SUPPLEMENT No. 1e: MATHEMATICA TUTORIAL

prepared for

NUMERICAL METHODS FOR SCIENTISTS AND ENGINEERS With Pseudocodes

By Zekeriya ALTAÇ

October 2024

Supplement No. 1e: The Mathematica Tutorial

LEARNING OBJECTIVES

The objective of this Mathematica tutorial is to

- present a short summary of the basics of numerical computation with Mathematica;
- describe the implementation of basic programming operations such as loops, accumulators, conditional constructs, etc.;
- explain how to prepare efficient and working functions or modules.

The textbook "*Numerical Methods for Scientists and Engineers: With Pseudocodes*" focuses on implementing numerical methods in science and engineering applications. Supplemental course materials and resources, including C/C++, Fortran 95, Visual Basic, Python, Matlab®, and Mathematica®, are provided to assist the instructors in their teaching activities outside the class.

The aim of this short tutorial is to enable students to acquire the knowledge and skills to convert the pseudocodes given in the text into running Mathematica codes. It is not intended to be a "complete language reference document." *This document does not cover symbolic processing aspects of the software*. The author assumes that the reader is familiar with programming concepts in general and may also be familiar with the Mathematica programming language at the elementary level. In this regard, this tutorial illustrates the conversion and implementation of pseudocode statements (such as formatted/unformatted input/output statements, loops, accumulators, control and conditional constructs, creating and using functions, and subprograms, etc.) to the Mathematica coding.

1 MATHEMATICA BASICS

Wolfram Mathematica is a powerful computational software system, designed for a wide range of tasks, including mathematical computations (performing algebraic manipulations, calculus, and other advanced mathematical operations), data analysis and visualization (analyzing and visualizing data using built-in functions and tools), symbolic computation (handling symbolic math), numerical computation (solving numerical problems with high precision), and programming (writing and running code in the Wolfram Language).

When you start up Mathematica and select *notebook* as a document option, you will have a window displaying the contents of a so-called *notebook*, which also has many of the features of common word processors. Mathematica actually runs two separate programs, referred to as the *front end* and the *kernel*. The user interface is the *front end* (i.e., menus, palettes, notebooks, and any element that accepts input from the keyboard or mouse). It is the interface between the user and the Mathematica *kernel*, which is the program that performs the computations.

1.1 NUMERICAL COMPUTATIONS

A Mathematica notebook is an interactive environment. When you type `1 + 1` and press enter, your command is executed, and the response is printed on the next line. Each command the user types (an input) is displayed in boldface and the output appears in a plain font.

A notebook consists of cells. On the right side of the window, you see cell brackets. Each cell in the notebook shown above is either an input cell, an output cell, or a graphics cell. MATHEMATICA, just like a calculator, can be used interactively using a selection of built-in commands and modules to carry out long and complex calculations. Yet it differs from calculators or computer programs in its ability to evaluate exact results and to compute to an arbitrary degree of precision.

The user can command it simply by typing the command in an input cell and hitting the **Enter** button. Then, MATHEMATICA evaluates the command entered and displays the results.

```
In[1]:= (2 + 3/5)^2     (* performs exact arithmetic                *)
Out[1]:= 289/49         (* result of exact operation                *)
In[2]:= (2. + 3/5)^2    (* uses floating-point arithmetic due to '2.'     *)
Out[2]:= 5.89796        (* approximate result                       *)
In[3]:= N[(2 + 3/5)^2]  (* converts result to approxiamate number using N *)
Out[3]:= 5.89796
```

Note that MATHEMATICA also tracks the session work; each time an expression is entered, it labels the input command lines as `In[1]:=`, `In[2]:=`, `In[3]:=` and the corresponding output lines as `Out[1]=`, `Out[2]=`, `Out[3]=`, as depicted above.

Throughout this tutorial, the input output labels (`In[]:=` or `Out[]:=`) are omitted to save space and to focus on writing script codes.

MATHEMATICA can handle exact integers, rational numbers, and symbolic expressions. For example, the number entered as 1/3 is stored and manipulated as a fraction rather than a floating-point approximation. On the other hand, MATHEMATICA uses floating-point arithmetic if a numerical calculation involves at least one number with a decimal point; e.g., the number `1/3.` is evaluated to give approximately `0.333333`.

MATHEMATICA handles approximate (real) numbers with any number of digits. The *precision of an approximate real number* is the effective number of decimal digits in it that are treated as significant for computations. The *accuracy* is the effective number of these digits that appear to the right of the decimal point. MATHEMATICA allows the user to control both precision and accuracy for numerical computations and evaluates numerical expressions to machine precision using the standard double-precision floating-point format (about 16 significant digits). The user can set the precision or accuracy of numbers or computations using the functions `SetPrecision` and `SetAccuracy`.

1.2 DATA TYPES

MATHEMATICA provides different types of data, such as numbers (Integers, Real, Rational, or Complex numbers), strings, matrices, lists, sets, symbols, patterns, logical, etc., that one can work with.

Integers: MATHEMATICA integers are whole numbers and can be categorized in a few distinct types based on their properties and representations: standard integers (i.e., `4`, `-7`, etc.), rational integers (i.e., `3/4`, `-27/2`, etc.), arbitrary-precision, positive and negative (signed) integers.

Real Numbers: MATHEMATICA represents floating-point numbers using machine precision, which typically corresponds to about 15-17 decimal digits. However, the working precision of real numbers can be set using the `SetPrecision` function or by using `N` with a second argument.

```
    987             (* an exact integer                *)
    987.            (* an approximate real number      *)
```

Complex Numbers: Complex numbers are defined with two parts: a real and an imaginary part. In Mathematica, the basic imaginary unit is $i = \sqrt{-1}$ and denoted by `I`. For example,

```
        3 - 2 I          (* an exact complex number          *)
       3. + 2. I         (* an approximate complex number    *)
```

Strings: In Mathematica, the term string refers to an array of characters. Strings are enclosed in double quotes, e.g., "Hello, World!", "123-13-14-333", etc. Mathematica provides a rich set of functionalities for working with strings, allowing for various manipulations and transformations.

Lists: In Mathematica, a *list*, an ordered collection of elements, is a fundamental data structure used in Mathematica to group objects together. For instance, numerical and personal data such as **`name`**, **`lname`**, **`dob`**, and **`height`** are regarded as lists. Mathematica also provides a large collection of built-in functions for manipulating and analyzing lists of data.

```
     dataset = {{1,3,4}, {5,9,2}, {8,4,4}}       (* list of numerical data  *)
     persons = {"Robert","Alex","Jill","Mary"}   (* list of string data     *)
     pdata = {"Robert","William", 2002, 1.82}    (* list of mixed type data *)
```

Patterns and Rules: Patterns are fundamental to symbolic computation and are used in pattern matching and replacement, e.g., **`x_`** for a pattern, **`x -> y`** for a rule.

Logical Types: The logical data type represents a logical True or False state, which are the two possible outcomes of logical (Boolean) expressions. Some logical Mathematica functions (such as **`NumberQ`**, **`EvenQ`**, etc.), logical comparisons (such as **`>`**, **`==`**, etc.), and logical operators (such as **`And`**, **`Or`**, etc.) return True or False to indicate whether a certain condition was met or not. For example, the statement (5 * 10) $>$ 40 returns a True value.

```
     10>9          (* is True since the logical expression is correct    *)
    (10>9)<(5<=25) (* is False since the logical expression is incorrect *)
    Or[10>9, 5<25] (* is True since the logical expression are correct   *)
```

Mathematica groups the objects that operate on into different types internally as integers, real or complex numbers, or lists. The data type is interpreted by the way the user supplies it to the system. The `Head` function identifies the types of objects. For example, the type of a variable or constant object is requested as follows:

```
    Head[11]     (Integer)     Head[11.]  (Real)       Head[1/5]    (Rational)
    Head[2+3 I]  (Complex)     Head["Hi"]  (String)    Head[{3, 4, 5}]] (List)
```

1.3 IDENTIFIERS IN MATHEMATICA

In Mathematica, identifiers can represent numbers, functions, lists, or any other data structure. An *identifier* is a symbol to which a value, list, or expression can be assigned. Identifiers, which are also case sensitive, may involve uppercase or lowercase letters and Greek symbols in different combinations; however, an identifier must start with a symbol and must not be a 'reserved name' in the system. Special symbols such as , #, $, %, ˆ, &, *, !, ∼, ' and _ cannot be used in names.

In Mathematica, variables and functions are not explicitly typed in the same way as in some other programming languages like C, Java, or Python. Mathematica is a dynamically typed system, which means it does not require or support explicit type declarations for variables or function arguments. The type of an object (whether it's an integer, real number, symbol, list, etc.) is determined automatically at runtime based on the value or structure of the object.

Avoid using single-letters such as `C`, `D`, `E`, `I`, `N`, and `K` when naming objects as these are already reserved for other purposes in MATHEMATICA; e.g., `E`=e, `C`=C (integration constants), `I`=i (imaginary number), `D` for differentiation (`D[x^2,x]=2 x`), `N` for numerical approximation (`N[3/2]=1.5`), and `K` for default index name of a symbolic sum.

