, increments *c* and calls the *d* function to mark and explore the connected

landmass. It returns the total number of landmasses found (value of c). Function $SMC(m)$ is the old, havely used function across this book, that converts a 2D matrix (m) into a human-readable string representation. It iterates through the matrix and constructs a string (r) where each row of the matrix is separated by a newline character and elements within each row are separated by spaces. In short, the code prints the total number of islands found by calling $SCAN(a)$ and prints the entire grid a in a human-readable format using the $SMC(a)$ function.

11.1.6 Ex. (212) – Count islands over the matrix and count the characters in each

```
a = num2cell([
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1];
    [1, 0, 1, 0, 1, 1, 0, 0, 1, 1];
    [1, 1, 1, 0, 1, 1, 0, 0, 0, 1];
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 1];
    [1, 1, 1, 0, 1, 1, 1, 0, 1, 0];
    [1, 0, 1, 1, 1, 1, 0, 0, 0, 0];
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1];
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1];
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 1]
    ]);

% Convert to string.
for i = 1:numel(a)
    if a{i} == 0
        a{i} = '0';
    else
        a{i} = '1';
    end
end

b = [
    [+1, 0]; % right.
    [-1, 0]; % left.
    [0, +1]; % up.
    [0, -1]; % down.
    [+1,+1]; % up-right.
    [-1,-1]; % down-left.
    [+1,-1]; % down-right.
    [-1,+1]  % up-left.
    ];

% Cell array of characters
% for marking islands.

q = {'*', '#', '%', '&'};

% Initialize p with zeros
% for each label in q.

p = zeros(1, numel(q));
```

Output:

```
Islands = 3

* 0 * * * * 0 # # #
* 0 * 0 * * 0 0 # #
* * * 0 * * 0 0 0 #
0 0 0 0 * 0 0 0 0 #
* * * 0 * * * 0 # 0
* 0 * * * * 0 0 0 0
* 0 0 0 0 0 0 0 0 %
* 0 * * * * 0 % % %
* * 0 0 0 0 0 0 0 %

Island sizes:[34 8 5]
```

```
[a, p] = SCAN(a, b, q, p);

% Count the number of
% non-zero elements in p.

num_islands = length(p(p > 0));

fprintf('Islands = %d\n\n', num_islands);
fprintf('%s', SMC(a));
fprintf('\nIsland sizes:');
fprintf('%s\n', mat2str(p(p > 0)));

function [a, p] = d(a,i,j,n,m,c,b,q,p)
    if i<1 || j<1 || i>n || j>m || ...
        ~strcmp(a{i,j}, '1')
        return;
    end

    % Mark the cell with
    % a character from q.

    a{i,j} = q{c};
    p(c) = p(c) + 1;

    for k = 1:size(b,1)
       [a, p] = ...
       d(a,i+b(k,1),j+b(k,2),n,m,c,b, q,p);
    end
end

function [a, p] = SCAN(a, b, q, p)
    c = 0;           % Island count.
    n = size(a, 1);  % Rows.
    m = size(a, 2);  % Columns.

    for i = 1:n
        for j = 1:m
            if strcmp(a{i,j}, '1')

                % Increment island count
                c = c + 1;

                if c > numel(q)
                    error('Not enough labels');
                end

                % Initialize size count
                % for the new island.

                p(c) = 0;

                [a, p] = ...
```

```
                d(a,i,j,n,m,c,b,q,p);
            end
        end
    end
end

function r = SMC(m)
    r = '';
    for i = 1:size(m,1)
        for j = 1:size(m,2)
            r = [r, m{i,j}, ' '];
        end
        r = [r, newline];
    end
end
```

Here, the same as in the previous example, the code defines a program to identify and label islands in a binary grid represented by a 2D array *a*. This time, the code also calculates the size of each island and prints the results. The code starts by initializing two 2D arrays, *a* and *b*. Array *a* represents the binary grid, where 1 s indicate land and 0 s indicate water. Array *b* is an array of pairs used to navigate in all eight possible directions (up, down, left, right, and diagonally) from a given cell. Next, it defines an array *q* containing characters ("*", "#", "%", "&") and a zero filled array *p*. Note that the code then prints the following: i) The number of islands found in the grid, which is calculated by the *SCAN* function. ii) The grid itself with islands labeled by characters from the *q* array. iii) An array *p* containing the sizes of each island. There are the same three main functions as before. Function $d(a, i, j, n, m, c, b, q, p)$ is a recursive function used to

traverse the grid and label the islands. It takes as input the grid a, current coordinates (i, j), grid dimensions (n, m), an island counter c, and the references for arrays b, q, p. Next, it checks if the cell is out of bounds or not part of an island (1 indicates land), and if so, it returns. Otherwise, it labels the cell, increments the size of the current island in the p array, and recursively explores neighboring cells in all eight directions. Function *SCAN*(a) iterates through the entire grid and finds the number of islands. For each unvisited land cell (1), it increments the island counter, initializes the size of the island in the p array, and calls the d function to label and explore the island. Function *SMC* from the previous example remains unchanged. The code is now designed to identify and label islands in a binary grid and provides information about the number of islands and their sizes.

11.1.7 Ex. (213) – Count islands and calculate their percentage coverage

```
global p_percent;

a = num2cell([
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1];
    [1, 0, 1, 0, 1, 1, 0, 0, 1, 1];
    [1, 1, 1, 0, 1, 1, 0, 0, 0, 1];
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 1];
```

```
Output:

Islands = 3

* 0 * * * * 0 # # #
* 0 * 0 * * 0 0 # #
* * * 0 * * 0 0 0 #
```

```
    [1, 1, 1, 0, 1, 1, 1, 0, 1, 0];
    [1, 0, 1, 1, 1, 1, 0, 0, 0, 0];
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1];
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1];
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 1]
    ]);

for i = 1:numel(a)
    if a{i} == 0
        a{i} = '0';
    else
        a{i} = '1';
    end
end

b = [
    [+1, 0];  % right.
    [-1, 0];  % left.
    [0, +1];  % up.
    [0, -1];  % down.
    [+1,+1];  % up-right.
    [-1,-1];  % down-left.
    [+1,-1];  % down-right.
    [-1,+1]   % up-left.
    ];

q = {'*','#','%','&','@','$','!','+','^'};
p = zeros(1, numel(q));
p_percent = cell(1, length(q));

[a, p] = SCAN(a, b, q, p);
num_islands = length(p(p > 0));

fprintf('Islands = %d\n\n', num_islands);
fprintf('%s\n', SMC(a));

for i = 1:num_islands
    fprintf('%c %d (%s)\n', ...
             q{i}, p(i), p_percent{i});
end

function [a, p] = d(a,i,j,n,m,c,b,q,p)

    if i < 1 || j < 1 || i > n ...
            || j > m || ~strcmp(a{i,j}, '1')
        return;
    end

    a{i,j} = q{c};
    p(c) = p(c) + 1;

    for k = 1:size(b,1)
        [a, p] = ...
```

```
0 0 0 0 * 0 0 0 0 #
* * * 0 * * * 0 # 0
* 0 * * * * 0 0 0 0
* 0 0 0 0 0 0 0 0 %
* 0 * * * * 0 % % %
* * 0 0 0 0 0 0 0 %

* 34 (38%)
# 8 (9%)
% 5 (6%)
```

```
        d(a, i+b(k,1), j+b(k,2),n,m,c,b,q,p);
    end
end

function [a, p] = SCAN(a, b, q, p)

    global p_percent;

    c = 0;          % Island count.
    n = size(a, 1); % Number of rows.
    m = size(a, 2); % Number of columns.

    for i = 1:n
        for j = 1:m
            if strcmp(a{i,j}, '1')

                c = c + 1;

                if c > numel(q)
                    error('Not enough labels');
                end

                p(c) = 0;
                [a, p] = d(a,i,j,n,m,c,b,q,p);

                p_percent{c} = [num2str( ...
                round(100 * p(c) / (n*m)) ...
                ), '%'];

            end
        end
    end
end

function r = SMC(m)
    r = '';
    for i = 1:size(m,1)
        for j = 1:size(m,2)
            r = [r, m{i,j}, ' '];
        end
        r = [r, newline];
    end
end
```

This latest version of the "island" code adds a couple of new features. It defines a program that processes a 2D array a representing a grid of land and identifies and labels islands on the grid. The code begins by defining a 2D array a, which represents a grid of land with 1 s representing land and 0 s representing water. Another 2D array b is defined, which represents directional offsets. Each element in b is a pair of coordinates that represent movement in different directions (right, left, up, down, etc.). An array q is defined, containing various characters used to label the islands on the grid (more than before). An 2D array p is created to store information about the islands, and a global array

named *p_percent* is also created to later store the island coverage percentages derived from the counts of the islands (i.e., from *p*). Next, the code calls the *SCAN*(*a*) function, which scans the grid to identify and label the islands. It then prints the number of islands found and two representations of the grid: the original grid with islands labeled and a grid representation of island information (please see the output above). Inside, the *SCAN*(*a*) function scans the entire grid, calling *d* when it encounters a land element (1). It keeps track of the number of islands found and calculates the percentage of land each island occupies. Next, it returns the total number of islands. Now, the *d*(*a*, *i*, *j*, *n*, *m*, *c*, *b*, *q*, *p*) function is a recursive function used to mark and label islands on the grid. The function takes the grid *a*, current coordinates *i* and *j*, grid dimensions *n* and *m*, a label *c*, and the references for arrays *b*, *q*, *p*. The very first line inside the function checks if the current position is within bounds and if it is part of an island. If so, it marks the position with the label from *q*, updates island information in *p*, and recursively explores neighboring positions. The *SMC*(*m*) function is used to convert a 2D array *m* into a string with proper formatting. As a small conclusion, the code effectively identifies and labels islands on the grid, calculates their percentage of coverage, and displays the grid with labels and island information. The significance of this code lies in its practical utility in various applications and its educational value in teaching fundamentals. In order to better understand this example, consider an image that is represented as a matrix *a*. This image can represent the development of bacterial colonies. Thus, the above algorithm can accurately tell the number of colonies, their area and more.

