People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University of Sciences and Technology of Oran - Mohamed BOUDIAF -
Faculty of Mechanical Engineering
Department of Naval Architecture and Marine Engineering

Introduction to FORTRAN 77 Programming

Dr. ABED Bouabdellah

2025

Course Manual

Contents

Introduction

This educational document provides an introduction to computer programming using FORTRAN 77, one of the earliest high-level programming languages designed specifically for scientific and engineering computations. Despite its age, FORTRAN remains widely used in fields such as physics, computational chemistry, engineering simulations, and numerical weather prediction due to its exceptional efficiency in mathematical and array-based computations.

Why Learn FORTRAN 77?

While modern Fortran standards (e.g., Fortran 90/95/2003/2008) offer advanced features, FORTRAN 77 remains a valuable learning tool for several reasons:

- **Historical Importance**: Many legacy scientific codes, particularly in high-performance computing (HPC), are written in FORTRAN 77.
- **Numerical Efficiency**: Optimized for mathematical operations, it is still favored in computationally intensive tasks.
- **Conceptual Clarity**: Its simple syntax helps beginners focus on algorithmic thinking and structured programming.
- **Continued Relevance**: Many academic and industrial applications still rely on FORTRAN-based numerical libraries.

Course Objectives

By the end of this course, you will be able to:

- Write and debug basic FORTRAN 77 programs.
- Use control structures (loops, conditionals) to implement computational algorithms.
- Apply fundamental numerical methods (root-finding, integration, linear algebra).

- Work with arrays, functions, and subroutines for modular programming.
- Translate mathematical formulations into efficient FORTRAN code.

Course Outline

The material is structured to progressively develop programming skills with an emphasis on scientific applications. Key topics include:

- Basic Syntax and Program Structure
- Data Types, Variables, and Operators
- Input/Output (I/O) Handling
- Control Flow: IF Statements and DO Loops
- Procedural Programming: Functions and Subroutines
- Array Manipulation and Matrix Operations
- File Handling for Data Processing
- Implementing Numerical Algorithms

Each section includes worked examples, exercises, and programming assignments to reinforce learning through practical application.

How to Use This Document

To maximize learning:

- **Read actively**: Follow along with code examples by writing and testing them yourself.
- **Experiment**: Modify provided programs to observe how changes affect output.
- **Practice consistently**: Complete all exercises to solidify understanding.
- **Apply concepts**: Try implementing small numerical methods from your field of study.

Final Remarks

Programming is fundamentally about structured problem-solving. Learning FORTRAN 77 equips you with timeless skills in algorithmic thinking, code efficiency, and computational precision—whether you continue with modern Fortran or transition to Python, C++, or MATLAB. The principles you master here will serve as a strong foundation for future work in scientific computing.

Chapter 1

Fundamentals of Computer Science

1.1 Information Processing

Computer Science (from French ”informatique”, coined in 1962 by Philippe Dreyfus by combining ”information” and ”automatique”) is the scientific discipline concerned with the automatic processing of information using computers. This field fundamentally relies on three core components:

Algorithms

An **algorithm** is a finite sequence of clear, unambiguous, and ordered instructions that can be executed by a computer to solve a specific problem or perform a particular task. Algorithms define how data is processed to achieve desired outcomes.
In programming, algorithms are designed to manipulate data structures, perform calculations, make decisions, and control the flow of execution. They can be implemented in any programming language, including legacy ones such as Fortran 77, which was widely used for scientific computing due to its efficiency in handling numerical operations.

Data

Data refers to the collection of values, facts, or information that is processed by algorithms. Data can take many forms such as numbers, text, images, or complex structures like databases or arrays. The quality and structure of data significantly affect the performance and accuracy of algorithms.

Computers

Computers are physical devices capable of executing operations automatically based on given algorithms applied to provided data. They process encoded

instructions at high speed and deliver the desired results. Modern computers include not only personal devices but also servers, embedded systems, supercomputers, and cloud-based systems.

Problem-Solving Process

To solve a problem using a computer:

1. Identify the problem and the necessary **data**
2. Design an **algorithm** that specifies how to process the data
3. Encode both the algorithm and the data into a form understandable by the computer (programming)
4. The result is a **computer program**, written in a programming language

Fortran 77 Example

Here's a simple Fortran 77 program illustrating these concepts:

```
      PROGRAM SUM_TWO_NUMBERS
C     DATA: Declaration and initialization
      INTEGER A, B, RESULT

      A = 5
      B = 7

C     ALGORITHM: Addition operation
      RESULT = A + B

C     OUTPUT: Displaying the result
      WRITE(*,*) 'Result: ', RESULT

      END
```

Data: A = 5, B = 7

Algorithm: RESULT = A + B

Computer: Executes the program and outputs the result

Learning Objectives

- Understand the fundamental concepts of computer science
- Learn to design basic algorithms
- Gain familiarity with programming concepts

1.2 Computer Programs and Programming Languages

A **computer program** is a sequence of encoded instructions that strictly follow the syntax and semantic rules defined by a specific programming language. These instructions are written into a *source file* using a plain text editor. Unlike natural languages, programming languages are formal and require precise adherence to structure and syntax. A single typo or incorrect symbol can prevent the program from compiling or executing correctly.

Standardized Programming Languages

Most modern programming languages are **standardized** (e.g., machine language, assembly language, Pascal, Python, Fortran, C, etc.). This standardization provides an essential advantage: **program portability**, meaning a program can be executed on different systems with minimal or no modifications.
Once written, the program must be read and translated into **machine language** (binary code) so the computer can execute each instruction by mapping every symbolic command to a specific action.

Translation into Machine Language

The execution of an algorithm written in a high-level programming language requires a **translation phase** from this symbolic language into the internal machine language understood by the computer's processor.
There are two main strategies for this translation process:

1.3 Compilation

Compilation involves translating the entire source program into machine language all at once, using a special program called a **compiler**. The result is an **object program** (or executable), which is directly understandable by the computer.
- The object program is saved and can be executed independently of the original source code.
- Compilation usually results in faster execution since the translation is done once before running the program.