1.4 BRACKETING IN MATHEMATICA

In MATHEMATICA, bracketing plays a critical role in distinguishing different types of expressions, input structures, and functions. MATHEMATICA uses four kinds of bracketing, with each kind having different meaning or purpose: parentheses `( )`, square brackets `[ ]`, curly braces `{ }`, and double square brackets `[[ ]]`.

Parentheses `()` are used for grouping expressions to control the order of operations, just like in ordinary mathematical operations. Square brackets `[]` are used for function arguments, i.e., they denote function applications. Curly braces `{}` are used to denote lists or sets of elements. Double Square Brackets `[[ ]]` are used for part extraction parts (accessing elements) from lists, matrices, etc.

```
(a + b * c)              (* ( ) parentheses for grouping  *)
f[x] + f[y]              (* [ ] for functions, f(x) + f(y) *)
{a, b, c}                (* { } for lists                  *)
a[[k]]                   (* [[ ]] for indexing, Part[a,k] *)
```

1.5 THE USE OF SEMICOLON (;) IN MATHEMATICA

The semicolon (`;`) has two primary uses: *separating expressions* and *suppressing output*.

As for *separating expressions*, the semicolon allows multiple expressions to be executed sequentially within the same line or within a block without necessarily displaying the output of each intermediate expression. When expressions are separated by semicolons, MATHEMATICA only returns the result of the final expression (unless the final expression is also followed by a semicolon, in which case no output is shown).

MATHEMATICA automatically displays the result of any expression entered in an input cell, not followed by a semicolon, in the output cell. Adding a semicolon after an expression *suppresses* its *output*, which is especially useful when an operation or assignment needs to be executed without needing to display its result. This way, using a semicolon helps keep the workspace clean by controlling what gets displayed.

```
x = 1;                               (* no output is displayed     *)
y = x (x+3);                         (* no output is displayed     *)
z = y (y - 2);                       (* no output is displayed     *)
x = 1; y = x (x+3); z = y (y - 2)    (* 8 is the displayed output  *)
```

1.6 BUILT-IN MATHEMATICA FUNCTIONS

MATHEMATICA accommodates thousands of easy-to-use built-in functions (or commands) to perform all sorts of tasks, from computing elementary mathematical functions to simplifying an algebraic expression to solving linear and nonlinear equations to graphing 2D or 3D plots.

The built-in commands always begin with a capital letter. Also, the name of a command consisting of more than one word, such as `ArcSin` or `FullSimplify`, is typed with the first letter of each word capitalized without space between the words. The arguments are separated by commas and always enclosed in square brackets. The functions may be typed from the keyboard or entered by using the *palette buttons*, as appear in mathematical equations. The Basic Commands section of the Basic Math Assistant palette includes the most common commands that users resort to, such as summation, differentiation, integration, and so on.

Mathematical Constants: Some of the most commonly used constants are as follows:

`Pi`	($\pi \approx 3.14159$)	`E`	($e \approx 2.71828$)	`I`	($i = \sqrt{-1}$)	`Infinity`	(∞)

Complex Numbers: Consider a complex number defined as $z = a+ib$. Some of the mathematical functions that support evaluation of complex numbers are as follows:

`Re[z]`	(real part)	`Im[z]`	(imaginary part)	`Conjugate[z]`	(conjugate of z)
`Arg[z]`	(argument of z)	`Abs[z]`	(modulus of z)	...	

Numerical Functions: Some of the numerical functions frequently used are given below:

`N[x]`	(gives floating-point value of x)
`Abs[x]`	($\|x\|$, absolute value of x)
`Round[x]`	(integer closest to x)
`Floor[x]`	(greatest integer less than or equal to x)
`Ceiling[x]`	(smallest integer greater than or equal to x)
`Max[`*list*`]/Min[`*list*`]`	(maximum/minimum value in the *list*) ...

Elementary Functions: Some of the most commonly encountered elementary functions are defined as follows:

`Sqrt[x]`	($\sqrt{x}$)	`Exp[x]`	(e^x)	`Log[x]`	($\ln x$)	`Log[b,x]`	($\log_b x$)
`Sin[x]`	($\sin x$)	`Cos[x]`	($\cos x$)	`Tan[x]`	($\tan x$)	`ArcSin[x]`	($\sin^{-1} x$)
`Sinh[x]`	($\sinh x$)	`Tanh[x]`	($\tanh x$)	`ArcSinh[x]`	($\sinh^{-1} x$)	...	

Integer Functions: Some of the most commonly used functions with integer arguments are defined as follows:

`n!`	(factorial of n)
`Mod[x,y]`	(remainder on division of x by y)
`EvenQ[x]`	(test whether x is even, gives True or False)
`OddQ[x]`	(test whether x is odd, gives True or False) ...

Predicates: Predicates are functions that return `True` or `False`, usually based on whether an expression satisfies a certain condition. The predicates are used for testing properties of variables, expressions, numbers, lists, and other structures. Some of the most commonly used predicates are as follows:

`IntegerQ[x]`	(test whether x is a number (integer, real, complex, etc.)
`NumberQ[x]`	(test whether x is an integer)
`ListQ[x]`	(test whether x is a list)
`EvenQ[x]`	(test whether x is even)
`OddQ[x]`	(test whether x is odd) ...

1.7 DEFINING A USER-DEFINED FUNCTION (UDF)

MATHEMATICA provides many built-in functions to perform various calculations, yet we often find ourselves in need of preparing customized (user-defined) functions that are not available in the MATHEMATICA built-ins and that best suit our specific needs.

Defining a *basic user-defined function* in MATHEMATICA is reserved for functions of a single expression. The syntax for a general n variable function is

```
fname[arg1_, arg2_, ···, argn_ ] := expression
```

where *expression* denotes the expression defining the task that the function will carry out, `fname` is a proper (not conflicting with the Mathematica functions and commands) and meaningful function name, and arg_1, arg_2, ..., arg_n denote the arguments (or patterns) of the function that end with the underscore (_). Notice that the delayed assignment sign (`:=`) is used instead of the `=` sign. For now, we suffice with this much information about functions, but we cover the topic of procedural functions in greater detail in Section 7.

Consider the one- and two-variable functions $f(x) = x^3/(4x+3)$ and $g(x,y) = xy/(x^2+y^2)$. They are defined as basic functions in Mathematica as follows:

```
f[x_ ]:= 2 x^2 + x/(4 x + 3)   (* a function with one variable, x        *)
g[x_, y_ ]:= (x - y)/(x + y)   (* a function with two variables, x and y *)
```

Here, `x_` or `y_` is a pattern that matches any expression passed to the function. Each time a function is called by invoking `f[ ]` or `g[ ]`, just like a built-in function, the prescribed computation is carried out and the result is returned. For instance, `f[2]` and `g[4,1]` give `90/11` and `3/5`, respectively.

1.8 COMMENTING

In computer programming, commenting is done to allow *human-readable* descriptions detailing the purpose of some of the expressions and/or to create *in situ* documentation. Mathematica kernel ignores the properly inserted comments or texts. In Mathematica, two methods are used to insert comments in a notebook (or script) file: create a `Text Cell` or insert single- or multi-line comments.

Explanatory text using `Text Cells` can be used to create comments that are visually distinct using text-style cells in a notebook environment. Using a text cell (from the Format menu), one can gain greater control over the formatting of the text, such as italic, bold, or implementing formulas, etc.

The content delimited by `(* ...  *)` syntax is used to create comments that can span multiple lines or just be a single line, as illustrated below. These comments that are ignored by Mathematica during execution are useful for documenting the code.

```
y = 2 * x + 5    (* this is an example of a single line comment  *)
(*  This is an example of
     a comment that spans
     multiple lines  *)
```

1.9 HOW TO USE MATHEMATICA HELP MENU

Documentation Center: Mathematica comes with a large set of documentation that the users can access in a variety of ways. Mathematica documentation center may also be accessed through the **Help** menu. The documentation center provides the descriptions of all built-in functions available in Mathematica, along with examples of their use, links to related tutorials, and so on.

Function pages: When you search for or click on a function (like `Integrate` or `Plot`), a detailed page opens, providing a brief description of the function and its arguments, examples of using the function, optional parameters and settings, explains related functions and concepts, and so on.

Using ? for Quick Help: One can get a quick overview of any function by typing `?fnc`, where `fcn` is the function name in question. This command provides a brief information about its syntax and usage without going through the Documentation Center. For example, `?Integrate` will bring information on `integrate` and `?*Integrate*` where `*`, a "wild card" in such queries, returns a list of built-in symbols that include the string “`Integrate`”.

2 ASSIGNMENT OPERATIONS

In Mathematica, an assignment to a variable (identifier) is to give that variable a specific value and then use the identifier (variable) name to represent that value in subsequent operations. Mathematica provides different types of assignment operators depending on how and when you want the assignment to occur. The types of assignment operations are *immediate assignment* (`=`), *delayed assignment* (`:=`), *compound assignment*, and several more.

An *assignment* denoted by ← (a left-arrow) in pseudocode notation is replaced with `=` or `:=` sign, depending on the nature of the assignment operation. An assignment operation (immediate or delayed) is carried out as follows:

```
Left-hand side  = value         (* immediate assignment *)
Left-hand side  = expression    (* immediate assignment *)
Left-hand side := expression    (* delayed assignment   *)
```

where the *expression* on the rhs can be arithmetic expressions involving arithmetic operators (`+`, `-`, `*`, `/`, `^` (exponentiation), and `Mod`), relational (`>`, `<`, `==`, ...) as well as logical operators (And, Or, Not, ...). In Table 1.1, the arithmetic, relational, and logical operators and their uses have been illustrated with examples.