11.1.8 Ex. (214) – Show similarities between two strings by sequence alignment

```
% Local sequence alignment
% algorithm and the layout.
% Parameters for sequence
% alignment:

Mat = +2; % Match.
Mis = -1; % Mismatch.
gap = -2;

% Sequences.
s0 = '00111111111001';
s1 = '00000011111111100000';

% Alignment strings.
AA = "";
AM = "";
AB = "";

% Space character.
e = ' ';

% Initialize the score matrix.
```

Output:

```
    00111111111001
    ||||||||||||
00000011111111100000
```

```
n_0 = length(s0) + 1;
n_1 = length(s1) + 1;

% Score matrix.
m = zeros(n_0, n_1);

% Matrix initialization
% and completion.

for i = 1:n_0
    for j = 1:n_1
        if i == 2 && j > 1
            m(i,j) = m(i, j-1) + gap;
        elseif j == 2 && i > 1
            m(i,j) = m(i-1, j) + gap;
        end

        if i > 1 && j > 1
            A = m(i-1, j-1) + ...
                f(s0(i-1), s1(j-1), Mat, Mis);
            B = m(i-1, j) + gap;
            C = m(i, j-1) + gap;
            m(i,j) = max([A, B, C, 0]);
        end
    end
end

% Find the maximum
% score in the matrix.

[MMax, ind] = max(m(:));
[x, y] = ind2sub(size(m), ind);

% Traceback & text alignment.
i = x;
j = y;

while i > 1 && j > 1
    Ai = s0(i-1);
    Bj = s1(j-1);
    A = m(i-1, j-1) + f(Ai, Bj, Mat, Mis);
    B = m(i-1, j) + gap;
    C = m(i, j-1) + gap;

    if m(i, j) == A
        AA = [Ai AA];
        AB = [Bj AB];
        if Ai == Bj
            AM = ['|' AM];
        else
            AM = [e AM];
        end
        i = i - 1;
```

```
        j = j - 1;
    elseif m(i, j) == B
        AA = [Ai AA];
        AB = ['-' AB];
        AM = [e AM];
        i = i - 1;
    else
        AA = ['-' AA];
        AB = [Bj AB];
        AM = [e AM];
        j = j - 1;
    end
end

% Add the unaligned beginnings.
r1 = i - 1;
r2 = j - 1;
AA = [s0(1:r1) AA];
AB = [s1(1:r2) AB];

% Padding for the beginnings.
if r1 > r2
    padding = repmat(e, 1, r1-r2);
    AB = [padding AB];
    AM = [padding AM];
else
    padding = repmat(e, 1, r2-r1);
    AA = [padding AA];
    AM = [padding AM];
end

% Add the unaligned endings.
AA = [AA s0(x:end)];
AB = [AB s1(y:end)];

% Print the alignment
fprintf('%s',AA);
fprintf('\n');
fprintf('%s',AM);
fprintf('\n');
fprintf('%s',AB);

% Matching function.
function s = f(a1, a2, Mat, Mis)
    if a1 == a2
        s = Mat;
    else
        s = Mis;
    end
end
```

The above source code performs sequence alignment using a dynamic programming approach, specifically for pairwise sequence alignment of two strings [2]. The code begins

by defining several variables to be used in the sequence alignment algorithm. These variables include *Mat* (Match), *Mis* (Mismatch), and *gap* penalties, two input sequences (*s0* and *s1*), and three strings (*AA*, *AM*, and *AB*) to store the aligned sequences and matching characters. Note that *AA* means alignment of sequence A, *AM* means the alignment of sequence in the middle (the connection vertical lines between the characters of A and B), and *AB* means alignment of sequence B. Additionally, the code initializes an empty string *e* for placeholder characters (i.e., the space character in this particular case), and array *m* for matrix used in the alignment calculations. It also sets variables *MMax* and *MMin* to store the maximum and minimum values found in the alignment matrix, and initializes *x* and *y* to track their positions. The next section of the code initializes and completes the alignment matrix *m*. It sets up a nested loop to iterate through the matrix, calculating alignment scores based on the dynamic programming algorithm. It uses the variables *A*, *B*, *C*, and *D* to calculate the maximum alignment score at each cell of the matrix. The code keeps track of the maximum and minimum scores, as well as their corresponding positions in the matrix (*x* and *y*). After completing the matrix, the code proceeds to perform traceback to find the aligned sequences. It starts from the position (*x*, *y*) with the maximum score and traces back through the matrix, building the aligned sequences and using the *Mat*, *Mis*, and *gap* penalties as needed. Next, the code adjusts the layout of the aligned sequences, ensuring they have the same length and align correctly. It adds placeholder characters (*e*) to the beginning and end of the sequences if needed to align them properly. The final section of the code prints the aligned sequences (*AA*, *AM*, and *AB*) to the console window, representing the aligned sequences, matching characters, and gaps. Additionally, there is a matching function *f*(*a1*, *a2*) defined in the code, which returns a *Mat* or *Mis* penalty based on whether two characters are equal or not. Overall, this code performs sequence alignment between two input sequences and prints the aligned sequences along with matching characters and gaps. This provided MATLAB code has several practical applications in bioinformatics, computational biology, and related fields. For example, in the field of Genetics, sequence alignment is commonly used to compare DNA, RNA, or protein sequences to identify similarities or differences. This is important in understanding evolutionary relationships, identifying functional elements in genomes, and annotating genes. Also, sequence alignment is crucial for searching biological databases, such as *GenBank* or *UniProt*, to find sequences similar to a query sequence. This is often used to identify potential homologous genes or proteins. Nonetheless, sequence alignment has many applications, from biology to antivirus engines, where the algorithm is used to find common points between multiple files infected with polymorphic viruses; and the cases for applications go on as a function of need and imagination.

Randomnes and Programming

12

Randomness plays a significant role in various aspects of computer programming, and MATLAB, as a versatile computer language, offers powerful tools and techniques to incorporate randomness into software applications [1]. Randomness refers to the concept of unpredictability and uncertainty, essential for tasks like generating random numbers, shuffling data, simulating random events, or creating games and simulations [1]. In MATLAB, developers can use this unpredictability via built-in functions and libraries designed to handle randomization effectively [2]. However, this chapter will make an introduction into the significance of randomness in computer programming, its applications, and how MATLAB empowers scientists and engineers to implement various software projects, improving on reliability.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_12

12.1.1 Ex. (215) – Get complementary array by using random values

```
% understand randomness - get
% complementary array by using
% random values.

a = [1, 0, 0, 1, 1, 1, 0];
disp(mutate(a))

function result = mutate(a)
    m = length(a);
    n = 200; % Number of attempts to mutate.
    s = 0;   % Score.

    for i = 1:n
        s = 0;
        b = zeros(1, m);

        for j = 1:m

            % Generate 0 or 1 randomly.
            b(j) = round(rand(1), 0);

            % If elements are complementary,
            % increase score.

            if b(j) ~= a(j)
                s = s + 1;
            end

            % To check for identical arrays,
            % uncomment the following.

            % if b(j) == a(j)
            %     s = s + 1;
            % end
        end

        % If all elements are
        % complementary, return b.

        if s >= m
            result = b;
            return;
        end
    end

    % If not found by random
    % means, return a message.

    result = 'not found by random means.';
end
```

Output:

```
0,1,1,0,0,0,1
```

The code from above aims to demonstrate randomness and create a complementary array through random values. It starts by defining two arrays, *a* and *b*, with some initial values. The *mutate* function is defined to mutate the array *a* into a complementary array *b* using random values. Inside the function, the code sets *m* as the length of array *a* and variable *n* is set to 200. It initializes a variable *s* (score) to keep track of how many elements in the *b* array differ from the corresponding elements in the *a* array. It then enters a loop that runs *n* times, attempting to create a complementary array. Within this loop, it resets the score *s* to 0 for each iteration. Inside the nested loop, it generates random values (0 or 1) by using the *rand*() function for each element of the array *b*. If the generated value is different from the corresponding element in the array *a*, it increments the score *s*, indicating a difference between the arrays. After generating a random array *b*, it checks if the score *s* is greater than or equal to the length *m* of the array *a*. If this condition is met, it returns the complementary array *b*. If the condition is not met after *n* iterations, it returns the message "not found by random means." Outside the function, the source code prints the result of calling *mutate*(*a*) to the console window, by using the *disp* function.