Example (Fortran 77):

```
    PROGRAM HELLO
    WRITE(*,*) 'Hello, World!'
    END
```

To compile and run:

```
gfortran hello.f -o hello
./hello
```

2. Interpretation

Interpretation translates and executes each line of the program **on the fly**, one at a time, using an **interpreter**.
- No separate object file is generated.
- Execution is slower than compiled code because translation occurs during runtime.
- However, it allows for immediate feedback and easier debugging, making it ideal for development and scripting.

Example (Python):

```
    print("Hello, World!")
```

To run:

```
    python hello.py
```

Summary Table

Feature	Compilation	Interpretation
Translation method	Entire program at once	Line-by-line during execution
Output	Object/Executable file	No object file
Execution speed	Faster	Slower
Debugging	More complex	Easier and more interactive
Examples	C, Fortran, Pascal	Python, JavaScript

Conclusion

Whether a program is compiled or interpreted depends on the language and the environment. Both methods serve the same ultimate goal: **executing algorithms efficiently on a computer.**

Chapter 2

Introduction to FORTRAN 77

2.1 Historical Background

The programming language **FORTRAN** (an acronym for ”FORmula TRANslator”) was designed by **John Backus** in **195**4, and the first version became available in 1955. With the release of the first commercial computers in 1956, FORTRAN was introduced to the scientific community as a powerful tool for numerical computation.

The first reference manual published by IBM defined the initial version of the language: **FORTRAN I**. Over time, successive versions were developed, each incorporating new features and improvements:

- **FORTRAN II (1957)**: Introduced support for subroutines, functions, and common blocks (`COMMON`), enabling better code organization and modularity.
- **FORTRAN III (1958)**: Added capabilities for machine-dependent features but was not widely adopted due to lack of portability.
- **FORTRAN IV (1962)**: Marked a significant step forward with:
- Explicit type declarations (`REAL`, `INTEGER`, etc.)
- The `DATA` statement for initializing variables.
- The `BLOCK DATA` statement for initializing common blocks.

One of the most influential and long-lasting versions was **FORTRAN 77**, officially approved by ANSI (American National Standards Institute) in 1977. This version gained worldwide acceptance and became known as the **universal FORTRAN**.

Key enhancements in FORTRAN 77 included:

- Structured control statements such as `IF-THEN-ELSE-ENDIF`, significantly reducing reliance on the `GOTO` statement.
- Support for character data types (`CHARACTER`), allowing manipulation of strings and textual information.

These improvements made FORTRAN 77 more readable, maintainable, and suitable for a wide range of scientific and engineering applications.

2.2 Definition and Semantic Elements of Fortran 77

Like any programming language, **Fortran 77** uses a set of basic elements called **words**, which represent different types of entities:
- **Constants**: Fixed values used in computations.
- **Variables**: Named storage locations that can hold different values during program execution.
- **Functions**: Predefined or user-defined operations that return a single value.
- **Subprograms (Subroutines)**: Modular blocks of code that perform specific tasks and can be called from other parts of the program.
- **Keywords**: Reserved words that have special meaning in the language (e.g., `IF`, `DO`, `READ`, etc.).

These elements are combined into **statements** (instructions) following strict syntactic rules defined by the language.
Each statement defines a specific operation to be performed on data. A complete Fortran program is simply a sequence of such statements.

2.2.1 Data Types in Fortran 77

In Fortran 77, data — including constants and variables — can be expressed using six fundamental data types:

1. **Integer mode** (`INTEGER`) Represents whole numbers without decimal points.
2. **Real mode** (`REAL`) Represents floating-point (decimal) numbers with single precision.
3. **Complex mode** (`COMPLEX`) Used for complex numbers in the form $a + bi$.
4. **Double precision mode** (`DOUBLE PRECISION`) Provides higher-precision real or complex numbers for scientific computation.
5. **Logical mode** (`LOGICAL`) Boolean values used for conditional expressions: `.TRUE.` or `.FALSE.`
6. **Character mode** (`CHARACTER`) Strings of text characters, allowing manipulation of textual data.

2.2.2 Execution Flow in Fortran Programs

In Fortran, the execution of a program is **sequential** by default. This means that instructions are executed in the order they appear in the source code, unless modified by control structures like loops or conditional statements. For example:

```
    PRINT *, 'Starting program'
    A = 5
    B = 10
    SUM = A + B
    PRINT *, 'Sum = ', SUM
```

This sequence will always execute from top to bottom unless explicitly redirected by an instruction such as GOTO, IF, or DO.

2.3 Module: Fortran Programming Syntax

2.3.1 The Fortran Character Set

The Fortran alphabet consists of three distinct character classes:

- **Digits**: 0 1 2 3 4 5 6 7 8 9
- **Letters**:
  - Lowercase: a b c ... x y z
  - Uppercase: A B C ... X Y Z
- **Special Characters**: - + / * , . ( ) ” ’ = (space)

2.3.2 Lexical Elements

(a) Words

Fortran recognizes two types of words:

- **Keywords**: Reserved words with specific meanings (e.g., DO, WRITE, IF, END)
- **Symbols**: Programmer-defined identifiers with constraints:
  - Maximum length: 6 characters (FORTRAN 77 standard)
  - First character must be a letter
  - Subsequent characters: letters or digits
  - Used for variables, functions, and subprograms

2.3.3 Instruction Structure

Fortran instructions follow strict formatting rules:

Columns 1-5	Column 6	Columns 7-72	Columns 73+
Label field	Continuation marker	Instruction field	Comments (ignored)

Key Features:

- **Label**:
  - Optional numeric identifier (1-99999)
  - Must start in column 1
- **Continuation**:
  - Any character (except '0' or space) in column 6 indicates continuation
  - Maximum 19 continuation lines (FORTRAN 77)
- **Instruction Field**:
  - Contains executable statements or declarations
  - Must begin in column 7

2.3.4 Program Structure

A complete FORTRAN 77 program has the following skeleton:

```
    PROGRAM program_name
    [specification statements]
    [executable statements]
    END
```

Example

```
    PROGRAM HELLO
    INTEGER I, J
    DO 100 I = 1, 10
    J = I**2
    WRITE(*,*) 'Square of', I, 'is', J
    100   CONTINUE
    END
```

2.3.5 Important Notes

- Fortran is **case-insensitive**
- Fixed-format requirements are relaxed in modern Fortran (90/95+)
- The `PROGRAM` statement is optional (default program name is used)
- The `END` statement must be the last line

Chapter 3

Constants and Variables

In computer programming, we frequently need to manipulate and temporarily store information during program execution. This data may originate from user input or be generated by the computer as intermediate or final results. Data can be of several types: numeric, alphanumeric, or boolean. For this purpose, we use constants and variables in computer programs.