2.1 IMMEDIATE ASSIGNMENT (=)

In *immediate assignment*, an *expression* on the right-hand side of the "=" sign is evaluated first, and its value is placed at the allocated memory location of the *variable* on the left-hand side. Once the assignment is made, the variable (or symbol) on the left-hand side will always hold that value unless it is changed later. The most recently computed immediate value of *variable* replaces its previous value.

Consider the following code:

```
1   x = 2                   (* x is set to 2                      *)
2   y = x^2 + 2*x           (* y becomes 8                        *)
3   x = z = 3               (* x and z are set to 3               *)
4   y                       (* y is 10 since it was assigned when x was 2  *)
5   Clear[x]                (* removes any value assigned to 'x'  *)
6   {x,y,.. } ={a,b,..}     (* set x to a, y to b, and so on      *)
```

Here, lines 1-3 are examples of immediate assignment. The variable `x` is set to `2` in line 1, and then to `2` in line 3. Line 2 is an example of assignment by an expression, which assigns the result of the *rhs* to the *lhs* (i.e., `y`). As seen in line 3, the same value can be assigned to more than one variable at the same time. Note that in the above example the value of `x` variable is changed with another assignment (in line 3) or removed using the built-in function, `Clear[]`. As shown in line 5, values or symbols can be assigned to several variables at the same time using a list.

2.2 DELAYED ASSIGNMENT (:=)

The delayed assignment operator (`:=`) assigns an expression to a symbol without evaluating it immediately. Instead, the expression on the right-hand side is evaluated whenever the symbol is called, using the most up-to-date values for any variables. The syntax for delayed assignment is given as

```
    lhs := expression   (* delayed assignment to the variable on the lhs *)
```

where `lhs` is a variable or pattern and *expression* is a valid expression.

Delayed assignments are often used for defining functions that involve patterns. This permits us to define

Table 1.1: Arithmetic, equality/relational, logical and assignment operators in MATHEMATICA.

Operator	Description	Example
ARITHMETIC OPERATORS		
±	Addition/subtraction (scalar/element-wise)	`a ± b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`^`	Exponentiation	`a^b`
`!`	factorial	`4! = 24`
RELATIONAL OPERATORS		
`==` or `Equal`	compare for equality	`a == b, Equal[a,b]`
`!=` or `Unequal`	compare for unequality	`a != b, Unequal[a,b]`
`>` or d `Greater`	compare 1'st operand for largeness	`a > b, Greater[a,b]`
`<` or `Less`	compare 1'st operand for smallness	`a < b, Less[a,b]`
`<=` or `LessEqual`	compare 1'st operand for smallness or equality	`a <= b`
`>=` or `GreaterEqual`	compare 1'st operand for largeness or equality	`a >= b`
LOGICAL OPERATORS		
`&&` or `And`	Logical `And` operator	`a&&b` or `And[a,b]`
`||` or `Or`	Logical `Or` operator	`Or[a,b]` or `a||b`
`!` or `Not[]`	Logical `Not` operator	`Nor[a]` or `!a`
COMPOUND OPERATORS		
±=	Addition (`p = p ± q`) assignment	`p ±= q`
`*=`	Multiplication (`p = p * q`) assignment	`p *= q`
`/=`	Division (`p = p /q`) assignment	`p /= q`

`++` (as *prefix* or *postfix*)	Increment operator, which increments the value of the operand by 1	`i++` or `Increment[i]`
		`++i` or `PreIncrement[i]`

`--` (as *prefix* or *postfix*)	Decrement operator, which decrements the value of the operand by 1	`i--` or `Decrement[i]`
		`--i` or `PreDecrement[i]`

a function where the evaluation of the function body occurs when it is called with specific arguments, *see* **Section 1.7**. In the following code, line 2 is an example of delayed assignment.

```
    x = 2               (* x is set to 2                          *)
    y := x^2 + 2*x      (* y becomes 8                            *)
    x = 3               (* x is set to 3                          *)
    y                   (* y becomes 15 since x has changed to 3 *)
```

Note that, in this case, `y` in line 4 is only evaluated when called (invoked), which depends on the current value of `x`. When `x` changes, `y` reflects the new value when it is evaluated.

2.3 COMPOUND ASSIGNMENTS

In computer programming, the same variable is commonly observed on the left- and right-hand sides as in `x = x + y` (i.e., $x \leftarrow x + y$) and so on. In MATHEMATICA, such expression can be compressed as `x += y`, where the operator `+=` now denotes an assignment operator (*see* **Table 1.1**). The arithmetic operators (`+`, `-`, `*`, and `/`) have corresponding assignment operators: `+=` (`AddTo[x,dx]`, $x = x + dx$), `-=` (`SubtractFrom[x,dx]`, $x = x - dx$), `*=` (`TimesBy[x,dx]`, $x = x * dx$), and `/=` (`DivideBy[x,dx]`, $x = x/dx$).

Consider the following expressions:

```
1  X = 0      (* X is initialized by zero                          *)
2  X +=2      (* Equivalent to X = X+2, add 2 to X, X becomes 2    *)
3  X = X + 4  (* Add 4 to X (equivalent to X += 4), X becomes 7    *)
4  X -=3      (* Equivalent to X = X-3, subtract 3 from X, X becomes 3  *)
```

In line 1, `X` is initialized to zero, which makes the memory value of `X` zero. In line 2, the memory value of `X` is substituted in the rhs, which updates the value of `X` as 2. In line 3, the memory value of `X` is substituted in the rhs, which updates the value of `X` as 6. Finally, in line 4, the rhs is evaluated first (`6-36=3`), and the result is placed in the memory location of `X`.

The increment operator, `Increment[x]` or `x++`, increases the value of `x` by `1`, returning the old value of `x`. However, `PreIncrement[x]` or `++x` increases the value of `x` by `1`, returning the new value of `x`. On the other hand, the decrement operator, `Decrement[x]` or `x--`, decreases the value of `x` by `1`, returning the old value of `x`. `PreDecrement[x]` or `--x` decreases the value of `x` by `1`, returning the new value of `x`.

Consider the following code segment. In line 2, the displayed result is `0` but stored as `X=1`. In line 3, the displayed and stored results of incrementing are `X=2`. In line 4, the displayed and stored results of decrementing are `X=1`. In line 4, the displayed result is `1` but stored as `X=0`.

```
1  X = 0     (* X is initialized to zero                          *)
2  X++       (* Equivalent to X = X +1, adds 1 to X, but displayes old X  *)
3  ++X       (* Equivalent to X = X+1, adds 1 to X, displayes new X      *)
4  --X       (* Equivalent to X = X-1, subtracts 1 from X, displays new X *)
5  X--       (* Equivalent to X = X-1, subtracts 1 from X, displays old X *)
```

The above operations may be carried out using the functions `Increment[x]`, `PreIncrement[x]`, `Decrement[x]`, and `PreDecrement[x]`.

2.4 CLEARING VALUES OR REMOVING VARIABLES

It is important to realize that values assigned to variables or symbols are permanent during a work session. There are cases where it is necessary to clear (or remove) a symbol, value, or rule in Mathematica, especially when working on large-scale projects. For instance, when a variable name or symbol is used for something else, the previous value can lead to conflicts and errors. Clearing the symbol resets it and ensures no prior definitions interfere with the code. Also, in iterative algorithms, clearing the variables allows resetting the state before a new calculation, ensuring that only the latest definitions or input values are used.

Here are the most common commands for such tasks:

```
x = 2; y=3             (* x is set to 2, y is set to 3        *)
x =.                   (* unset the value of x or Unset[x]    *)
Clear[x]               (* clear the value of x                *)
Clear[x, y, ...]       (* clear the values of x, y, ...       *)
ClearAll[x, y, ...]    (* clear the values of x, y, ...       *)
Remove[x]              (* remove x completely                 *)
Remove[x, y, ...]      (* remove x, y, ... completely         *)
```

`Unset[]` (or `=.`) unsets a specific value or definition of a symbol. `Clear[]` function clears any values, definitions, and rules associated with the given symbol(s), but keeps the symbol itself, while `ClearAll[]` not only clears values but also attributes, messages, and any other definitions associated with the symbol(s). `Clear[]` clears values of variables or symbols, but the variables still exist as symbols. The `Remove[]` function completely removes the symbol(s) from the session, as if they never existed. This function deletes the variables entirely, so they are no longer valid symbols in the session.

2.5 REPLACING OR INSERTING VALUES

If we want to calculate the value of an *expression* for a specific variable, we can use the replacement method. Suppose the *expression* involves x and we desire to evaluate it for $x = a$. The replacement operator (`/.`) is used to insert values into expressions. Following illustrates how to replace or insert value(s) to variable(s) in an expression:

```
expression/.x->a          (* replace x by a                            *)
u=expression/.x->a        (* replace x by a and assign the result to u *)
expression/.{x->a,y->b}  (* replace several variables, x→a and y→b   *)
```

where the inserted value of a symbol can also be any expression, not just a number.