12.1.2 Ex. (216) – Take the first 20% of the closest solutions (mutation/selection)

```
% understand randomness, take the
% first 20% of the closest solutions.

a = [1,0,0,1,1,1,0];

fit = mutate(a);
fprintf('%s', SMC(fit));

function c = mutate(a)
    m = length(a);
    n = 100;  % number of mutations.
```

Output:

```
0 0 0 1 1 1 0 6
1 0 0 0 1 1 0 6
1 0 0 1 1 1 1 6
1 0 0 1 1 0 0 6
1 0 0 1 0 1 0 6
1 0 0 1 1 1 0 7
1 1 0 1 1 1 0 6
```

```
    p = 20;   % percentage of best to select.
    b = zeros(1, m);
    c = [];   % This will store the mutations.

    % Calculate threshold
    % score to select.

    q = round(m * (1 - p / 100));

    for i = 1:n
        s = 0; % score

        % Create mutation
        b = round(rand(1, m));

        % Compare mutation to
        % original array a.

        for j = 1:m
            % Increase score if matching.
            if b(j) == a(j)
                s = s + 1;
            end
        end

        % If score is high
        % enough, add to c.

        if s >= q
            % Append mutation and score.
            c(end+1, :) = [b, s];
        end
    end
end

function r = SMC(m)
    r = '';
    for i = 1:size(m, 1)
        for j = 1:size(m, 2)
            r = [r, num2str(m(i, j)), ' '];
        end
        r = [r, newline];
    end
end
```

This MATLAB code initializes a binary array *a* (i.e., [1,0,0,1,1,1,0]) and then performs a mutation process on it through the *mutate* function. The *mutate* function creates several mutations of the original array, selects certain mutations based on a scoring mechanism, and stores them in array *c*. The *SMC* function is used to format the output for display. In the code, *a* is the initial binary array to be mutated. The *mutate* function takes this array as input and performs the mutation operation. Within *mutate*, the parameters *m*, *n*, and *p* represent the length of the array, the number of mutations to perform, and the percentage of the best mutations to select, respectively. The array *b* is used to store individual

mutations, and c is an array to accumulate selected mutations based on their scores. The mutation process involves randomly flipping the bits in the array b and then comparing this mutated array with the original array a. Each time an element in b matches the corresponding element in a, a score counter s is incremented. After computing the score for each mutation, the mutation is added to c if its score is greater than or equal to a threshold score q, which is calculated based on the length of a and the percentage p. After performing the mutations, the results are formatted by the *SMC* function. This function concatenates the elements of the mutations and their scores into a string r for display. Each mutation and its score are separated by spaces and new lines for clarity. Next, the script displays the formatted string of selected mutations using the *fprintf* function, which outputs the binary representations of the selected mutations and their scores. Therefore, this stochastic process mimics a simple genetic algorithm where random mutations are generated and selected based on their similarity to an original sequence (the requirement). Note that the expression $c(end + 1,:) = [b, s]$ is used to append a new row to the bottom of an existing matrix or cell array c. The keyword *end* in MATLAB refers to the last index of an array dimension. Thus, $end + 1$ effectively refers to the next index after the last one, which in this case is a new row being added to c. Also, $[b, s]$ by horizontally concatenates the contents of b and s.

12.1.3 Ex. (217) – Find complementary matrix by using stochastic means

```
% Find complementary matrix by using
% stochastic means (uniform distribution).

a = [
    0, 0, 0;
    0, 1, 0;
    0, 0, 0
    ];

disp(SMC(mutate(a)));

function b = mutate(a)
    [n, m] = size(a);
    q = 200;              % number of iterations.
    for k = 1:q
        s = 0;            % score.
        b = zeros(n, m);
        for i = 1:n
            for j = 1:m
                b(i, j) = round(rand);
                if b(i, j) ~= a(i, j)
                    s = s + 1;
                end
            end
        end
```

Output:

```
1 1 1
1 0 1
1 1 1
```

```
        if s >= m * n
            return;
        end
    end
    b = 'not found by random means.';
end

function r = SMC(m)

    % If m is a string,
    % simply return it.

    if ischar(m)
        r = m;
    else

        % Otherwise, format the
        % matrix m into a string
        % representation.

        r = '';
        [rows, cols] = size(m);
        for i = 1:rows
            for j = 1:cols
                r = [r, num2str(m(i, j)), ' '];
            end
            r = [r, newline];
        end
    end
end
```

This code is designed to find a complementary matrix of a given binary 3×3 matrix by using a stochastic approach with a uniform distribution. It comprises two functions: *mutate* and *SMC*. In function *mutate*, the code begins with a binary matrix a, which serves as the target for finding its complementary matrix. The size of a is determined using $[n, m] = size(a)$. The function then enters a loop that will run $q = 200$ times, representing different stochastic attempts to find the complementary matrix. Within each iteration, a temporary matrix b of the same size as a is initialized to zeros. Each element of b is then set to either 0 or 1 randomly, using $round(rand)$, which generates random numbers following a uniform distribution and rounds them to the nearest integer. As b is populated, a score s is computed by incrementing it every time an element in b is not equal to the corresponding element in a. The goal is to create a matrix b that is completely different from a (hence, complementary) where s will equal $m \times n$, the total number of elements in the matrix. If b is found within the q iterations, then function *mutate* returns this matrix immediately. If not, the function concludes that a complementary matrix was "not found by random means" and returns this message. The *SMC* function is a utility to convert the output of *mutate* into a human-readable string format. If the output is a string (indicating that the complementary matrix was not found), it simply returns this string. Otherwise, it formats each element of the matrix b into a string, adding spaces between elements and

newline characters at the end of each row for readability. Next, $disp(SMC(mutate(a)))$ displays the result of the mutate function applied to the initial array a, with the *SMC* function ensuring the output is in a readable format. This code effectively demonstrates a stochastic method to solve a problem, typical in scenarios where deterministic algorithms may not be straightforward or efficient.

12.1.4 Ex. (218) – A two states *Markov Chain* simulator based on letters

```
function [z] = simulate_draws()
    Jar = ["WWBBBBBBBB"; "WWWWWBBBBB"];
    draws = 17;
    z = '';

    a = draw(1, Jar);
    z = z + "Jar W[" + a + "], ";

    for i = 1:draws
        if a == "W"
            a = draw(0, Jar);
            z = z + "Jar W[" + a + "], ";
        else
            a = draw(1, Jar);
            z = z + "Jar B[" + a + "], ";
        end
    end
end

function [ball] = draw(S, Jar)

    % Seeds based on
    % the current time.

    rng('shuffle');
    len = strlength(Jar(S+1,:));
    rc = randi(len);
    ball = extractBetween(Jar(S+1,:), rc, rc);
end
```

Output:

```
Jar W[B],
Jar B[B],
Jar B[W],
Jar W[W],
Jar W[B],
Jar B[W],
Jar W[B],
Jar B[W],
Jar W[B],
```

The point of this implementation is to demonstrate how to simulate a non-uniform random process based on quantities (drawing balls from jars) and keep track of the results [1]. Thus, the code simulates a series of draws from two jars, each containing a mix of white ("W") and black ("B") balls, and keeps track of the sequence of draws. The simulation is encapsulated in two functions, namely *simulate_draws* and *draw*. The *simulate_draws* function is the main function of the simulation. It initializes the contents of two jars in the *Jar* array, where each string represents the contents of a jar. The simulation is set to perform a total of 17 draws ($draws = 17$). The simulation starts with a draw from the first *jar* ($draw(1, Jar)$), assuming the first jar is represented by the first element of *Jar*. The outcome of this draw (either "W" or "B") determines which jar will be used in the next draw. The result of each draw is appended to the string z, which keeps a running record of

the draws and their outcomes. A loop then simulates the remaining draws. If the previous draw resulted in a white ball ("W"), the next draw is from the second jar (*draw*(0, *Jar*)), and vice versa. After each draw, the outcome is recorded in *z*. Next, function *draw* simulates drawing a ball from one of the jars. It first uses *rng*("shuffle') to seed the random number generator based on the current time, ensuring that the sequence of random numbers (and therefore the draws) is different each time the simulation runs. The function then randomly selects a position *rc* in the string representing the contents of the chosen jar. The function *randi*(*len*) is used to generate a random integer between 1 and the length of the string of the jar. Note that *extractBetween*() is a built-in function in MATLAB. Now, the line *extractBetween*(*Jar*($S + 1$,:), *rc*, *rc*) extracts the character at the randomly selected position, which represents the color of the drawn ball. Thus, this code is a simple simulation of a stochastic process involving conditional choices based on previous outcomes. It demonstrates basic concepts of probability and randomness in MATLAB, as well as string manipulation and the use of random number generation functions.