3.1 Constants

Constants are data values that cannot be modified during algorithm execution or program runtime. A constant is essentially a named value that remains unchanged throughout program execution. Attempting to modify a constant's value will result in an error.
For example, we can use a constant named `PI` to store the value of π:

```
PI = 3.141592653589793
```

3.1.1 Integer Constants

Integer constants represent signed whole numbers. They must follow these rules:

- May begin with an optional sign (+ or -)
- Must contain only digits (0-9)
- Cannot contain other characters

Valid Examples	Invalid Examples
123	3 14 (contains space)
-18	3.14 (contains decimal point)
+4	2,71828 (contains comma)

3.1.2 Real Constants (Single Precision)

Real constants represent floating-point numbers in either:

- Standard decimal notation (must contain a decimal point), or
- Scientific notation (must contain ’E’)

For numbers between -1 and 1, the leading zero may be omitted.

Decimal Notation Examples:

```
0.      0.0     .0     1.     1.0
0.001   .001   -36.   -36.0   -36.00
3.1415
```

Scientific Notation Examples:

```
31415E-4     31415E-04    314.15E-02
1.E12        1.0E12       1.0E012
1.0E+12      1.0E+012     5.3E-8
5.30E-8      5.3E-08      -5.30E-08
-5.30E-008
```

3.2 Double Precision Constants

Double precision constants use ’D’ instead of ’E’ in scientific notation and provide approximately double the precision of real constants (typically 64 bits).

```
0.D0              31415D-4          31415D-04
314.15D-02        1.D12             1.0D12
1.0D012           1.0D+12           1.0D+012
5.3D-8            5.30D-8           5.30D-08
-5.30D-08         -5.3D-008
```

3.3 Complex Constants

Complex constants combine two real or double precision numbers in parentheses, representing the complex number a + ib as (a, b).

Examples:

```
(0., 0.)              (1., -1.)            (2.5, 1.)
(1.34E-7, 4.89E-8)    (1.34D-7, 4.89D-8)
```

3.4 Logical Constants

Logical constants have only two possible values:

.TRUE. and .FALSE.

3.5 String Constants

String constants are sequences of characters enclosed in apostrophes. To include an apostrophe within a string, double it:

```
'This is a string'
'/home/user/data'
'Don''t forget to double apostrophes'
```

3.6 Constant Declaration

Constants are declared using the PARAMETER attribute. They cannot be modified during program execution.

Syntax:

```
PARAMETER (name1 = value1, name2 = value2, ...)
```

Examples:

```
PARAMETER (MAX_ITER = 1000, PI = 3.141592653589793D0)
PARAMETER (E_CHARGE = 1.602176634D-19, TOL = 1.0E-6)
```

3.7 Variables

Variables are named memory locations that can be read and modified during program execution. They are essential for:

- Symbol manipulation
- Formula implementation

Before use, variables must be:

- Typed (explicitly or implicitly)
- Named according to language rules

3.7.1 Variable Declaration

Fortran provides several intrinsic data types:

Declaration	Type	Description
REAL	Single precision	32-bit floating point
DOUBLE PRECISION	Double precision	64-bit floating point
INTEGER	Integer	Whole numbers
LOGICAL	Boolean	.TRUE. or .FALSE.
COMPLEX	Complex	Real-imaginary pairs
CHARACTER	String	Text data

Examples:

```
    INTEGER :: i, j, k, counter
    REAL :: x_coord, y_coord, temperature
    DOUBLE PRECISION :: exact_value, tolerance
    COMPLEX :: impedance, wave_function
    CHARACTER(LEN=20) :: student_name, course_code
    LOGICAL :: converged, debug_mode
```

3.8 Variable Types and Ranges

3.8.1 Integer Variables

Integer variables can be declared with different sizes:

Type	Bytes	Range
INTEGER(1)	1	-128 to 127
INTEGER(2)	2	$-32,768$ to $32,767$
INTEGER(4)	4	$-2,147,483,648$ to $2,147,483,647$

Syntax:

```
    INTEGER(kind) :: var1, var2, ...
```

Examples:

```
INTEGER       counter
INTEGER(2)    short_value
INTEGER(4)    large_number
```

3.8.2 Real Variables

Real variables store floating-point numbers with different precision levels:

Type	Bytes	Precision	Range
REAL(4)	4	6-7 digits	$\pm 1.18 \times 10^{-38}$ to $\pm 3.40 \times 10^{38}$
REAL(8)	8	15-16 digits	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$

Syntax:

```
REAL  var1, var2, ...
DOUBLE PRECISION   var1, var2, ...
```

3.8.3 Complex Variables

Complex numbers store real and imaginary components:

Type	Description
COMPLEX(4)	Single precision (2×REAL(4))
COMPLEX(8)	Double precision (2×REAL(8))

Syntax:

```
COMPLEX    var1, var2, ...
```

3.8.4 Character Variables

Strings are declared with fixed lengths:

Syntax:

```
CHARACTER(len)   var1, var2, ...
CHARACTER(len=20)   string_var
```

3.8.5 Logical Variables

Boolean variables store truth values:

Syntax:

```
    LOGICAL   var1, var2, ...
```

3.9 Type Declaration Best Practices

- Always use `IMPLICIT NONE` to require explicit declarations
- Use meaningful, descriptive variable names
- Initialize variables when declaring them
- Use parameters for constant values
- Choose appropriate precision for numerical values
- Prefer modern Fortran syntax (e.g., `REAL(8)` over `REAL*8`)

Chapter 4

Operators and Expressions

III.1 Assignment Statement

The assignment statement stores the result of an expression in a variable.