3 EXPRESSIONS

An expression in Mathematica is the basic unit of computation. Everything in Mathematica—whether it is a number, variable, function, or more complex structure—is treated as an expression. *Numeric expressions* contain numbers and arithmetic operations. *Symbolic expressions* involve symbols and may not evaluate to a specific number until further input is provided. *Function expressions* allow defining functions or rules, while *logical expressions* require the use of logical operators like `&&`, `||`, `==`, etc.

Following illustrates various aspects of expressions in Mathematica:

```
2 + 5/4-2*Sin[Pi/3]        (* numerical expression  *)
x^3+4*x*Exp[x]             (* symbolic expression   *)
f[x_] := x^2               (* a function expression *)
(x > 2)(a && b)            (* logical expression    *)
```

3.1 ALGEBRAIC EXPRESSIONS

Arithmetic expressions are straightforward and follow typical mathematical conventions. Arithmetic operators in such expressions are used to perform numeric computations such as adding, subtracting, multiplying, and dividing two scalar or array variables. Following are examples of arithmetic expressions:

```
a = {1, 2, -1}; b = {3, -2, 4}; (* initialize arrays a and b *)
a + b            (* addition of a and b gives {4,0,3}  *)
a^2              (* gives {1, 4, 1}  *)
a*b              (* multiplication gives {3, -4, -4}   *)
a/b              (* division gives {1/3, -1, -(1/4)}   *)
Exp[a]           (* exponentiation gives {E, E^2, 1/E} *)
```

In arrays, arithmetic operators work on corresponding elements of the same size arrays. If one operand is a scalar, Mathematica applies the scalar to every element of the other operand, or applying functions directly to collections. These kinds of operations are referred to as *pointwise* or *element-wise* operations.

3.2 RELATIONAL AND LOGICAL EXPRESSIONS

Branching in a computer program causes a computer to execute a different block of instructions, deviating from its default behavior of executing instructions sequentially. Such operations are carried out using a combination of relational and/or logical operators. A logical operator is defined as an operator on numeric, character, or logical data that yields a logical result (*see* **Table 1.1**).

Branching structures are controlled by *logical variables* and *logical operations*. A *logical_expression* is a combination of logical constants, logical variables, and logical operators. Logical operators evaluate relational expressions to either `True` or `False`. There are two basic types of logical operators: relational operators (<, >, <=, >=, ==, !=) and combinational (logical) operators (`And[], Or[], Not[], ...`). MATHEMATICA supports a large collection of relational operator characters and Boolean expressions. Logical calculations are carried out with an immediate assignment statement and logical operators can be employed for both programming and mathematical operations.

Logical operators are used in a program together with relational operators to control the flow of the program. The `And` and `Or` operators connect at least one pair of conditional expressions. The order of evaluation of `And` and `Or` is from left to right. The logical `Not` operator (`!`) negates the value of a Boolean operand, and it is a unary operator. Every operator in standard form (`&&`), `||`, `==`, ...) has a functional form (`And`, `Or`, `Equal`, ...) as well.

Let L_1 and L_2 be two logical prepositions. In order for L_1 `&&` L_2 to be True, both L_1 and L_2 must be True. In order for L_1 `||` L_2 to be True, it is sufficient to have either L_1 or L_2 to be True. When using `Not[]` `(!)`, a unary operator, in any logical statement, the logic value is changed to True when it is False or changed to False when it is True. These operators can be used to combine multiple expressions. For given `x=5`, `y=9`, `a=18`, and `b=3`, we can construct the following logical expressions:

```
And[x < y, y < a, a > x]        (* True  *)
(x < y)&&(y < a)&&(a > x)       (* Equivalent to the above expression *)
(And[x < y, y > a, a >= x       (* False *)
(x < y)&&(y > a)&&(a >= x)      (* Equivalent to the above expression *)
Or[And[x < y, y < a], a < b]    (* True  *)
(x < y)&&(y < a)||(a < b)        (* Equivalent to the above expression *)
Or[Or[x > y,Not[y < a]],a < b] {(* False  *)
Or[x > y, Not[y > a], a < b]    (* Equivalent to the above expression *)
(x > y)||!(y < a)||(a < b)      (* Equivalent to the above expression *)
```

The logical operators return `True` or `False`, or remain unchanged if the value of a logical expression can not be determined due to processing symbolic data. `TrueQ` can be used to verify if a result is indeed returned by applying it as `TrueQ[a&&b]`, `TrueQ[And[a,b]]`, or `TrueQ[a||b]`, and so on.

4 CONTROLLING INPUT AND OUTPUT

In MATHEMATICA, input and output control is highly customizable, allowing the user to format, manipulate, and structure the input and output in various ways. MATHEMATICA provides multiple functions and methods to tailor the interaction to the user's needs.

In most cases, input data may be supplied by assigning it to a (input) variable using `'='` before an expression or statement, as shown in line 1. However, in large programs, input data for a variable may be read interactively using the `Input` command. `Input` command returns the expression it reads. Though it may vary from one computer system to another, it typically works through a dialog box. With a notebook front end, `Input` by default puts up a dialog window with a standard appearance. The prompt given can be text, graphics, or any expression, as in line 3.

```
n = 10                                  (* Initialize n to 10         *)
Sum[i^2 , {i,n}]               (* sum of i^2 from 1 to n      *)
x = Input["Enter x (0 <= x <= 1) "] (* accepts x thru a dialog box *)
Print[x]                                (* displays entered x value    *)
```

The **Print** function generates a cell with style **Print**. It has the capability of printing any expression, including graphics and dynamic objects. The syntax is as follows:

```
      Print[ expr1, expr2, ··· ]
```

where $expr_i$ are constants, variables, lists, or any other data type. Several numerical or string expressions can be concatenated together. With a text-based interface, **Print** ends its output with a single newline (line feed). **Column** is used to arrange to have expressions on several lines.

Consider the following code:

```
Print[x, y, z]                     (* displays     xyz     *)
x = 1; y = 2; z = 3; Print[x, y, z] (* displays     123     *)
Print[x + y + z]                   (* displays     6       *)
Print[a + b + c]                   (* displays  a + b + c  *)
Print[u^2, "=", x+y+z]             (* displays     u^2=6   *)
```

Note that, in line 1, the variables **x**, **y**, and **z** are printed with no prior numerical assignments. Therefore, the displayed output is **xyz**, with no spaces in between. In line 2, the variables are assigned numerical values and then displayed as **123**, with no spaces between them. The numbers are assumed integers since they are specified without a decimal point. In line 3, since **x**, **y**, and **z** have preassigned numerical values, their sum is printed. But, in line 4, the symbolic sum is displayed **a+b+c** since **a**, **b**, and **c** have no pre-assigned numerical values. Finally, in line 4, the **Print** statement involves symbolic, string, and numerical expressions, which are displayed in the order typed.

4.1 FORMATTING NUMBERS

An improved way of displaying output is to combine the string conversion commands with **Print**. In this regard, Mathematica has a number of flexible mechanisms for displaying numbers of any magnitude and precision to optimize readability. The output form and the number of digits that are displayed can be controlled by using **NumberForm** or **EngineeringForm**.

```
NumberForm[expr]          (* prints value of expr with default option   *)
NumberForm[expr,n]        (* prints value of exp> to n-digit precision  *)
NumberForm[expr,{n,f}]    (* to the right of the decimal point          *)
EngineeringForm[expr]     (* prints value of expr> with default option  *)
EngineeringForm[expr,n]   (* prints value of expr to n-digit precision  *)
PaddedForm[expr,n]        (* prints value of expr to n digits, padded
                             with leading spaces if necessary           *)
PaddedForm[expr,{n,f}]    (* prints value of expr with exactly f digits
                             after decimal point                        *)
TableForm[list]           (* prints a matrix in table form              *)
```

Here, **n** is the number of digit precision, and **f** is the number of digits to the right of the decimal point.

Consider the following code statements:

```
s = Sum[1./i^2, {i,1,5}]     (* displays 1.46361                          *)
NumberForm[s]                (* displays 1.46361, 5 digits, default opt.  *)
NumberForm[s,15]             (* displays 1.46361111111111, 15 digits      *)
NumberForm[s,{15,10}]        (* displays 1.4636111111 10 digits after '.' *)
EngineeringForm[s/1000]      (* displays 1.46361×10^-3,  default option    *)
EngineeringForm[s/1000,10]   (* displays 1.463611111×10^-3, 10 digits aft. *)
```

In the first of the following examples, the numbers are padded with spaces to make room for up to 6 digits. The second expression prints each number with room for a total of 5 digits and with 3 digits after the decimal point. Finally, the matrix is printed with the default setting.

```
PaddedForm[{123, 12345, 12}, 6]          (* displays with default opts  *)
PaddedForm[{1.23, 1.2345, 1.2}, {5, 3}] (* displays using w=5, d=3     *)
matr={{6, 36, 216}, {7, 49, 343},  {8, 64, 512}}
TableForm[matr]                          (* displays with default opts  *)
```

The output is

```
{   123,  12345,    12}
{ 1.235,  1.235, 1.200}
 6    36    216
 7    49    343
 8    64    512
```

MATHEMATICA provides many more forms for numbers and strings; the readers can access these forms at reference.wolfram.com.