12.1.5 Ex. (219) – A two states *Markov Chain* simulator based on probability values

```
function simulate_jar_draws()
    draws = 8;
    z = '';

    Jar = ["", ""];

    % Fill jars with a specific
    % proportion of W and B.

    Jar = Fill_Jar(Jar, 1, 0.2);
    Jar = Fill_Jar(Jar, 2, 0.6);

    % Initial draw from Jar 2
    % (indexing starts from 1 in MATLAB)

    a = Draw(2, Jar);
    z = strcat(z, " Jar W[", a, "],");

    % Perform the specified number of draws.

    for i = 1:draws
        if a == "W"
            a = Draw(1, Jar); % Draw from Jar 1
            z = strcat(z, " Jar W[", a, "],");
        else
            a = Draw(2, Jar); % Draw from Jar 2
            z = strcat(z, " Jar B[", a, "],");
        end
    end
```

Output:

```
Jar W[B],
Jar B[W],
Jar W[B],
Jar B[W],
Jar W[B],
Jar B[W],
Jar W[B],
Jar B[W],
Jar W[B],
```

```
    disp(z);
end

function ball = Draw(S, Jar)

    % Draw a ball from
    % the specified Jar.

    rc = randi([1, strlength(Jar{S})]);
    ball = extractBetween(Jar{S}, rc, rc);
end

function Jar = Fill_Jar(Jar, S, p)

    % Fill the specified
    % Jar with W and B.

    Balls_W = round(100 * p);
    Balls_B = 100 - Balls_W;

    % Initialize the jars as
    % empty character arrays,
    % and ensure Jar{S} is a
    % character array.

    Jar{S} = '';

    for i = 1:Balls_W
        % Concatenate W characters.
        Jar{S} = [Jar{S} 'W'];
    end
    for i = 1:Balls_B
        % Concatenate B characters.
        Jar{S} = [Jar{S} 'B'];
    end
end
```

In this code, there is a simulation of drawing balls from two jars, each containing white ("W") and black ("B") balls. In other words, it is the same non-uniform random process as shown previously, but this time, instead of a sequence of objects (characters) that sets the probability distribution, the implementation now uses transition probability values to perform the simulation. Thus, this implementation of *simulate_jar_draws* is designed to simulate a process of drawing from two jars with different proportions of "W" (white) and "B" (black) balls. The simulation is defined in several functions that fill the jars, draw balls from them, and record the sequence of draws. The *simulate_jar_draws* function initializes the simulation. It sets up a number of draws and two jars as elements of a string array *Jar*. Each jar is filled with a specific proportion of "W" and "B" balls using the *Fill_Jar* function. The proportion is determined by a parameter *p*, which represents the percentage of "W" balls in a jar. The filling of each jar is done in *Fill_Jar*. This function calculates the number of "W" and "B" balls based on the percentage

p and concatenates these characters to represent the contents of the jar. This is a simple way to model a jar filled with a mix of different types of balls. The drawing process starts from *Jar* 2, and subsequent draws depend on the outcome of the previous draw. If a "W" ball is drawn, the next draw is from *Jar* 1; if a "B" ball is drawn, the next draw is from *Jar* 2. The *Draw* function randomly selects a ball from the specified jar. It generates a random index within the length of the string representing the jar contents and extracts the character at that position, simulating a random draw. As the draws are made, the sequence is recorded in a string *z*, which concatenates information about each draw, including which jar was drawn from and what type of ball was drawn. This results in a string that provides a step-by-step account of the drawing process. Next, the sequence of draws is displayed using the *disp* function, showing the order and outcomes of all draws made during the simulation. Therefore, this source code provides a simple yet effective simulation of a probabilistic process, demonstrating the use of arrays, string manipulation, and basic control structures in MATLAB.

12.1.6 Ex. (220) – Multiply a probability vector with a probability matrix *n* times

```
a = [
    1.0, 0.0, 0.0, 0.0;
    0.5, 0.0, 0.5, 0.0;
    0.0, 0.5, 0.0, 0.5;
    0.0, 0.0, 1.0, 0.0
    ];

v = [
    0, 0, 0, 1;
    0, 0, 0, 0
    ];

c = 5;
n = size(a, 1);
m = size(a, 2);

for k = 1:c

    for i = 1:n
        for j = 1:m
           v(2,j) = v(2,j) + (v(1,i) * a(i,j));
        end
    end

    for i = 1:n
        v(1,i) = v(2,i);
        v(2,i) = 0;
    end

    fprintf('k(%d)=[%f,%f,%f,%f]\n', ...
             k, v(1,1), v(1,2), v(1,3), v(1,4));
end
```

Output:

```
k(1)=[0,0,1,0]
k(2)=[0,0.5,0,0.5]
k(3)=[0.25,0,0.75,0]
k(4)=[0.25,0.375,0,0.375]
k(5)=[0.44,0,0.56,0]
```

The primary use of this code is related to predictions. Thus, it repeatedly performs matrix–vector multiplication c times, accumulating the results in v, and displaying the updated v vector after each iteration. This could be part of a numerical computation or simulation where iterative updates to a vector are necessary, such as in some mathematical or scientific simulations of *Markov Chains*. The code starts by defining two matrices, a and v, as well as two scalar variables, c, and n and m. Matrix a is a 4×4 matrix with specific numerical values. Matrix v is a 2×4 matrix initialized with zeros. Scalar c is set to 5, representing the number of iterations in the subsequent loop. Scalar n is assigned the value of the number of rows in matrix a (which is 4) whereas scalar m is assigned the value of the number of columns in matrix a (which is 4). The code then enters a nested loop structure. There are two outer loops controlled by the variable k, which ranges from 1 to c. These loops are responsible for performing matrix–vector multiplications. Within the loop, there are nested loops controlled by i and j, which iterate through the rows and columns of matrices a and v. Inside these loops, the code performs calculations to update the values in the v matrix based on matrix multiplication between a and v. After each iteration of the k loop, there is a block of code that updates the v matrix for the next iteration. The values in $v(1)$ are updated with the values calculated in the previous iteration, and $v(2)$ is reset to zeros. Next, the code uses the *fprintf* function to print the result of each iteration, showing the values of $v(1)$ in a formatted string. Thus, this code is performing iterative matrix–vector multiplications c times and displaying the results for each iteration.

12.1.7 Ex. (221) – A *Markov Chain* framework for simulation

```
P = { 'A', 'B', 'C', 'D';
      0.00, 0.50, 0.50, 0.00;
      0.33, 0.00, 0.33, 0.33;
      0.00, 1.00, 0.00, 0.00;
      0.00, 0.00, 1.00, 0.00 };

[n, m] = size(P);
Jar = cell(1, m);

for j=2:n
    Jar{j-1} = Fill_Jar(j, P, m);
end

draws = 10;

% Perform the first draw.

a = Draw(1, Jar);
q = "";

% Perform the sequence of draws.
```

Output:

```
Q = CBCBCBCBDC

BBBBBCCCCC
AAACCCDDD
BBBBBBBBBB
CCCCCCCCC
```

```
for i=1:draws
    for j=1:m
        if strcmp(a, P{1, j})
            a = Draw(j, Jar);
            q = q + P{1, j};
            break;
        end
    end
end

% Display the simulation.

disp(['Q = ', q]);

% Display the contents of Jar.

disp(SMC(Jar));

function ball = Draw(S, Jar)
    rc = rand() * strlength(Jar{S});
    ball = extractBetween(Jar{S}, floor(rc) + 1,
floor(rc) + 1);
end

function b = Fill_Jar(S, P, m)
    Ltot = 10;
    b = "";
    for i=1:m
        k = round(Ltot * P{S, i});
        for j=1:k
            b = b + P{1, i};
        end
    end
end

function r = SMC(m)
    r = '';
    for i=1:length(m)
        r = r + m{i} + newline;
    end
end
```

The purpose here is to create a simulation of drawing items from a jar with predefined probabilities and capture the sequence of draws. It demonstrates how to model a random process and calculate outcomes based on specified probabilities. The code starts by defining a two-dimensional array P, representing a probability distribution for drawing items from the jar. It contains items labeled as "A," "B," "C," and "D," along with associated probabilities. Next, it calculates the number of rows (n) and columns (m) in the P array, and it initializes an empty array *Jar*. A variable *draws* is set to 10, indicating the number of draws to be simulated. The code initializes variables a, q, and z for tracking the draws. A loop iterates over the number of draws specified. Within this loop, another loop

iterates over the columns of the *P* array to determine which item is drawn based on the probabilities. The selected item is added to the *q* variable, which records the sequence of draws. There is a *Draw*(*S, Jar*) function that simulates drawing an item based on probabilities specified in a specific row of the *P* array. It randomly selects an item from the jar and returns it. Another function *Fill_Jar*(*S, P, m*) is defined to fill the jar based on probabilities from a specific row of the *P* array. It calculates the number of items to add to the jar for each item type based on the probabilities. The code then prints the contents of the *Jar* array by using the *SMC*(*m*) function. Therefore, the code simulates drawing items from a jar based on specified probabilities and records the sequence of draws in the *q* variable. One last thing to note, is that the above example, is the most primitive version of a *GPT*-like (generative pre-trained transformer-like) system. Once the reader understands the principles, it will get the point of the example by looking at the output.