General syntax:

```
variable = expression
```

Characteristics:

- The expression and variable must be of compatible types
- Assignment uses the = operator
- Evaluation occurs at compile time
- Operation precedence can be modified with parentheses

Examples:

```
    pi = 3.1415926535          ! Real constant assignment
    counter = 0                ! Integer constant assignment
    active = .TRUE.            ! Logical assignment
    message = "Hello"          ! String assignment
    result = (a + b)*c         ! Expression assignment
```

[Additional note: Fortran uses single equals (=) for assignment, unlike some languages that use it for equality comparison]

III.2 Expression Composition

An expression may consist of:

Element	Example
Constant	3.14
Variable	x
Function	SQRT(x)
Operation	a + b
Combination	a**2 + SQRT(b)

[Note: Fortran expressions follow standard mathematical conventions but have some unique operators]

III.3 Variable Naming Rules

- **Composition**:
  - Must begin with a letter (a-z, A-Z)
  - May contain digits (0-9)
  - Maximum length: 6 characters (FORTRAN 77 standard)
- **Prohibited**:
  - Accented characters
  - Spaces
  - Special symbols ($, @, etc.)
  - Reserved keywords (PROGRAM, END, etc.)
- **Case insensitivity**: MyVar myvar MYVAR

[Historical context: Modern Fortran versions allow longer names, but these rules reflect classic FORTRAN 77 constraints]

III.4 Fortran Operators

III.4.1 Arithmetic Operators

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
**	Exponentiation	a**b

III.4.2 Relational Operators

Operator	Description	Example
.EQ. or ==	Equal	a .EQ. b
.NE. or /=	Not equal	a .NE. b
.GT. or >	Greater than	a .GT. b
.GE. or >=	Greater or equal	a .GE. b
.LT. or <	Less than	a .LT. b
.LE. or <=	Less or equal	a .LE. b

[Note: Modern Fortran prefers ==, /= etc., but the dotted forms (.EQ., .NE.) remain valid]

III.4.3 Logical Operators

Operator	Description	Example
.NOT.	Negation	.NOT. x
.AND.	Logical AND	a .AND. b
.OR.	Logical OR	a .OR. b
.EQV.	Equivalence	a .EQV. b
.NEQV.	Non-equivalence	a .NEQV. b

[Explanation: .EQV. tests if both operands have the same truth value, while .NEQV. is the XOR operation]

III.5 Operator Precedence

Evaluation order (from highest to lowest precedence):

1. Parentheses ()
2. Functions (SIN, COS, etc.)
3. Exponentiation **
4. Multiplication/Division * /
5. Addition/Subtraction + -
6. Relational operators .EQ. etc.
7. .NOT.
8. .AND.
9. .OR.

10. .EQV., .NEQV.

[Important: This precedence differs from many modern languages, especially regarding logical operators]

Complete Example

```
    PROGRAM EXAMPLE
    IMPLICIT NONE
    REAL :: x, y, result
    LOGICAL :: condition

    x = 5.0
    y = 2.0

    ! Complex arithmetic expression
    result = (x**2 + y**2)/2.0

    ! Logical expression
    condition = (x > 0.0) .AND. (y < 10.0)

    PRINT *, 'Result:', result
    PRINT *, 'Condition:', condition
    END PROGRAM EXAMPLE
```

[Note: The IMPLICIT NONE statement forces explicit variable declaration, considered good practice in modern Fortran]

4.1 Arithmetic Operators and Mathematical Functions

4.1.1 Arithmetic Operators (2019)

Numerical expressions in Fortran are composed of operands and operators, combined according to Fortran syntax rules. A numerical expression is one whose operands are of one of the three numerical types: integer, real, and complex (either single or double precision).
The Fortran arithmetic operators are as follows:

- + for addition
- - for subtraction
- * for multiplication
- / for division
- ** for exponentiation (I**2 means I squared, X**Y with X and Y real means exp(Y*log X) when log X is defined)

Operands can be:

- Constants
- Simple or indexed variables
- Functions
- Complex expressions

Examples (2019):

```
    3.14159
    K
    A(I)
    SIN(A + B)
    -1.0 / X + Y / Z ** 2
```

4.1.2 Order of Evaluation in Arithmetic Expressions

The evaluation of an arithmetic expression follows this established order:

1. Sub-expressions in parentheses (starting with the innermost)
2. Standard numerical functions

3. Exponentiation (raising to a power)
4. Multiplication and division (evaluated from right to left)
5. Addition, subtraction, or negation

Examples of execution order:

1. `A+B*C**2`
   (a) `C**2`
   (b) `B*C**2`
   (c) `A+B*C**2`
2. `A/(B*C)`
   (a) `(B*C)`
   (b) `A/(B*C)`

4.1.3 Type Conversion in Results from +, -, *, /

Types are ranked in increasing order as follows:

1. `INTEGER*2` (lowest rank)
2. `INTEGER*4` (`INTEGER`)
3. `REAL*4` (`REAL`)
4. `REAL*8` (`DOUBLE PRECISION`)
5. `COMPLEX*8` (`COMPLEX`)
6. `COMPLEX*16` (highest rank)

The operand with the highest rank determines the result type. During computation, if operands have different types, the operand with the lower-ranked type (using less memory) will be converted to the type of the higher-ranked operand.