5 VECTOR AND MATRIX (ARRAY) OPERATIONS

An array (variable with subscripts) is another data form that MATHEMATICA uses to store and manipulate data. In MATHEMATICA, a *vector* is a one-dimensional array with n-elements denoted with a *list* of n elements that are *not* lists themselves, i.e., $\{v_1,\, v_2, \ldots, v_n\}$. On the other hand, a matrix (a two-dimensional array) consisting of elements arranged in rows and columns is represented by lists of lists, i.e., $\{\{a_{11}, \ldots, a_{1n}\}$, $\{a_{21}, \ldots, a_{2n}\}, \ldots, \{a_{n1}, \ldots, a_{nn}\}\}$.

Vectors and matrices can always mix numbers and arbitrary symbolic or algebraic elements. The use of arrays is useful in applications where information and data are stored in tabular form.

5.1 CREATING AND USING ARRAYS

To create an array of values, the Matrix/Table palette may be used, or the elements of a vector are themselves typed by hand in the suitable format, as shown below:

```
v = {1, 3, 7}                             (* v is a vector of length 3 *)
A = {{1, -1, 1}, {1, 0, 2}, {0, 1, -2}}   (* A is a matrix of size 3x3 *)
```

Note that a 3×3 matrix is created by replacing each element of a vector of length 3 with the matrix rows (i.e., a vector of three elements).

The following built-in functions are used to build or create vectors:

```
Table[f,{i,n}]      (* build a vector by evaluating f with i = 1,...,n *)
Array[a,n]          (* build a vector of the form a[1], a[2], .., a[n] *)
Range[n]            (* create the list {1,2,3, ...,n}                  *)
Range[n1, n2]       (* create the list {n1, n1 + 1, n1 + 2, ..., n2}   *)
Range[n1, n1, dn]   (* create the list {n1, n1 + dn, n1 + 2dn, ..., n2} *)
```

Mathematica functions for building or creating matrices are:

```
Array[a,{m,n}]          (* build an mxn matrix with i,j'th element a[i,j] *)
IdentityMatrix[n]       (* generates an nxn identity matrix                *)
DiagonalMatrix[list]    (* generates a square matrix with the elements in
                           list on the main diagonal                      *)
ZeroMatrix[m,n]         (* creates an mxn matrix filled with zeros        *)
ConstantArray[c,{m,n}]  (* creates an mxn matrix filled with constant c   *)
```

Implementation of these functions is depicted in the following examples:

```
Table[i^2,{i,3}]        (* displays {1, 4, 9}                  *)
Array[b,3]              (* displays {b[1], b[2], b[3]}         *)
Array[a,{2,3}]          (* displays {{a[1,1], a[1,2], a[1,3]},
                           {a[2, 1], a[2, 2], a[2, 3]}}        *)
Range[3]                (* displays {1,2,3}                    *)
Range[5,10]             (* displays {5, 6, 7, 8, 9, 10}        *)
Range[1,2,0.25]         (* displays {1., 1.25, 1.5, 1.75, 2.} *)
IdentityMatrix[n]       (* displays {{1, 0}, {0, 1}}           *)
ConstantArray[3,{2,2}]  (* displays {{3, 3}, {3, 3}}           *)
```

5.2 EXTRACTING A RANGE OR SLICES FROM ARRAYS

Mathematica provides numerous functions for manipulating lists (arrays). Specific rows/columns or submatrices can be extracted using part extraction functions, primarily using **Part** (or shorthand **[[ ]]**). Consider a vector **v** and a matrix **A**. Below are some of the available Mathematica functions for extracting or slicing operations.

```
v[[i]]         (* or Part[v,i] gives the i'th element in the vector v     *)
v[[i;;j]]      (* extracts elements i thru j of vector v                  *)
A[[i,j]]       (* or Part[A,i,j] gives the i,j'th element in the matrix A *)
A[[i]]         (* or Part[A,i] gives the i'th row in the matrix A   |     *)
A[[i,All]]     (* same as above                                           *)
A[[All,j]]     (* or Part[A,All,j] gives the j'th column in the matrix A  *)
A[[i,All]]=c   (* resets i'th row of matrix A to c                        *)
A[[i,j]]=c     (* reset (i,j)'th element of matrix A to c                 *)
A[[i1;;i2,j1;;j2]] (* extracting submatrix, i1<=i<=i2, j1<=j<=j2          *)
Diagonal[A]    (* extracts the diagonal of matrix A                       *)
Drop[A,{n,m}]  (* deletes the rows n<=i<=m                                *)
```

Also, multiple elements can be extracted by specifying the indices as a list of positions, i.e., **matrix[[position1, position2, ...]]**. Implementation of element extractions is depicted in the following examples:

```
v = Range[10]    (* create the vector v = {1,2,...,10} and the matrix A   *)
A= {{1, 2, 3}, {2, 3, 4}, {3, 4, 5}}
v[[3]]           (* displays 3                                            *)
v[[4;;7]]        (* displays {4, 5, 6, 7}                                 *)
A[[2,3]]         (* or Part[A,2,3] displays 4                             *)
A[[3]]           (* displays {3, 4, 5}, the 3rd row                       *)
A[[2,All]]       (* displays {2, 3, 4}, the 2nd row                       *)
A[[All,1]]       (* displays {1, 2, 3}, the 1st column                    *)
A[[3,All]]=5     (* resets 3rd row of matrix A to 5                       *)
A[[1,3]]=8       (* reset (1,3)'th element of matrix A to 8               *)
A[[2;;3,2;;3]]   (* displays {{3, 4}, {4, 5}}                             *)
Diagonal[A]      (* displays the diagonal elements {1, 3, 5}              *)
Drop[A,{2,3}]    (* displays {{1, 2, 3}} by deleting 2nd and 3rd rows     *)
Drop[A,None,{2,3}]  (* displays the 1st column, {{1}, {2}, {3}}           *)
```

5.3 INSERTING OR DELETING A ROW (OR COLUMN) INTO OR FROM A MATRIX

Inserting rows or columns into matrices or tables can be achieved using various MATHEMATICA functions. A row can be inserted at a specific position using the `Insert` function, as follows:

```
Insert[A,elem,n]        (* inserts elem at position n in matrix A       *)
Insert[A,elem,{n,m}]    (* inserts elem at position n,m in matrix A     *)
Drop[A,n]               (* gives A with its first n elements dropped    *)
Drop[A,{n}]             (* gives A with its first n'th elements dropped *)
Drop[A,{m,n}]           (* gives A with elements m through n dropped    *)
Delete[A,n]             (* deletes the element at n'th position         *)
Delete[A,{m,n}]         (* deletes the part at position {i,j}           *)
```

Here are some examples of inserting or deleting rows from a matrix:

```
A = {{1, 1}, {2, 2}, {3, 3}} (* define matrix A of size 3x2              *)
b = {4, 4}                   (* define vector b of length 2              *)
Ab = Insert[A, b, 2]         (* inserts b at position 2 to get a 4x2
                                matrix, {{1, 1}, {4, 4}, {2, 2}, {3, 3}} *)
Ac = Insert[A, 99, {2, 3}]   (* gives {{1, 1}, {2, 2, 99}, {3, 3}}       *)          Delet
```

Inserting a column is achieved by using `Transpose` and `Insert`; that is, the matrix is transposed first. A row is inserted (which is a column in the original matrix), and then it is transposed back.

5.4 HANDLING MATRIX OPERATIONS

MATHEMATICA provides a rich set of built-in functions for managing and handling arrays (vectors and matrices). Since MATHEMATICA uses lists to represent vectors and matrices, there is no need to distinguish between *row* and *column* vectors. MATHEMATICA allows element-by-element (*element-wise*) operations. Thus, arithmetic operations (addition, subtraction, and scalar multiplication) with matrices and vectors of the same sizes are quite straightforward and often resemble common mathematical notation. Below are examples of key operations you can perform:

```
1   u = {1, 2, 3};          (* define vectors u and v of length 3         *)
2   v = {5, 4, 6};
3   u + v                   (* performs u+v and displays the result       *)
4   A = {{1, 2}, {3, 4}};   (* define matrices A and B of size 2x2        *)
5   B = {{2, 1}, {6, 8}};
6   A + B               (* performs A+B and displays the result           *)
7   2 A                 (* performs 2*A scalar multiplication             *)
8   3*v                 (* performs 3*v scalar multiplication             *)
9   2*B - 3 A           (* performs 2B - 3A matrix operation              *)
10  u*v                 (* displays {5, 8, 18}, element-wise op.          *)
11  v/u                 (* displays {5, 2, 2}, element-wise op.           *)
12  A*B                 (* displays {{2, 4}, {18, 32}}, element-wise op.  *)
13  B/A                 (* displays {{2, 1}, {2, 2}}, element-wise op.    *)
```

Note that the vectors `u` and `v` and the matrices `A` and `B` have the same size. The additions and subtractions in rows 3, 6, and 9 are done element-by-element, which conforms with the definitions for addition, subtraction, and scalar multiplication of vectors and matrices. However, the multiplication and division operations with vectors and matrices depicted in lines 10-13 are performed element-wise.