MATLAB Specific

13

MATLAB, a high-level computer language and interactive environment for numerical computation, visualization, and computer programming, offers a wide array of built-in functions and tools that make it a powerful resource for various computational tasks. Among these features, MATLAB provides functionalities for Base64 encoding and decoding, which are essential for handling data encoding in web communications and data storage. Furthermore, MATLAB supports JSON (JavaScript Object Notation), a lightweight data-interchange format, which is particularly useful for data exchange between servers and web applications. File *Input*/*Output* (I/O) is another strength of MATLAB, with straightforward functions to read and write text files, allowing for easy data manipulation and storage. This capability is crucial for handling large datasets, logging data, or interacting with external data sources. In case of data visualization, MATLAB excels with its extensive capabilities for creating figures and charts. These visualizations are not only powerful for data analysis but also for communicating complex information

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53805-6_13

graphically. The MATLAB plotting tools are highly customizable, ranging from simple line plots to sophisticated 3D visualizations, and they are instrumental in understanding and interpreting data. Much beyond these important numerical computations and data visualization, MATLAB tools allow for developing Graphical User Interfaces (GUIs). These GUIs can include elements like buttons, text areas, and events, providing an interactive way to visualize data and control MATLAB applications. The GUI development environment in MATLAB allows users to build intuitive interfaces for their programs, making the software accessible to a broader range of users, including those without extensive programming experience.

13.1.1 Ex. (222) – *Base64* encoding and decoding via built-in functions

```
ori = "this is a text!";

% Encode a string.
enc = matlab.net.base64encode(ori);

% Decode the string.
dec = matlab.net.base64decode(enc);

disp(['Encoded: ', enc]);
disp(['Decoded: ', dec]);
```

```
Output:

dGhpcyBpcyBhIHRleHQh
this is a text!
```

Base64 encoding is and will likely be indispensable for MATLAB web applications. Many file types can be embedded as text in a MATLAB source code file via Base64 encoding. Thus, within the above script, there are several operations performed on strings using the *base64encode* and *base64decode* methods. First, a variable *ori* is defined and initialized with the string “this is a text!” Then, the *base64encode* method is used to encode the *ori* string into a Base64-encoded string, and the result is stored in a variable called *enc*. Next, the *base64decode* method is used to decode the Base64-encoded string stored in *enc*, and the result is stored in a variable called *dec*. After these encoding and decoding operations, the code displays the results. The first *disp* line displays the Base64-encoded string (*enc*), and the second *disp* line displays the decoded string (*dec*). Thus, this code encodes a string using Base64 encoding and then decodes it back to its original form, demonstrating how to use the *base64encode* and *base64decode* functions for encoding and decoding strings in MATLAB. Note that the *base64encode* and *base64decode* functions are available in MATLAB R2016b or later.

13.1.2 Ex. (223) – A kind of local storage

```
a = ["a", "b", "c"];

b = [
    [0, 1, 0];
    [1, 1, 1];
    [0, 1, 0]
    ];

c = struct("c1", a, "c2", b, "c3", 42);
obj = struct("v1", a, "v2", b, "v3", c);

% Convert object to JSON
% string and store in
% a text file.

txt = jsonencode(obj);
fid = fopen('data.txt', 'w');

if fid == -1
    error('File could not be opened');
end

fwrite(fid, txt, 'char');
fclose(fid);

% Retrieve data from the
% text file and convert
% from JSON string to object.

fid = fopen('data.txt', 'r');

if fid == -1
    error('File could not be opened');
end

txt = fread(fid, '*char')';
fclose(fid);
```

Output:

```
From string:
------------------
{"v1":["a","b",
"c"],"v2":[[0,1,0],
[1,1,1],[0,1,0]],
"v3":{"c1":["a",
"b","c"],"c2":
[[0,1,0],[1,1,1],
[0,1,0]],"c3":42}}

From object:
------------------
b
1
0,1,0,1,1,1,0,1,0
1
```

```
obj = jsondecode(txt);

msg = ['From string:' newline txt newline];
msg = [msg 'From object:' newline];
msg = [msg char(obj.v1(2)) newline];
msg = [msg num2str(obj.v2(1, 2)) newline];
msg = [msg mat2str(obj.v3.c2) newline];
msg = [msg num2str(obj.v3.c2(2, 2)) newline];

disp(msg);
```

The source code provided above is a sequence of operations involving data structure definition, serialization to JSON, file I/O, deserialization from JSON, and data display in a formatted manner. Initially, the code defines several data structures. Variable *a* is a 1×3 array of characters, essentially a string array containing the letters "a", "b", and "c". Variable *b* is a 3×3 numeric matrix with rows [0, 1, 0], [1, 1, 1], and [0, 1, 0]. The variable c is a structure with three fields: c1 containing the array *a*, c2 containing the matrix *b*, and c3 holding the numeric value 42. The structure *obj* is another structure combining all the previous variables *a*, *b*, and *c* under the fields v1, v2, and v3, respectively. Following the data structure definitions, the code serializes the *obj* structure into a JSON string using the *jsonencode* function. This string represents the object in a text format that can be easily stored and transferred. Then, the code opens a file named *data.txt* for writing. If the file cannot be opened, an error is thrown. Upon successfully opening the file, the JSON string is written into it using the *fwrite* function, and the file is closed with *fclose*. The next block of code reopens the *data.txt* file, this time for reading. If the file cannot be opened for reading, an error is thrown. Otherwise, the JSON string is read from the file into the variable *txt* using the *fread* function and then the file is closed. The *jsondecode* function is then used to convert the JSON string back into a MATLAB object, reconstructing the *obj* structure with its original content. The subsequent lines construct a message string *msg* that will display the contents of the JSON string and the deserialized object. It starts with the literal "From string:" followed by the JSON text. The message then appends "From object:" and proceeds to display elements from the deserialized object, including an element from the v1 field (*obj.v1*(2) which is "b"), an element from the v2 matrix (*obj.v2*(1, 2) which is 1), the entire *v3.c2* matrix in text form, and a specific element from that matrix (*obj.v3.c2*(2, 2) which is also 1). To ensure that the numbers and arrays are properly converted to strings, the *num2str* and *mat2str* functions are used where needed. Next, the constructed message *msg* is displayed in the MATLAB Command Window using the *disp* function, showing the contents of the JSON string and the corresponding parts of the deserialized object. The use of newline ensures that each piece of information is displayed on a new line for readability.

13.1.3 Ex. (224) – File I/O

```
% ####### Open the file for reading ########
```

Output:

```
fileID = fopen('test.txt', 'r');

% Check if the file was
% opened successfully.

if fileID == -1
    error('File could not be opened');
end

% Initialize a cell array
% to hold the lines of the file.

lines = {};

% Read the file line by line.

lineNumber = 1;
while ~feof(fileID)
    lines{lineNumber} = fgetl(fileID);
    lineNumber = lineNumber + 1;
end

% Close the file
fclose(fileID);

% Display the contents
% of the file.

disp('Contents of the file:');
for i = 1:length(lines)
    disp(lines{i});
end

% ####### Open the file for writing ########

fileID = fopen('data.txt', 'w');

% Check if the file
% was opened successfully.

if fileID == -1
    error('File could not be opened');
end

% Write a string to the file.
fprintf(fileID, 'This is a line of text.\n');

% Write another string to
% the file just because ...
% we can.

fprintf(fileID, 'This is another line of text.\n');
```

```
This is
content from
test.txt
```

```
% Close the file.
fclose(fileID);

disp('This is content from data.txt');
```

The example is a MATLAB script that makes a proper introduction to file read and write operations. First case, it reads the contents of a text file named *test.txt* and displays those contents in the MATLAB command window. It begins by attempting to open *test.txt* for reading, with the *fopen* function returning a file identifier, *fileID*. This file identifier is used to interact with the file in subsequent operations. The script then checks if the file was opened successfully by examining if *fileID* is -1, which would indicate an error in opening the file. If the file could not be opened, an error message "File could not be opened" is displayed, and the script execution stops. If the file is opened successfully, the script initializes an empty cell array named *lines*. This array will store each line of the text file as a separate cell, preserving the line structure of the original text file. Next, the script enters a *while-loop* that continues to execute as long as the end of the file (*feof*) has not been reached. Inside the loop, the *fgetl* function reads one line from the file associated with *fileID*. Each line read is then stored in the *lines* cell array, with *lineNumber* keeping track of the current index. After storing the line, *lineNumber* is incremented, preparing the index for the next iteration of the loop. Once the end of the file is reached, the loop exits, and the *fclose* function is called to close the file and free up system resources. Closing the file is an important step to avoid resource leaks, which can occur if too many files are left open, especially when running scripts that handle multiple files or very large files. Next, the script displays the contents of the *test.txt* file. It does so by iterating over the *lines* cell array and displaying each line in the MATLAB command window using the *disp* function. This loop ensures that each line of the file is printed on a new line in the command window, mirroring the format of the text file. The second case snippet is for writing text to a file named *data.txt*. It begins by attempting to open the file for writing using the *fopen* function, which is passed the name of the file and the mode "w", indicating that the file should be opened for writing. If the file does not exist, MATLAB will create it. The *fopen* function returns a file identifier, *fileID*, which is a number that MATLAB uses to refer to the file. This identifier is then used in subsequent operations on the file. After the *fopen* call, there is a check to ensure that the file was opened successfully. The *fopen* function will return -1 if the file cannot be opened for some reason (for example, if permission to write in the directory do not exist). If *fileID* is -1, an error is thrown with the message "File could not be opened" (just like before in the first case), and the execution of the script stops. Once the file is successfully opened, the *fprintf* function is used to write strings to the file. The *fprintf* function takes the file identifier, a string to write, and a newline character "\n" indicating the end of a line. In this case, two separate lines of text are written to *data.txt*. After writing to the file, the *fclose* function is called with the file identifier as its argument to close the file. This is an

important step because it releases the file so that other programs can use it, and ensures that all data is actually written to the disk. Next, the script displays a message with *disp*, which outputs the string "This is content from test.txt".