Examples:

1. For addition, subtraction, and multiplication operations with integer operands, the result will be an integer:

$$3 + 4 = 7$$
$$4 \times 3 = 12$$
$$3 - 4 = -1$$

2. For division, the result will be the integer part of the quotient:

$$4/3 = 1$$
$$3/4 = 0$$
$$-9/2 = -4$$

3. For mixed-mode operations (note the importance of typing):

$$(9/2) + 6.2 = 4 + 6.2 = 10.2$$
$$(9./2) + 6.2 = 4.5 + 6.2 = 10.7$$

4.1.4 Additional Examples

Example 4.1.1 *Calculate the expression 2**3 + 5*4 - 6/2:*

1. *Exponentiation first: 2**3 = 8*
2. *Then multiplication: 5*4 = 20*
3. *Then division: 6/2 = 3*
4. *Finally additions and subtractions: 8 + 20 - 3 = 25*

Example 4.1.2 *Mixed-type operations:*

- *5 + 2.5 results in 7.5 (integer promoted to real)*
- *10.0/4 results in 2.5 (integer promoted to real)*
- *(1.0, 2.0) * 3 results in complex number (3.0, 6.0)*

4.2 Intrinsic Arithmetic Functions

4.2.1 Trigonometric Functions

Fortran provides the following intrinsic trigonometric functions:

- ASIN(X): Arc sine (inverse sine)
- ACOS(X): Arc cosine (inverse cosine)
- ATAN(X): Arc tangent (inverse tangent)
- SIN(X): Sine (angle in radians)
- COS(X): Cosine (angle in radians)
- TAN(X): Tangent (angle in radians)
- SINH(X): Hyperbolic sine
- COSH(X): Hyperbolic cosine
- TANH(X): Hyperbolic tangent

Examples:

```
PI = 4.0*ATAN(1.0)   ! Calculate
X = SIN(PI/2)        ! X = 1.0
Y = ACOS(0.5)        ! Y =  /3   1.0472
Z = COSH(1.0)        ! Z   1.54308
```

4.2.2 Other Mathematical Functions

Additional intrinsic functions include:

- ABS(X): Absolute value
- SQRT(X): Square root
- LOG(X): Natural logarithm (base e)
- LOG10(X): Common logarithm (base 10)
- EXP(X): Exponential function (e^X)

Examples:

```
A = ABS(-5.2)        ! A = 5.2
B = SQRT(16.0)       ! B = 4.0
C = LOG(10.0)        ! C   2.30259
D = EXP(1.0)         ! D   2.71828
```

4.2.3 Minimum and Maximum Functions (2019)

- `MIN(X1,X2)`: Minimum of two real numbers
- `MAX(X1,X2)`: Maximum of two real numbers

These functions can also handle more than two arguments:

```
X = MIN(5.0, 3.0, 8.0)  ! X = 3.0
Y = MAX(2.0, 9.0, 4.0)  ! Y = 9.0
```

4.3 Logical Expressions

4.3.1 Definition

A logical expression allows comparison between two numerical expressions (or character strings). An elementary logical expression consists of two arithmetic quantities connected by a logical relational operator.

Arithmetic constant
Arithmetic variable
Logical relational operator
Arithmetic variable
Arithmetic expression

The result of a logical comparison is of logical type, having one of the values:

- `.TRUE.` (true)
- `.FALSE.` (false)

4.3.2 Relational Operators

The relational operators in Fortran are:

- `.LT.` or `<` (less than)
- `.LE.` or `<=` (less than or equal to)

- .GT. or > (greater than)
- .GE. or >= (greater than or equal to)
- .EQ. or == (equal to)
- .NE. or /= (not equal to)

Examples:

```
LOGICAL   L1, L2, L3
L1 = (5 < 3)         ! L1 = .FALSE.
L2 = (4.0 == 4)      ! L2 = .TRUE.
L3 = ('A' /= 'B')    ! L3 = .TRUE.

IF (X >= 0.0) THEN
Y = SQRT(X)
END IF
```

4.3.3 Compound Logical Expressions

Logical expressions can be combined using:

- .AND.: Logical AND
- .OR.: Logical OR
- .NOT.: Logical NOT

Examples:

```
LOGICAL :: A, B, C
A = (X > 0.0) .AND. (X < 10.0)   ! True if 0 < X < 10
B = (Y < 0.0) .OR. (Y > 100.0) ! True if Y outside [0,100]
C = .NOT. (Z == 0.0)             ! True if Z   0
```

4.3.4 Operator Precedence in Logical Expressions

The evaluation order for logical expressions is:

1. Arithmetic expressions
2. Relational operators

3. .NOT.
4. .AND.
5. .OR.

Example:

```
LOGICAL  RESULT
RESULT = X > 5.0 .AND. .NOT. Y < 0.0 .OR. Z == 1.0
! Equivalent to: ((X > 5.0) .AND. (.NOT. (Y < 0.0))) .OR. (Z == 1.0)
```

Chapter 5

Input and Output Instructions

5.1 Generalities

The purpose of I/O (Input/Output) is to enable communication between a program and its external environment.
- **Input** occurs when the program reads data from an external device (keyboard, disks, etc.). - **Output** occurs when the program writes data to an external device (screen, disks, etc.).

5.2 Input/Output (I/O)

I/O involves specifying: 1. The type of data to be processed. 2. How the data will be processed. 3. The logical file to be used.
There are two types of I/O instructions: - **Free-format I/O**. - **Formatted I/O**. Four primary instructions are used for I/O operations: - `PRINT` or `WRITE`: Displays (or writes) information to a device or file. - `READ`: Reads data from an external device. - `FORMAT`: Describes how information is encoded/decoded.

5.2.1 The `PRINT` Instruction

Used to display results on the screen (monitor).

Syntax

a) **Free-format**:

`PRINT*, <listVar>`

`<listVar>`: A comma-separated list of variables.
Example:

```
    PRINT*, 'The result is: ', I, J, f
```

b) **Formatted:**

PRINT <fmt>, <listVar>

<fmt>: A format specifier (can be a FORMAT label or a valid format string).

Example:

```
    PROGRAM CH4
    PRINT '(A6)', 'ABED'
    PRINT 100, 'BOUABDELLAH'
    100 FORMAT(A12)
    STOP
    END
```

5.2.2 The READ Instruction

Used to read data from a specified logical file.

Syntax

READ (<unit>, <fmt>) <listVar>

- <unit>: An integer indicating the logical file number. If unit = *, input is from the keyboard; otherwise, it is from a data file. - <fmt>: A format specifier (can be a FORMAT label or a valid format string).

a) **Free-format:**

READ (<unit>, *) <listVar>

Example:

```
    REAL x, y
    INTEGER i, j
    READ (*, *) x, y, i, j
```

b) **Formatted:**

READ (<unit>, <fmt>) <listVar>

Example:

```
    INTEGER i, j
    READ (*, 100) i, j
    100 FORMAT (215)
```

5.2.3 The WRITE Instruction

Used to write data to a specified logical file.