Using the matrix and vector definitions above, the following are examples of *element-wise* operations:

```
1/u            (* gives {1, 1/2, 1/3}                              *)
Sin[v*Pi/2]    (* gives {1, 0, 0}                                  *)
Exp[A]         (* gives {{e, e^2}, {e^3, e^3}}           *)
Log[B]         (* gives {{Log[2], 0}, {Log[6], Log[8]}}            *)
```

Matrix multiplication can be done using the `Dot[]` function (or the `.` operator). This allows matrix-matrix, vector-vector, or matrix-vector multiplication of matrices and vectors of conforming dimensions. A short list of other MATHEMATICA functions designed for matrix manipulations is given below:

```
Cross[u,v]        (* computes cross product of thevectors u and v      *)
Det[A]            (* computes the determinant of the matrix A          *)
Dot[u,v]          (* computes matrix multiplication two conformable
                     matrices or dot product of two vectors            *)
A.B               (* shorthand notation of dot product of A and B      *)
Dimensions[v]     (* gives the dimensions of a vector or matrix        *)
Eigenvalues[A]    (* gives the eigenvalues of matrix A                 *)
Eigenvectors[A]   (* gives the eigenvectors of matrix A                *)
Norm[A]           (* computes L2 norm of a vector or matrix expression *)
Normalize[v]      (* normalizes vector v to unit vector                *)
Inverse[A]        (* computes the inverse of the matrix A              *)
Length[v]         (* length of (number of elements in) the vector v    *)
Tr[A]             (* Trace elements of matrix A                        *)
Transpose[A]      (* finds transpose of the matrix A                   *)
MatrixPower[A,n]  (* finds the matrix $\mathbf{A}^n$                   *)
```

MATHEMATICA provides many more built-in commands dealing with vector, matrix, and linear algebra operations. The reader is referred to reference.Wolfram.com for more information.

Notice that `A*A` and `A*B` are element-wise multiplications, not multiplication in matrix operation sense, i.e., not `Dot[A,B]` or `A.B`. Similarly, computing the integer powers of a matrix (i.e., $\mathtt{A}^n$) should be carried out by using `MatrixPower[A,n]`, as $\mathtt{A}^n$ or `A^n` returns all elements raised to the n'th power in place.

In Mathematica, both `MatrixForm` and `TableForm` can be used for displaying *lists* in a structured and readable way, but they have different formatting styles. In this regard, a vector or matrix can be displayed in equation form using the `MatrixForm` (displays with brackets) or `TableForm` (displays without brackets), as shown below:

$$\mathtt{TableForm[A]=}\quad \begin{matrix} 1 & -1 & 1 \\ 1 & 0 & 2 \\ 0 & 1 & -2 \end{matrix} \qquad \text{or} \qquad \mathtt{MatrixForm[A]=}\quad \begin{pmatrix} 1 & -1 & 1 \\ 1 & 0 & 2 \\ 0 & 1 & -2 \end{pmatrix}$$

One can fine-tune or modify the display formats of the lists using the customization options as well as additional Mathematica functions. Here are some key `options` to customize how matrices are displayed:

```
TableForm [list, options]
MatrixForm[list, options]
```

The `MatrixForm` is a visual formatting wrapper, and it is used to *display matrices* in a readable, traditional form. When used as a tool to display matrices, it does not affect the internal structure of the matrix or how Mathematica performs operations on it. But it is not advised to be used directly within an expression for operations (like matrix multiplication, addition, etc.), as it may interfere with the computation. Use the `MatrixForm` whenever you want to see the final product of your matrix operation. Both `MatrixForm` and `TableForm` can be used as postfix commands, i.e., the `A//MatrixForm` command displays in matrix form with default options.

The current or default options of the display functions can be accessed as follows:

```
Options[TableForm]      (* options or default settings can be changed *)
{TableAlignments -> Automatic, TableDepth -> ∞, TableDirections -> Column,
   TableHeadings -> None, TableSpacing -> Automatic }
```

These optional arguments are the same for both display functions.

`TableDirections` defines the direction (horizontal or vertical) in the consecutive dimensions: `Column` (default) specifies that successive dimensions should be arranged alternately as columns and rows, with the first dimension arranged as columns, and `Row` takes the first dimension to be arranged as rows.

`TableAlignments` specifies how entries in each dimension should be aligned. The alignment options for columns and rows are `Left`, `Center`, and `Right`, and `Bottom`, `Center`, and `Top`, respectively.

`TableHeadings` specifies how the labels are to be printed for entries in each dimension of a table or matrix. Option, `None` (default) does not assign labels in any dimension. `Automatic` gives successive integer labels for each entry in each dimension, or labels can be assigned with a list of strings.

`TableSpacings` specifies how many spaces should be left as `TableSpacing->{s1,s2,...}`.

In the following example, the columns are printed as rows:

```
TableForm[{{a, b}, {c, d}, {e, f}}, TableDirections -> Row]
```

$$\begin{array}{ccc} a & c & e \\ b & d & f \end{array}$$

In the next example, a 4×3 rectangular matrix consisting of the factorials of numbers from 1 to 12 is formed and displayed with the `TableForm`, with the columns right-aligned and the rows centered.

```
TableForm[Partition[Range[12]!, 3], TableAlignments -> {Right, Center}]
```

```
   1     2     6
  12    24    72
 540   432   860
2880  5798  6479
```

Headings for the rows or columns can also be specified as follows:

```
TableForm[{{a, b}, {c, d}, {e, f}}, TableHeadings -> {{"row1", "row2",
    "row3"}, {"col1", "col2"}}]
```

which displays

	col1	col2
row1	a	b
row2	c	d
row3	e	f

6 MATHEMATICA PROGRAMMING

Mathematica Programming is the process of combining several built-in functions or commands together to perform a task that no existing single Mathematica function or command can perform. The so-called *Wolfram Language* is designed especially for this purpose.

6.1 CONTROL CONSTRUCTION: Do LOOP

The `Do` constructions are best suited for situations where a task is repeated a number of times without accumulating or modifying complex results. The syntax is

```
Do[expr,n]              (* evaluates expr n times                    *)
Do[expr, {i,n}]         (* evaluates expr for i = 1, 2, , ..., n      *)
Do[expr,{i,m,n}]        (* evaluates expr for i = m, m + 1, , ..., n  *)
Do[expr, {i,m,n,di}]    (* evaluates expr for i = m, m + di, , ..., n *)
```

Here, i is the loop index variable, n and m denote the first and last values of the index, respectively, and di is increment size. Note that the increment size di need not be `integer`; the loop can be applied to any real between n and m. A `Do` loop is repeated the total of |`(n-m+di)/di`| times.

Following are examples of various `Do`-constructions:

```
Do[expr, {m, 0, 5}]          (* m takes the values of 0, 1, 2, 3, 4, 5 *)
Do[expr, {i, -3, 1}]         (* i takes the values of -3, -2, -1, 0, 1 *)
Do[expr, {i, 0, 9, 3}]       (* i takes the values of 0, 3, 6, 9       *)
Do[expr, {k, 5, 1, -1}]      (* k takes the values of 5, 4, 3, 2, 1    *)
Do[expr, {p, 10, 3, -2}]     (* p takes the values of 10, 8, 6, 4      *)
Do[expr, {u, 0.1, 0.5, 0.3}] (* u takes the values of 0.1, 0.4         *)
```

Note that if $n > m$ and di is negative, the loop will not execute. In general, the statements in *expr* are separated by semicolons and evaluated sequentially. If algorithms necessitate, expressions can be inserted before and after any one of the inner `Do`'s.

As in the case of other programming languages, nesting `Do` loops at as many levels as necessary is allowed. An example of nested loops is given as follows:

```
Do[              (* an example of three nested loops *)
    Do[
          Do[expressions,  {i, 0, 9, 3}],
    {j, -3, 2}],
{m, 0, 5}]
```

If the algorithm permits, the `Do` loops can be cast in a more compact form (with a single `Do`) as follows:

```
Do[ expressions, {i, 0, 9, 3}, {j, -3, 2}, {m, 0, 5}]
```

The examples of multiple `Do` loops are given below:

```
Do[Print["x=",x]; Do[Print[" i=",i], {i,1,2}], {x,1,3}] (* see Output 1 *)
Do[Print["x=",x," i=",i], {i,1,2}, {x,1,5,2}]           (* see Output 2 *)
Do[Print[i, " ",j], {i,0,2}, {j,0,i}]                   (* see Output 3 *)
```

Following are the displayed outputs of the `Do` constructions above:

```
Output 1            Output 2            Output 3
x=1                 x=1 i=1             0 0
 i=1                x=3 i=1             1 0
 i=2                x=5 i=1             1 1
x=2                 x=1 i=2             2 0
 i=1                x=3 i=2             2 1
 i=2                x=5 i=2             2 2
x=3
 i=1
 i=2
```

This construction is generally more readable for simple tasks because it clearly shows the range of iteration without the added syntax. Also, the `Do` constructions are slightly more efficient for simple iterations because it does not involve the extra overhead of testing conditions and updating the counter explicitly, as with `For`.

6.2 CONTROL CONSTRUCTION: `For` LOOP

In MATHEMATICA, both `For` and `Do` loops allow for repeated execution of a block of code, but they have different structures and are generally used for different types of tasks. The `For` loop is more flexible, similar to a for loop in other languages (equivalent to the **For** loop of pseudocode), and allows you to explicitly define initialization, a loop condition, and an increment step.