13.1.4 Ex. (225) – Figures and Charts

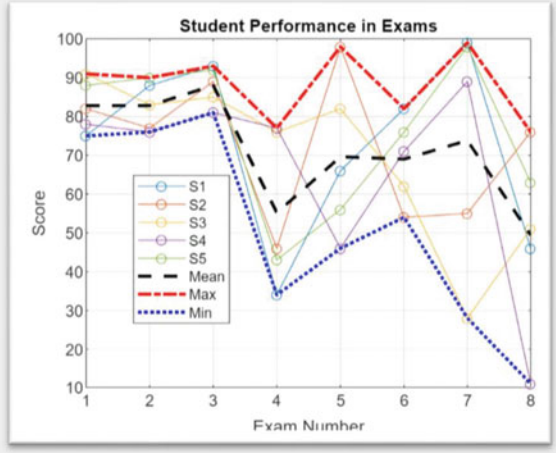
```
% Data for the chart.
data = [23, 45, 66, 77, 44, 33, 99];

% Create a new figure.
figure;

% Plot the data. Note that
% 'r' is the color red.

plot(data, 'r', 'LineWidth', 2);

% Set the axes limits.
axis([1, length(data), 0, max(data)]);

% Set the figure color.
% [0.945, 0.945, 0.945]
% is an approximation
% of '#f1f1f1'.

set(gca, 'color', [0.945, 0.945, 0.945]);
set(gca, 'XColor', 'none'); % Hide x-axis line
set(gca, 'YColor', 'none'); % Hide y-axis line

% Set tick direction to outwards
% box off; and turn off the box
% surrounding the plot.

set(gca, 'TickDir', 'out');
```

Output:

```
% ####### A more complex line chart ########

% Synthetic dataset: rows are
% students, columns are exam
% scores for Exam 1, 2, and 3.
```

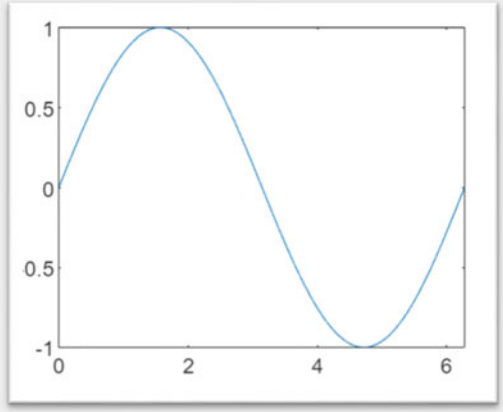
```
d = [75 88 93 34 66 82 99 46;
     82 77 89 46 98 54 55 76;
     91 83 85 76 82 62 28 51;
     78 76 81 77 46 71 89 11;
     88 90 92 43 56 76 98 63];

meanScores = mean(d, 1);   % Mean for each exam.
maxScores = max(d, [], 1); % Max score for each exam.
minScores = min(d, [], 1); % Min score for each exam.

% Plot each exam score
% as a separate line plot.

figure; % Create a new figure window.
plot(d', '-o'); % Transpose data.
title('Student Performance in Exams');
xlabel('Exam Number');
ylabel('Score');

legend('S1', 'S2', 'S3', 'S4', 'S5');

grid on; % Add a grid for easier reading.
hold on; % Hold the current plot.

plot(meanScores, 'LineWidth', 2, 'Color', 'k', 'LineStyle', '--');
plot(maxScores, 'LineWidth', 2, 'Color', 'r', 'LineStyle', '-.');
plot(minScores, 'LineWidth', 2, 'Color', 'b', 'LineStyle', ':');
legend('S1', 'S2', 'S3', 'S4', 'S5', 'Mean', 'Max', 'Min');
```

Output:



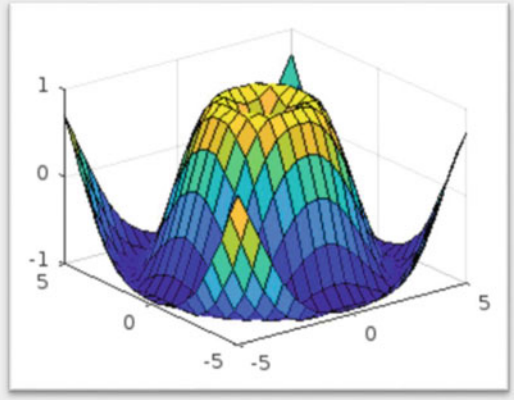
```
% Line plot.
x = 0:pi/100:2*pi;
y = sin(x);
plot(x, y);
```

Line plot output:



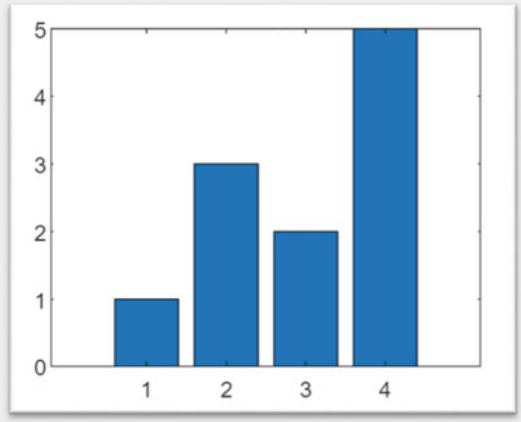

```
% 3D colored surface plot.
[X, Y] = meshgrid(-5:0.5:5);
Z = sin(sqrt(X.^2 + Y.^2));
surf(X, Y, Z);
```

3D colored surface plot output:



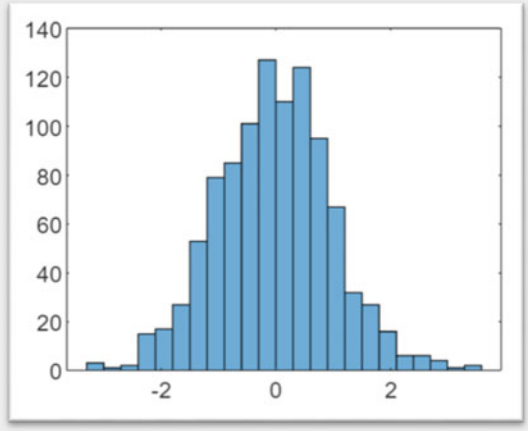
```
% Vertical bar graph.
y = [1, 3, 2, 5];
bar(y);
```

Vertical bar graph output:



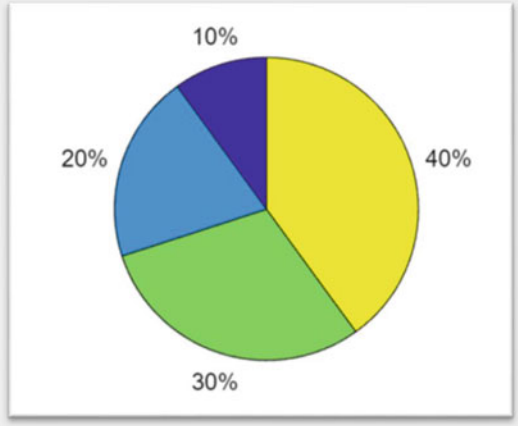

```
% Histogram plot.
data = randn(1000, 1);
histogram(data);
```

Histogram plot output:



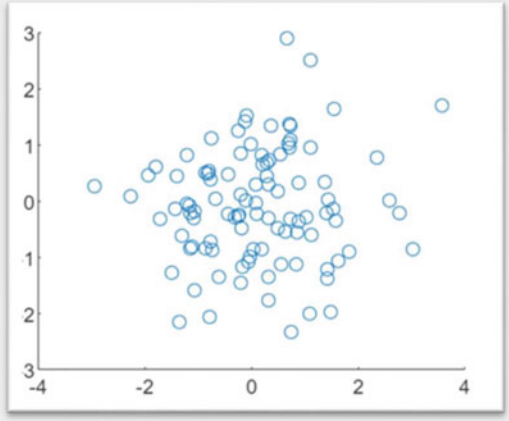
```
% Pie chart.
x = [1 2 3 4];
pie(x);
```