Syntax

```
WRITE (<unit>, <fmt>) <listVar>
```

a) **Free-format:**

```
WRITE (*, *) <listVar>
```

Example:

```
1    WRITE (*, *) I, J, K
2    WRITE (*, *) 'Final values of I, J, K: ', I, J, K
```

b) **Formatted:**

```
WRITE (<unit>, <fmt>) <listVar>
```

Example:

```
1    PROGRAM CH41
2    M = 123456
3    WRITE (*, '(I8)') M
4    END
```

5.2.4 Handling Indexed Variables

For arrays (vectors/matrices), use implicit loops for I/O operations.
Example 1: Reading/Writing a vector column-wise:

```
READ (<unit>, <fmt>) (V(k), k=1, 3)
WRITE (<unit>, <fmt>) (V(k), k=1, 3)
```

Equivalent to:

```
READ (<unit>, <fmt>) V(1), V(2), V(3)
WRITE (<unit>, <fmt>) V(1), V(2), V(3)
```

Example 2: Reading/Writing a matrix row-wise:

```
READ (<unit>, <fmt>) ((V(i,j), j=1,3), i=1,3)
WRITE (<unit>, <fmt>) ((V(i,j), j=1,3), i=1,3)
```

5.3 The FORMAT Instruction

A non-executable labeled instruction that defines how data is encoded/decoded during I/O operations.

Syntax

<label> FORMAT (<specifiers>)

- <label>: Links the FORMAT to an I/O instruction. - <specifiers>: Defines the layout, alignment, and data type for I/O.

Example:

```
    OPEN (11, FILE='DATA.INPUT')
    OPEN (12, FILE='DATA.OUTPUT')
    PI = 3.14159
    READ (11, *) I
    READ (11, 102) I
    WRITE (*, 100) PI
    WRITE (12, 100) PI
    102 FORMAT (I3)
    100 FORMAT ('THE VALUE OF PI IS: ', G13.6)
    STOP
    END
```

5.3.1 Format Specifiers

In the following, w, m, d, and e are unsigned integers representing:

w	Total field width (must be > 0), including sign and decimal point.
d	Number of digits after the decimal point (can be zero).
n	Repetition factor for the specifier.

5.3.2 Data Editing Specifiers

The I Specifier

For integer variables (no decimal points/exponents).

Syntax:

[n]Iw[.d]

- Writes/reads n integers, each in a field of w characters (right-justified).
- d: Minimum number of digits (padded with zeros if needed).

Note: If the number exceeds w, asterisks (****) are printed.

The F Specifier

Used for real numbers in fixed-point notation.

Syntax

[n]Fw.d

- w: Total field width (must accommodate sign, decimal point, and digits).
- d: Number of digits after the decimal point.

Example: Printing the number -123.4567:

Format	**Output**
F9.4	-123.4567
F11.4	__-123.4567
F8.4	******* (overflow)
F13.6	__-123.456700
F6.0	__-123.

The E Specifier

Used for real numbers in floating-point (exponential) notation.

Syntax

[n]Ew.d

- w must satisfy: $w \geq d + 7$ (to accommodate sign, exponent, etc.).

Example: Printing -123.4567:

Format	**Output**
E9.4	******* (overflow)
E11.4	_-.1235E+03
E8.4	*******
E13.6	_-.123457E+03
E6.0	0.E+03

The D Specifier

For double-precision numbers (identical to E in input; uses D instead of E in output).

Syntax

```
[n]Dw.d
```

Example: Printing -123.4567:

Format	Output
D9.4	*******
D11.4	_-.1235D+03
D8.4	*******
D13.6	_-.123457D+03
D6.0	0.D+03

The G Specifier

Automatically switches between F and E notation based on the magnitude of the number.

Syntax

```
[n]Gw.d
```

Example:

```
    DOUBLE PRECISION PI
    PI = 3.14159
    R = -100.6
    D = 123.456789E-03
    H = 2.99792458D+08
    WRITE(*,100) PI, R, D, H
    100 FORMAT(' ',4G10.3)
```

Output:

```
    3.14
    -101
    0.123
    0.300E+09
```

The L Specifier

For logical variables (.TRUE. or .FALSE.).

Syntax

[n]Lw

Example:

```
    LOGICAL X, Y
    X = .FALSE.
    Y = .TRUE.
    WRITE(*,100) X
    WRITE(*,200) Y
    100 FORMAT('X =', L2)
    200 FORMAT('Y =', L5)
```

Output:

```
    X = F
    Y = ____T
```

5.3.3 Layout and Editing Specifiers

The A Specifier

For character strings.

Syntax

[n]Aw

Example:

```
    CHARACTER x*4
    x = 'math'
    WRITE(*,10) x
    WRITE(*,11) x
    WRITE(*,12) x
    10 FORMAT(A2,'!')
    11 FORMAT(A5,'!')
    12 FORMAT(A,'!')
```

Output:

```
    ma!
    math !
    math!
```

The X Specifier

Inserts blank spaces.

Syntax

nX

Example:

```
WRITE(*,10) 'math'
10 FORMAT(2X,A,2X,'!')
```

Output:

```
__math__!
```

5.3.4 The / Specifier

Advances to the next record (line).

Syntax

/

Example:

```
WRITE(*,10) 'ABED', 'Bouabdellah'
10 FORMAT(A,/,A)
```

Output:

```
ABED
Bouabdellah
```

Character String Specifier

Prints literal strings enclosed in quotes.

Syntax

```
'text'
```

Example:

```
WRITE(*,10) 'ABED'
10 FORMAT('Nom: ', A)
```

Output:

```
Nom: ABED
```

Chapter 6

Subprograms

6.1 General Concepts

6.1.1 Purpose of Subprograms

During algorithm development, we often encounter sequences that perform identical functions, with only the data changing between uses.
While we could repeat and adapt these sequences as needed, it is more efficient to create the sequence once in a separate module that can be executed whenever required. This is the subprogram (S/P).
Additionally, software can become very large with numerous instructions, significantly reducing clarity. In such cases, we use S/Ps not just to avoid sequence repetition, but to break down the program into a set of shorter S/Ps that are easier to design, read, and debug.
Thus, a S/P appears as a separate module that takes input data, performs a processing sequence (matrix inversion, function integration, etc.), and produces results.