The `For` loop syntax is given below:

```
            For [start, condition, inc, expression]
```

where *start* denotes the initialization of the loop index (e.g., `n`=0, `k`=2, etc.), *condition* is a logical expression to be checked at each iteration (e.g., `n`<=99, `j`>5, etc.), and *inc* denotes the increments (or decrements) needed (e.g., `n`++, `k`--, etc.) for updating the loop index after each iteration. The `for` loop executes the *expression* repeatedly with increments (or decrements) of *inc* until the *condition* becomes `False`. As in the `Do` loop, the code statements in the *expression* block are separated by semicolons and executed sequentially.

An example of a `For` loop is illustrated below:

```
 For[i = 1, i <= n, i++, Print[i]]  (* <=  Do[Print[i], {i, 1, n}] *)
```

Note that this loop works similar to a `Do` loop; it starts with the initial value of `i=1` and updates the loop variable `i` with the increments of `1` after each iteration. The expression to be executed at each iteration is `Print[i]`, i.e., prints the value of `i`.

Following are examples of various `For`-constructions:

```
 For[m= 0, m <= 5, m++, expr]     (* m takes the values of 0, 1, 2, 3, 4, 5 *)
 For[i=-3, i < 2, i++, expr]]     (* i takes the values of -3, -2, -1, 0, 1 *)
 For[i= 0, i<= 9, i+= 3, expr]    (* i takes the values of 0, 3, 6, 9       *)
 For[k= 5, k > 0, k--, expr]      (* k takes the values of 5, 4, 3, 2, 1    *)
 For[p=10, p>= 3, p-= 2, expr]    (* p takes the values of 10, 8, 6, 4      *)
 For[u=0.1, u<= 0.5, u+=0.3,expr] (* u takes the values of 0.1, 0.4         *)
```

If a `Break[]` is generated in the *expression* body, the `For` loop exits. Also, when `Continue[]` is encountered, the loop exits the *expression* body to continue the loop by evaluating `di`.

The following loop displays `i` from `i = 1` to `i = 4` (i.e., `i<5`) with increments of 1 (i.e., `i++`).

```
  For[i = 1, i < 5, i++, Print["i=", i]]
```

In the following example, the loop runs from `i = 12` down to `i = 1` with decrements of 1 (i.e., `i--`). However, a `Break` is produced when `i < 10` (i.e., `i = 9`). Thus, the result of this operation becomes `9`.

```
  For[i = 12, i > 0, i--,
     If[ i < 10 , Break[] ];   (* Break when the condition is True *)
   ];
  Print[i]                     (* displays the final result => 9   *)
```

The `For` constructs are suitable for cases where the user needs to have control over the behavior of the loop, e.g., varying increments, complex stopping conditions, or dynamically modify the loop variable, and so on. The `For` construct is preferable when some results in a variable or list need to be accumulated.

6.3 CONDITIONAL CONSTRUCTION: While LOOP

A **While** loop is needed when a specific block of statements is repeated (for an unspecified number of times) until a condition is met; it is equivalent to **While** construction of the pseudocode. The While loop syntax is as follows:

```
While[ condition , expression ]
```

The loop evaluates the *condition* and then the *expression* repetitively until the *condition* becomes **False**. If **Break[]** is generated in the **expression**, the **While** loop terminates. If a **Continue[]** is encountered, the statements following it are skipped, and the loop is returned to the top to continue processing.

In the following example, the block statements are given with lines 3-5, and the results are displayed for **n=1, 2, 3**.

```
1  n = 1; s = 0;    (* initialize s and n *)
2  While[n < 4,     (* so long as n<4, repeat the block statements *)
3     s = s + 2;    (* increment s by 2 *)
4     Print["n=", n, " s=", s];   (* displayn & s *)
5     n++           (* increment n by 1 *)
6    ]
```

The **While** and **For** loops in Mathematica are similar to the control structures **while** and **for** in C and C++. But the roles of comma and semicolon are reversed in **For** loops of Mathematica relative to C language ones.

6.4 CONDITIONAL CONSTRUCTION: If

As conditional structures, If-structure permits one or two statement blocks to run separately when a *condition* is met or not. The syntax for the If constructions is given as follows:

```
If [ condition,  TrueExpression ]        (or)
If [ condition,  TrueExpression,  FalseExpression ]
```

The **If** structure may take only one argument, i.e., presented above with only *TrueExpression* block. In this case, the value for **False** is taken to be **Null**. However, in general, an **If** construct typically takes two additional arguments besides a condition. Here, *TrueExpression* and *FalseExpression* are evaluated if the *condition* is **True** or **False**, respectively.

Consider the following constructions for evaluating the square root of an input value of **X**. The first **If** construct handles only **X**$\geqslant 0$ cases. This construct will do its job so long as **X** is positive or zero. The second construct informs the processor what action to take in either case.

```
If [ X >= 0 ,
   X2 = N[Sqrt[X]];        (* Only X values of X $\ge$ 0 are processed *)
   Print["Square-root of X is ",X2]
   ];

If [ X >= 0 ,
   X2 = N[Sqrt[X]];        (* Only X values of X $\ge$ 0 are processed *)
   Print["Square-root of X is ",X2] ,
   Print["Illegal argument, X < 0"]   (* Displays a warning for X < 0 *)
   ];
```

In the `If` structure, there is another possibility that the *condition* will not evaluate to either `True` or `False`. Such cases are not considered as errors, and the `If` operator returns itself; for example, in the following, a comparison cannot be realized since `u` is unknown.

```
x = 5; If[x > u, "yes", "no"]
If[5 > u, "yes", "no"] (* returns the command itself *)
```

6.5 CONDITIONAL CONSTRUCTIONS: `Which` AND `Switch`

A more flexible construct that generalizes nested `If`-structures involves either `Which` or `Switch`. These structures are especially useful when one of several paths through an algorithm based on the value of a particular *expression* must be selected.

The general syntax for the `Which` construct is given as

```
Which [ condition_1,   result_1,
        condition_2,   result_2,
           ...    ,      ...   ,
           True   ,    result_n ]
```

This construction evaluates each of the $condition_i$ in turn, returning the value of the $result_i$, corresponding to the first one that yields `True`.

The general syntax for the `Switch` construct is given as

```
Switch [ expression,
         pattern_1,   result_1,
         pattern_2,   result_2,
            ...   ,     ...   ,
         pattern_n,   result_n,
             _    ,   result_{n+1}, ];
```

This construction evaluates *expression* first and then compares it with each of the $pattern_i$ in turn, evaluating and returning the $result_i$ corresponding to the first match found. If none of the patterns match the *expression*, the *Switch* is returned unevaluated.

Following examples depict two case blocks; however, one can have as many cases as needed, but only one *default block*.

```
Which [ x < -1, -1,              (* if x < -1      , result=-1        *)
      -1 <= x <= 1, Sin[Pi x],   (* if -1 <= x <= 1, result=sin(pi*x) *)
          True    ,  1]          (* otherwise,       result= 1        *)

Switch[val,                               (* valid data, val>=0        *)
    Alternatives[0,1], Print["0 or 1"],   (* displays '0 or 1'         *)
          2   , Print[val]      ,         (* displays '2' if val=2     *)
          _   , Print[" >2 "] ]           (* displays                  *)
```

7 PROGRAMS AS FUNCTIONS

7.1 USER-DEFINED FUNCTIONS (UDFs)

A user-defined function in Mathematica is a function coded (i.e., defined) by the user to perform a specific task, usually not available in the built-in functions. These functions provide reusable codes and make the programs modular. The syntax for a general n-variable function is

```
fname[arg1_, arg2_, ···, argn_ ] := function body
```

where **fname** is a proper and suitable function name, arg_1, arg_2, ..., arg_n are the function arguments (i.e., names of independent variables), and the *function body* contains the function expression, which may be either a single-line expression or a compound function.

Following are examples of one-, two-, and three-variable functions:

```
f[x1_] := x1 Exp[-2 x1] - Sin[Pi x1]   (* f(x1) = x1 e^{-2x1} - sin(πx1)     *)
Z[x1_, x2_]:= Cos[ Exp[x1^2+x2^2]]     (* Z(x1,x2) = cos(exp(x1^2 + x2^2))   *)
UnitVec[v_]:= v/Sqrt[v.v]              (* gives the unit vector of v   *)
```

When a function involves more than one expression (also known as auxiliary functions), then the syntax becomes

```
fname[arg1_, arg2_, ···, argn_] := ( expr1;  expr2; ···; exprn )
```

The left-hand side remains the same while $expr_1$, $expr_2$, and so on are expressions (also called *auxiliary* functions) defining the user-defined function. Expression, separated by semicolons, are enclosed in round brackets.

The following function definition corresponds to the series sum $SS(n,p) = \sum_{k=1}^{n} k^p$. The definition involves several expressions enclosed with round brackets, where **s** is protected in local context.

```
SS[n_, p_] := (s = 0; For[k = 1, k <= n, k++, s = s + k^p]; s)
SS[10, 1]    (* gives 55  *)
SS[10, 2]    (* gives 385 *)
```

This function could have also been defined as follows:

```
SS[n_, p_] := Sum[k^p, {k,1,n}]    (* using built-in function form *)
SS[n_, p_] := ∑_{k=1}^{n} k^p      (* using math pallet form       *)
```

7.2 RECURSIVE FUNCTIONS

Mathematica permits recursive programming style through the use of *recursive functions*. A recursive function can repeatedly call itself. Consider the following indexed recursive function **fx**:

```
fx[argument_] :=    expression
```

where *expression* involves **fx[arg]**.