Pie chart output:



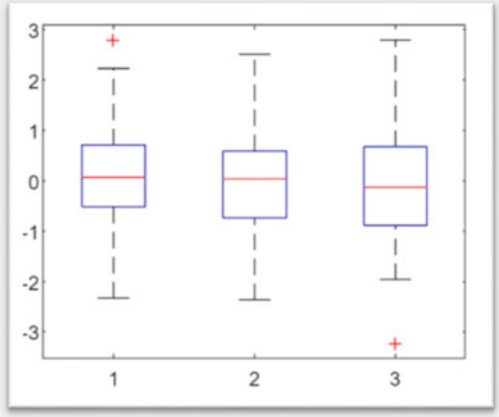

```
% Scatter plot.
x = randn(100, 1);
y = randn(100, 1);
scatter(x, y);
```

Scatter plot output:



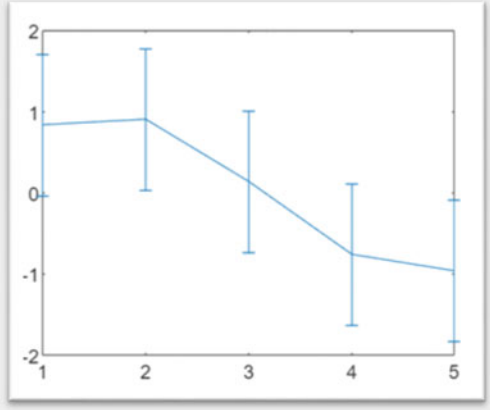
```
% Box plot.
x = randn(100, 3);
boxplot(x);
```

Box plot output:



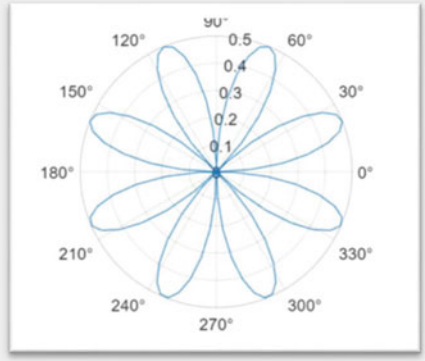

```
% Plot with error bars.
x = 1:5;
y = sin(x);
e = std(y) * ones(size(x));
errorbar(x, y, e);
```

Error bars output:



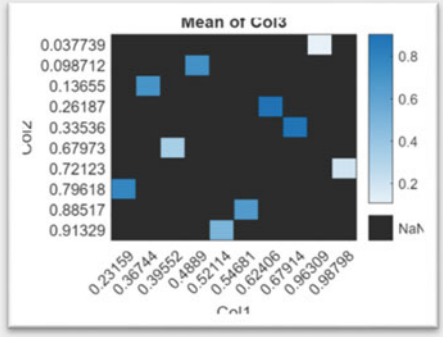
```
% Plot in polar coordinates.
theta = linspace(0, 2*pi, 100);
rho = sin(2*theta).*cos(2*theta);
polarplot(theta, rho);
```

Polar coordinates output:



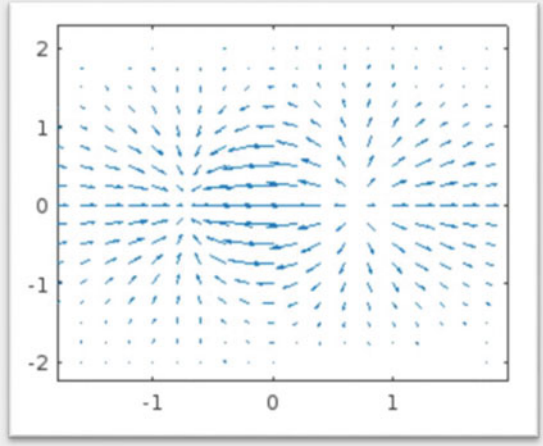

```
% Heatmap.
tbl = array2table(rand(10));
tbl.Properties.RowNames = {'Row1', 'Row2', 'Row3', 'Row4', ...
                           'Row5', 'Row6', 'Row7', 'Row8', ...
                           'Row9', 'Row10'};
tbl.Properties.VariableNames = {'Col1', 'Col2', 'Col3', 'Col4', ...
                                'Col5', 'Col6', 'Col7', 'Col8', ...
                                'Col9', 'Col10'};
heatmap(tbl, 'Col1', 'Col2', 'ColorVariable', 'Col3');
```

Heatmap output:



```
% Quiver plot.
[X, Y] = meshgrid(-2:.2:2, -2:.25:2);
Z = X .* exp(-X.^2 - Y.^2);
[U, V, W] = surfnorm(X, Y, Z);
quiver(X, Y, U, V);
```

Quiver output:



What is a figure and what is a chart in MATLAB? That is the first question. For the MATLAB environment, a figure is essentially the container or the graphical user interface window where plots and charts are displayed. It serves as the canvas for all visualizations, housing various graphical components like axes, panels, and the charts themselves. When the figure function is invoked, MATLAB opens a new window, ready to host any type of graphical representation. This flexibility allows for multiple figures to be opened simultaneously, each containing its own unique set of plots, customized in terms of size , position, and other properties. On the other hand, a chart in MATLAB

refers specifically to the actual plot or graph created within a figure. This is the visual representation of some data, plotted on axes, which are a part of the figure. MATLAB offers a variety of functions to generate different types of charts such as line plots, bar graphs, histograms, scatter plots, and more. These charts are the focal point of the data presentation, where the emphasis is on how the data is plotted, the type of graph, the way it must be annotated, and the overall visual representation of the information. The relationship between a figure and a chart can be thought of as that between a canvas and a painting. The figure provides the space and framework where the charts, akin to artistic strokes, come to life. This distinction is crucial in the approach to graphical representation that MATLAB uses, where the figure acts as a versatile platform for an array of visual data interpretations presented through various types of charts.

13.1 The Minimalistic Chart

In the first part of the example, from a total of three parts, the script starts by defining a set of data points (i.e., "23,45,66,77,44,33,99") that represent the values to be plotted on the chart. These values are stored in an array called *data*. A new figure window is then created using the *figure* command, which will contain the plot. The *plot* function is called with three arguments: the data to plot, a color specification, and a line width specification. The color "r" specifies that the plot should be red, corresponding to the color "#ff0000" in HTML. The "LineWidth" parameter with a value of 2 sets the width of the line in the plot, making it visibly thicker than the default line width. Next, the axis function is used to set the limits of the axes of the plot. The *x*-axis is limited between 1 and the length of the data array, ensuring that each data point is spread evenly along the *x*-axis. The *y*-axis is limited between 0 and the maximum value found in the array *data*, scaled to fit the data within the plot area. This scaling ensures that the highest data point reaches the top of the plot area. The background color of the plot is set using the *set* command, which changes properties of the axes. The color [0.945, 0.945, 0.945] is an RGB approximation of the color "#f1f1f1" in HTML. This makes the background of the plot a light gray color. Additional formatting commands are used to hide the default axes lines and ticks, just to show that it is possible to do so. The *set*(*gca*, 'XColor', 'none') and *set*(*gca*, 'YColor', 'none') commands make the *x*-axis and *y*-axis lines invisible. The "TickDir" property is set to "out" to ensure that any ticks (not removed by the previous commands) point outwards. The box off command removes the box that typically surrounds a MATLAB plot. This series of formatting commands simplifies the appearance of the plot, making it look more like the clean, minimalistic chart.

13.2 A Useful Case for a Line Plot

The second part of the examples includes a MATLAB code that is used for analyzing and visualizing eight exam scores for a group of five students. The *data* matrix contains the scores of five students (each row representing a student) across eight exams (each column representing an exam). The code then performs basic data analysis by calculating the *mean*, *maximum*, and *minimum* scores for each exam. In the visualization part, the code first creates a new figure window. It then plots the exam scores of each student as a separate line plot, with the transpose of the data matrix used to ensure that each score for the exams are plotted as a distinct line. This allows for a clear comparison of performance for each student across the eight exams. The plot is titled "Student Performance in Exams", with the *x*-axis labeled as "Exam Number" and the *y*-axis as "Score". A legend is added to identify each line for a particular student in the plot. The code enhances the plot by adding grid lines for a convenient visualisation of the graph. The *hold on* command is used to retain the current plot, allowing for additional lines to be overlaid on the existing plot. Three more lines are added to the plot representing the *mean*, *maximum*, and *minimum* scores for each exam. These lines are styled differently (dashed for *mean*, dash-dot for *maximum*, and dotted for *minimum*) and are in different colors (black for *mean*, red for *maximum*, and blue for *minimum*) to distinguish them from the individual student scores. The line width is also increased for these summary statistics lines to make them more prominent. Next, the legend is updated to include the *mean*, *max*, and *min* lines, providing a complete view of individual and summary performances in the exams. This kind of visualization is helpful in quickly assessing the overall performance trends in the exams, such as identifying particularly difficult exams (where scores are generally low) or standout performances by students.