6.1.2 Communication Between Subprograms and Their Environment

FORTRAN 77 enables subprograms to communicate with their external environments through an associated parameter list. This parameter list constitutes the ”window” through which the subprogram receives its data or returns its results. The precise description of this list is essential for proper use of the subprogram. We distinguish the following four classes of parameters:

1. **Input parameters**: These contain the subprogram’s data and are not modified by it.
2. **Input-output parameters**: These contain data before the subprogram call and results afterward. Care must be taken with these parameters as the data they contain is destroyed by the subprogram.

3. **Output parameters**: These contain the results produced by the subprogram.
4. **Work parameters**: These are parameters without specific meaning for either input or output, which the subprogram uses for its internal operations.

The purpose of a S/P is therefore either to avoid unnecessary repetition or to fragment programs to make them ”mentally accessible” to those who design them. It is a very powerful concept that allows, with a single instruction, the execution of predefined sequences and thus the creation of a veritable ”macro-language.”

6.2 Different Types of Subprograms in FORTRAN 77

FORTRAN 77 recognizes two types of S/Ps:

1. **Internal S/Ps**: Defined and usable only within a single module (program or S/P).
2. **External S/Ps**: Defined, compiled, and usable in any other module (program or S/P). They can potentially be stored in S/P libraries (as with standard functions: SIN, COS, etc.).

6.2.1 Internal Subprograms

These are functions of one or more variables whose definition can be given in a single FORTRAN 77 statement, placed before any executable instruction. They essentially represent assignments to express formulas that are used repeatedly.

Syntax:

```
FUNCTION_Name (<Parameter list>) = <Expression>
```

Where:

- `FUNCTION_Name`: Alphanumeric string of up to 6 characters, the first being a letter.
- `Parameter list`: Var.1, Var.2, ...
- `Expression`: Arithmetic expression of integer, real, or string operation type, etc., depending on the function type.

84-86

6.2.2 Function Type Declaration

The function type is defined, as with variables, either by implicit convention or by explicit declaration.

Examples:

```
    C Real-type function
    ER (A, B) = ABS ((A - B)/A)
    C Integer-type function
    MD (I, J) = (I + J)*N**J
    C Real-type function
    DELTA (A, B, C) = B**2. - 4.*A*C
    C Complex-type function
    COMPLEX CER, A, B
    CER (A, B) = A*B + (0., 1.)
    C Character-type function
    CHARACTER CAR*20, C*10
    CAR (C, I) = C(1 : I)//(CHAR (I+1))
```

These functions are used like standard functions by naming them in an expression of the corresponding mode.

Example Usage:

```
    X = -ALOG10(ER(X, Y)) + 7.
    INDICE = (MD(I, IB) + K)*(MD(I,J))
```

6.2.3 External Subprograms

Function Subprograms

These are functions of n variables. The difference with internal S/Ps is that they can be used by any other module and can be defined by any number of instructions.

Syntax:

A function subprogram is the sequence between the keywords **FUNCTION** and **END.**

```
    [<Type>] FUNCTION Name (<Parameter list>)

    [Declaration statements]

    [Executable statements]

    RETURN

    END
```

- The construction of `Name` and `Parameter list` follows the same rules as for internal functions. The function type is defined by implicit convention or explicit declaration.
- Return to the calling module is achieved using the **RETURN** instruction.

87-88

6.2.4 Function Type Declaration

The function type is defined, as with variables, either by implicit convention or by explicit declaration.

Examples:

```
    C Real-type function
    ER (A, B) = ABS ((A - B)/A)
    C Integer-type function
    MD (I, J) = (I + J)*N**J
    C Real-type function
    DELTA (A, B, C) = B**2. - 4.*A*C
    C Complex-type function
    COMPLEX CER, A, B
    CER (A, B) = A*B + (0., 1.)
    C Character-type function
    CHARACTER CAR*20, C*10
    CAR (C, I) = C(1 : I)//(CHAR (I+1))
```

These functions are used like standard functions by naming them in an expression of the corresponding mode.

Example Usage:

```
    X = -ALOG10(ER(X, Y)) + 7.
    INDICE = (MD(I, IB) + K)*(MD(I,J))
```

External Subprograms

Function Subprograms

These are functions of n variables. The difference with internal S/Ps is that they can be used by any other module and can be defined by any number of instructions.

Syntax:

A function subprogram is the sequence between the keywords **FUNCTION** and **END**.

```
    [<Type>] FUNCTION Name (<Parameter list>)

    [Declaration statements]

    [Executable statements]

    RETURN

    END
```

- The construction of `Name` and `Parameter list` follows the same rules as for internal functions. The function type is defined by implicit convention or explicit declaration.
- Return to the calling module is achieved using the **RETURN** instruction.
- The return of the single output value is done through the function name. This name must therefore be assigned at least once in the module (for example, appear on the left side of an equals sign).
- If the function is of character type, the number of characters must be specified.

Example:

Consider the following flowchart defining a function $f(x, y)$:

Algorithm 1 Function F(X,Y)

1: **Name:** F
2: **Parameters:** x, y
3: **if** $x > y$ **then**
4: $F = x^2 - y^2$
5: **else**
6: $F = x^2 + y^2$
7: **end if**
8: **Return**

The corresponding FORTRAN 77 implementation would be:

```
      FUNCTION F (X, Y)
      C Input parameters:
      C X: abscissa of the considered point (real variable)
      C Y: ordinate of the considered point (real variable)
      C
      C Output parameter:
      C F: function value
      IF (X .GT. Y) THEN
      F = (X - Y)/SQRT(X**2 + Y**2)
      ELSE
      F = X**2/ABS(X + Y)**1.5
      ENDIF
      RETURN
      END
```

This subprogram can be used like any other function:

```
      A = F(X1, X2)
      X = F(X, Y)
      U = F(X + 1., Y)**T + F(X, Y)**2
```

6.2.5 General Subprograms (Subroutines)

These are applications of n input variables to p output variables. Like functions, they can be used by any other module and can be defined with any number of instructions.
A general subprogram is the sequence between the keywords **SUBROUTINE** and **END**.