Some mathematical formulas are available in recursive form, such as $a_0 = \alpha$, and $a_n = f(a_{n-1})$ for $n =$ 1, 2, ... and so on. Also, some mathematical procedures may also be computed recursively. For instance, if $f_n = n!$ can be cast as a recursive function by setting $f_0 = 1$ and defining $f_n = n * f_{n-1}$. The **n!** (**fact(n)**) can also be programmed as a recursive function as follows:

```
    fact[0] = 1;                (* set up initial value      *)
    fact[n_] := n fact[n]       (* set up recursive function *)
```

Here **fact[5]** is computed as **fact[5]=5 fact[4]**, **fact[5]=5 *4* fact[3]**, until **fact[0]=1]** is reached.

Dynamic Programming: In dynamic programming, a recursive function stores the computed values. The syntax is given below:

```
    fx[argument_] :=  fx[argument] = expression
```

Saving the values of the function with **fx[n_] := fx[n] = ...** is called *caching*.

Now let us write a recursive function to generate Fibonacci numbers, where each number is the sum of the two preceding numbers mathematically defined as $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 2$. Using dynamic programming, we can define **Fibn** as

```
    Fibno[1] = 1;    (* initialize for n = 1 *)
    Fibno[2] = 1;    (* initialize for n = 2 *)
    Fibno[n] := Fibno[n] = Fibno[n-1] + Fibno[n-2] (* recursive rule for n>2 *)
    Fibno[6]         (* for n=8, it yields 8 *)
    ?Fibno         (* investigating function Fibno's status gives *)
    Fibno[1] = 1
    Fibno[2] = 1
    Fibno[3] = 2
    Fibno[4] = 3
    Fibno[5] = 5
    Fibno[6] = 8
    Fibno[n_] := Fibno[n] = Fibno[n - 1] + Fibno[n - 2]
```

This means that after calculating **Fibno[6]**, all the numbers up to **n=6** are actually available as while computing **Fibno[6]**.

Nonindexed Recursive Formulas: Nonindexed recursive formulas can be defined in the same way. For example, consider writing our own rule for the logarithm function to expand products and powers. Such a definition can be accomplished as follows:

```
    loga[x_ y_] := loga[x] + loga[y]
    loga[x_^ y_] := y loga[x]
```

In this example, the first definition requires the argument to be in the product form, while the second definition applies to arguments of the power form. These definitions are recursive in that they call themselves. For example,

```
    loga[2^x 3^y z^2]
    x loga[2] + y loga[3] + 2 loga[z]
```

Having defined the recursive rules, the rules are applied every time **loga** is invoked in an expression until the result no longer changes.

8 MODULES PROGRAMMING

Numerical algorithms often require performing a task numerous times to accomplish an intended objective, which is sometimes not built into the MATHEMATICA packages. To simplify matters, it is generally desired to collect all the statements under one function or module, which also makes large notebooks easier to manage.

In MATHEMATICA, a specific task in a more complicated program is often prepared as a module. The modules in general have input and output arguments, which are communicated with the calling program. A **Module** localizes the variable names to avoid the name conflicts between the local and global names used in a program. That is why it is more suitable to prepare and use modules that define variables locally, i.e., define variables in an isolated part of the main or other subprograms. In such *localized environments*, the variables are allowed to have different sets of values (from the global definitions) and are used locally. These local environments are called Modules.

The general syntax for a **Module** is given as follows:

$$\texttt{Module[}\{arg_1,\ arg_2,\ \cdots,\ arg_n\},\quad expressions\texttt{]}$$

This means that $|arg_1|$, $|arg_2|$, and so on in the **expression** will be treated as local variables only. The value returned by **Module** is the value returned by the last operator in the body of the *expressions* unless an explicit **Return[]** statement is used within the body of **Module** in which case, the argument of **Return[arg]** is returned.

A **Module** allows setting up local variables with identifiers unique to the module. Within the module, one can define variables to have local values that are valid only in the module and can perform various operations on those variables within the module. However, if the module is constructed as follows, the *local variables* $|arg_1|$, $|arg_2|$, etc. can be set to predefined values of $|a_1|$, $|arg_2|$, and so on.

Alternatively to the syntax above, it is acceptable to initialize the local variables in the declaration line with some global values, as illustrated below:

$$\texttt{Module[}\{arg_1 = a_1,\ arg_2 = a_2,\ \cdots,\ arg_n = a_n\},\quad expressions\texttt{]}$$

However, a local variable initialized in this way cannot be used to initialize another local variable inside the declaration list.

Consider the following example. The value of local variable **x** is set to 4 in the module, and the module returns the value of 6 (i.e., **x = 4 + 2**). Here, **x** is local to the module. However, in the next line, the value of the globally defined variable **x** remains valid; that is why the **Print** command for **x** yields 3.

```
x = 3; y = Module[ {x = 4}, x = x + 2] (* module yields the value of 6 *)
Print["x=",x]                           (* Print gives the value of 3   *)
```

A **Return** statement may be used to return the value of a single or a list of outputs (**output**) from anywhere within the **Module**. This feature is illustrated in the following example:

```
(* x0 and y0 on input are global; x and y are local variables. *)
fun[x0_, y0_]:=Module[{x = x0, y = y0}, (* x, y are initialized to x0, y0 *)
   x += y;              (* 1st expression: x = x + y       *)
   y += x;              (* 2nd expression: y = y + x       *)
   Return[{x, y}]       (* module returns two values to f  *)
  ]
```

Here, by initializing the local variables **x, y** with the input global variables **x0, y0**, the local variables are modified within the module. When the **Return** is reached, the computed values of **x, y** are returned as a list.

The module **Fun** can be used like a function, with a full argument list, as follows:

```
    fun[3, 4]         (* gives {7, 11} as a list                         *)
    fun[3, 4][[1]]    (* gives 7, isolates the 1st part of the list      *)
    fun[3, 4][[2]]    (* gives 11 isolates the 2nd part of the list      *)
```

Consider the following function example, which uses a **Module** to find the sum of natural numbers from 1 up to **n**.

```
   ISum[n_] := Module[{i, s = 0}, (* local variables defined and initialized *)
        Do[s += i, {i, 1, n}];    (* a Do loop sums i's from 1 to n          *)
        Return[s]                  (* Module returns s to ISum                *)
         ];
```

When using **ISum** in algebraic operations, as illustrated below, we the local variables (**i** and **s**) are confined to the **ISum**. That is why the **Print** statement returns the variable names, not values.

```
    ISum[10]                       (* For n=10, ISUM returns 55             *)
    Print[i, " ", n, " ", s]       (* i, n, and s are unknown outside ISum *)
```

However, in the following module usage, the variable **s** is removed from the list of local variables.

```
   ISum[n_] := Module[{i}, (* local variables defined *)
        s = 0;                     (* variable s is initialized *)
        Do[s += i, {i, 1, n}];    (* a Do loop sums i's from 1 to n *)
        Return[s]                  (* Module returns s to ISum       *)
         ];
```

This time the variable **s** (and its numerical value) is accessible outside of the **ISum** or **Module** definition, which is why the value of **s** (55) is printed.

```
    ISum[10]                       (* For n=10, ISUM returns 55      *)
    Print[i, " ", n, " ", s]       (* returns  i n 55                *)
```

In this way, local variables are set up to which values can be assigned and then changed. Often, however, all you need are local constants, to which you assign a value only once. The Mathematica With construct allows you to set up such local constants.

Bibliography

[1] GAYLORD, R. J., KAMIN, S. N., WELLIN, P. R., *An Introduction to Programming with Mathematica*(R). Springer New York, 2012.

[2] GLYNN, J., GRAY, T. *The Beginner's Guide to MATHEMATICA*$^{(R)}$, Version 4, Cambridge University Press, 2000.

[3] HASSANI, S., *Mathematical methods using Mathematica for students of physics and related fields*.Springer, New York, 2003.

[4] MAGRAB, E. B.. *An Engineer's Guide to Mathematica*. Wiley, 2014.

[5] MANGANO, S. *Mathematica Cookbook*. O'Reilly Media, 2010.

[6] RUSKEEPÄÄ, H. *Mathematica Navigator: Mathematics, Statistics, and Graphics*. Elsevier Science, 2004.

[7] TORRENCE, B. F., TORRENCE, E. A. *The Student's Introduction to Mathematica and the Wolfram Language*. Third Ed., Cambridge University Press, 2019.

[8] *Wolfram Mathematica Tutorial Collection: CORE LANGUAGE*.Wolfram Research, Incorporated, 2008.

[9] *Wolfram Mathematica Tutorial Collection: MATHEMATICS AND ALGORITHMS*.Wolfram Research, Incorporated, 2008.

[10] *Wolfram Mathematica Tutorial Collection: NOTEBOOKS AND DOCUMENTS*.Wolfram Research, Incorporated, 2008.

[11] reference.wolfram.com site