13.3 Default Charts

The third and final part of the examples includes various (not all) default charts that are regularly used for data visualization: (a) **Line plot.** The *plot* function in MATLAB is used for creating two-dimensional line plots. It is one of the most common functions for visualizing relationships between two sets of data. With *plot3*, one can extend this visualization into three dimensions, effectively showing the relationship between three sets of

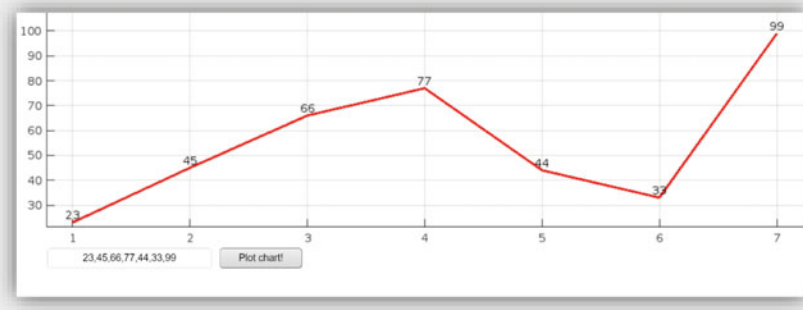
data. (b) **3D colored surface plot.** The *surf* function creates three-dimensional colored surface plots, which are useful for visualizing complex datasets like topographical maps or heat maps. The *mesh* function, on the other hand, generates a wireframe mesh that is transparent, which can be useful for understanding the structure underlying the surface. (c) **Vertical bar graph.** Bar graphs created with *bar* for vertical and *barh* for horizontal orientations are fundamental for categorical data visualization. They are often used to compare the size of different groups or categories. The *bar3* function extends this into three dimensions, allowing for the visualization of categorical data across two independent variables. (d) **Histogram plot.** Histograms, made with the *histogram* function, show the frequency distribution of a dataset and are essential for statistical analysis, helping to quickly understand the distribution and skewness of the data. (e) **Pie chart.** The *pie* and *pie3* functions are perfect for displaying proportional data, where we need to show the contribution of each component to the whole, like in budget allocations or survey results. (f) **Scatter plot.** The *scatter* and *scatter3* functions are used to create scatter plots in two and three dimensions, respectively. These plots are useful when we wish to observe and analyze the distribution and relationship between two or three variables. (g) **Box plot.** Box plots made with *boxplot* are another staple of statistical visualization, providing a visual summary of one or more distributions of the datasets, showing *median*, *quartiles*, and *outliers*. (h) **Plot with error bars.** Error bars are essential for visualizing potential error or uncertainty in measurements or estimates, and the *errorbar* function provides this functionality, which is critical in scientific publications where indicating the margin of error is necessary. (i) **Plot in polar coordinates.** Polar plots, generated with *polarplot*, are used when data is best represented in polar coordinates, often used in fields like signal processing or for representing wind directions and speeds. (j) **Heatmap chart.** Heatmaps, made using the heatmap function, are effective for visualizing complex data where color is used to represent the values, often seen in correlation matrices or as a method for visualizing complex matrices. (k) **Quiver plot.** Vector fields are visualized using *quiver* for two-dimensional fields and *compass* for representing vectors with direction and magnitude on a circular plot, often used in physics and engineering to represent forces or velocities.

13.1.5 Ex. (226) – The Graphical User Interface (GUI)

```
function chart_gui
    % Create a figure window.
    fig = figure('Name', 'Chart Plotter', ...
                 'NumberTitle', 'off', ...
                 'Position', [100, 100, 1100, 400]);

    % Create axes for plotting.
    ax = axes('Parent', fig, ...
              'Units', 'pixels', ...
              'Position', [50, 100, 1000, 300]);

    % Create an input field.
    inputField = uicontrol('Parent', fig, ...
                           'Style', 'edit', ...
                           'String', '23,45,66,77,44,33,99', ...
                           'Position', [50, 50, 200, 25]);

    % Create a button to plot the chart.
    plotButton = uicontrol('Parent', fig, ...
                           'Style', 'pushbutton', ...
                           'String', 'Plot chart!', ...
                           'Position', [260, 50, 100, 25], ...
                           'Callback', @plotChart);

    % Callback function for plotting.
    function plotChart(~, ~)

        % Get the number sequence
        % from the input field.

        numStr = get(inputField, 'String');
        numArray = str2double(strsplit(numStr, ','));

        % Clear existing plot.
        cla(ax);

        % Plot the chart. We use red color
        % (r) and line width of 2.

        plot(ax, numArray, 'r-', 'LineWidth', 2);

        % Adjust the axes limits.

        axis(ax, [0 length(numArray) 0 max(numArray)]);

        % Add a grid for
        % better visibility.
```

```
        grid(ax, 'on');

        % Annotate the plot with values.
        for i = 1:length(numArray)
            text(ax, i, numArray(i), num2str(numArray(i)), ...
                 'HorizontalAlignment', 'center', ...
                 'VerticalAlignment', 'bottom');
        end
    end
end
```

Output:



The example code defines a function named *chart_gui* that, when executed, creates a graphical user interface (GUI) consisting of a plotting area and controls for user interaction. The GUI is intended to replicate the behavior of a web-based interface for plotting a simple chart based on a sequence of numbers. When the *chart_gui* function is called, it begins by creating a figure window. This figure serves as the container for the GUI elements, including the axes for plotting the chart and the controls for user input. The figure function is called with properties that set the window title to "Chart Plotter", turn off the default numbering of the figure title, and specify the position and size of the window on the screen. Within this figure, an axes object is created, which is where the chart will be plotted. This object is positioned within the figure by setting its "Position" property, which takes an array of four values representing the distance from the left, distance from the bottom, width, and height, all in pixels. Next, a user control of the type "edit" is created using the *uicontrol* function. This editable text box allows users to input a sequence of numbers in the form of a string, initially populated with the value "23,45,66,77,44,33,99". The position of this control within the figure is also specified. A button control is then created, also using *uicontrol*, and labeled "Plot chart!". The "Callback" property of the button is set to a function named *plotChart*, which will be executed when the button is clicked. The position of the button is set so that it appears next to the input field. The

plotChart function is defined as a nested function within *chart_gui*, which allows it to access the variables in the parent workspace of the function, such as the *inputField* and *ax* (the axes object). When the button is clicked, *plotChart* reads the string from the input field, splits it at the commas into an array of strings using *strsplit*, and converts those strings to numerical values with *str2double*. This results in an array of numbers, *numArray*, which can be plotted. In the plotting section, the *cla* function is first called to clear any existing content in the axes object, ensuring that the new plot does not overlay an old one. The plot function is then called with the axes handle *ax*, the array of numbers, and plot options specifying a red solid line with a line width of 2. The axes limits are set with the axis function to accommodate the length of the number sequence and the range of the values. The *grid* function adds a grid to the axes to enhance the readability of the plot. Next, a *for-loop* is used to iterate over each number in the sequence. The *text* function is called to place text annotations on the plot at each point, displaying the numeric value at the corresponding position on the chart. The horizontal and vertical alignment properties ensure that the text is centered and appears just above the plotted line. When this script is run, it provides a simple and interactive chart plotting application within MATLAB, allowing for real-time visualization of user-entered data.

References

1. P.A. Gagniuc, *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments.* Synthesis Lectures on Computer Science, Springer Nature, pp. 1–280 (2023)
2. P. Gagniuc, C. Ionescu-Tîrgovişte, Dynamic block allocation for biological sequences. Proc Rom Acad Series B **15**(3), 233–240 (2013)
3. P. Gagniuc, C. Ionescu-Tîrgovişte, C.H. Rădulescu, Automatic growth detection of cell cultures through outlier techniques. Int. J. Comput. Commun. **8**(3), 407–415 (2013)
4. K. Thomas, P. Gagniuc, E. Gagniuc, Moonlighting genes harbor antisense ORFs that encode potential membrane proteins. Sci. Rep. **13**, 12591 (2023)
5. P. Gagniuc et al., A sensitive method for detecting dinucleotide islands and clusters through depth analysis. Romanian J. Diabetes Nutrition Metabolic Diseases **18**(2), 165–70 (2011)
6. P.A. Gagniuc, C. Ionescu-Tirgoviste, E. Gagniuc, M. Militaru, L.C. Nwabudike, B.I. Pavaloiu, A. Vasilăţeanu, N. Goga, G. Drăgoi, I. Popescu, S. Dima, Spectral forecast: a general purpose prediction model as an alternative to classical neural networks. Chaos **30**, 033119–033126 (2020)
7. P.A. Gagniuc, *Algorithms in Bioinformatics: Theory and Implementation, Hoboken* (Wiley, New Jerse, USA, 2021)
8. P.A. Gagniuc, *Markov Chains: From Theory to Implementation and Experimentation*, Hoboken, NJ; USA: Wiley (2017)

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53805-6