Syntax:

```
SUBROUTINE Name (<parameter list>)

[Declaration statements]

[Executable statements]

[RETURN]

END
```

Key characteristics:

- Unlike functions, subroutines can return multiple values through their parameter list
- The subroutine name doesn't return a value (no type declaration needed)
- Parameters must be properly declared with their types
- The RETURN statement is optional (END implies return)

6.2.6 General Subprograms (Subroutines) (Continued)

The symbolic name and parameter list construction follows the same rules as for functions. Return to the calling module is achieved using the **RETURN** instruction.

Example:

Consider the following algorithm defining subroutine SP1:

Algorithm 2 Subroutine SP1

Name: SP1
Parameters: x, y
$T = 0.0$
for $i = 1$ to n **do**
  $T = T + i \times x$
end for
$P = x + I$
$Q = T/P$
Return

The corresponding FORTRAN 77 implementation would be:

```
      SUBROUTINE SP1 (X, N, Q, P)
      C This subroutine calculates P and Q...
      C
      C Input parameters:
      C X: abscissa of the considered point (real variable)
      C N: order of the expansion to compute (integer variable)
      C
      C Input-output parameter:
      C Q: input - value of...
      C    output - result of...
      C
      C Output parameter:
      C P: value of...
      T = 0.0
      DO I = 1, N
      T = T + X * I
      END DO

      P = X + 1.
      Q = P / T

      RETURN
      END
```

Usage:

This subprogram is called using the **CALL** statement:

```
1    CALL SP1 (X, NOMB, U, V)
2    CALL SP1 (A, N, P(1), P(2))
3    CALL SP1 (A + B, K, U, V)
```

Important Remarks:

- Input parameters can be variables or expressions
- Output or input-output parameters can only be variables
- There must be strict correspondence between:
  - The formal parameter list in the subroutine definition
  - The actual parameter list in the subroutine call
- Correspondence must be maintained in:
  - Number of parameters
  - Position in the parameter list
  - Parameter mode (input, output, input-output)

6.2.7 Local Variables

All variables that do not appear in the parameter list are called local variables (for example: I, T in previous examples). They only exist and are defined during the subprogram call. In particular, they do not retain their values between calls.

VI.4. Passing Array Parameters

Frequently, it is necessary to include arrays in the parameter list. In this case, these variables must be declared as arrays using ”DIMENSION” statements.
FORTRAN 77 offers two possibilities for this declaration:

- Fixed dimensioning
- Adjustable dimensioning

Fixed Dimensioning

Identical to dimensioning in a main program, the minimum and maximum index values are constants. For example, in the case of a subprogram calculating the maximum modulus of elements in a vector:

```
      FUNCTION VMAX(V, N)
      C Calculates the maximum absolute value in vector V of length N
      DIMENSION V(100)
      REAL VMAX
      VMAX = 0.0
      DO 10 I = 1, N
      IF (ABS(V(I)) .GT. VMAX) VMAX = ABS(V(I))
   10    CONTINUE
      RETURN
      END
```

Key characteristics:

- The array size is fixed at 100 elements in the subprogram
- Only the first N elements are actually used (N ≤ 100)
- The calling program must ensure the array doesn't exceed this size
- Simple but inflexible - requires knowing maximum size in advance

Adjustable Dimensioning

Adjustable dimensioning replaces constants in array declarations with variables. These variables must be present in the parameter list. The function from the previous section can be rewritten as follows:

```
      FUNCTION VNORM (VECT, N)
      C
      C Input parameters:
      C    VECT : vector to compute norm
      C    N    : vector dimension
      C
      C Output parameter:
      C    VNORM : norm of VECT
      C
```

```
10      DIMENSION VECT(N)
11      VNORM = ABS(VECT(1))
12      DO I = 2, N
13      VNORM = AMAX1(VNORM, ABS(VECT(I)))
14      END DO
15      RETURN
16      END
```

6.2.8 Advantages of Adjustable Dimensioning

- **Generality**:
  - The subprogram works for vectors of any dimension N
  - No need to modify code for different array sizes
- **Memory efficiency**:
  - No wasted memory reservation
  - Matches exactly the size needed
- **Maintainability**:
  - Single version handles all cases
  - No coordination needed between calling and called programs

Important Notes

- The dimension variable (N) must be passed as a parameter
- The actual array size must match or exceed the dimension specified
- Adjustable dimensions can only be used in subprograms
- The upper bound must be a variable, not an expression

6.2.9 Common Blocks

FORTRAN 77 provides the ability to pass information to subprograms without explicit parameters through **common blocks**. Variables are placed in common blocks using the COMMON statement.

Example:

```
    CHARACTER*1 CAR, LIGNE*80, BUFF*100
    COMMON /ZONE1/ A, B, I
    COMMON /ZONE2/ C, J, K
    COMMON /CAR1/ CAR, LIGNE, BUFF
```

Key characteristics:

- Variables A, B, I are in common block ZONE1
- Variables C, J, K are in common block ZONE2
- Character variables must be grouped separately (CAR1)
- Common blocks must be declared identically in all routines using them

Example Usage:

```
    FUNCTION VNORM(N)
    C Input parameter:
    C    N: vector dimension
    C Output parameter:
    C    VNORM: vector norm
    DIMENSION VECT(10)
    COMMON /ZONE1/ VECT
    VNORM = ABS(VECT(1))
    DO I = 2, N
    VNORM = AMAX1(VNORM, ABS(VECT(I)))
    END DO
    RETURN
    END

    PROGRAM ESSAI
    DIMENSION VECT(10)
    COMMON /ZONE1/ VECT
    N = 7
    X = VNORM(N)
```

```
20      END
```

6.2.10 Advantages and Disadvantages

Advantage:

- Reduces parameter list length for routines with many parameters

Disadvantages:

- Loss of generality:
  - Cannot use adjustable dimensioning
  - Requires fixed common block structure
  - Cannot use different variable names
- Reduced readability:
  - Modified variables not visible in parameter list
  - Harder to understand data flow

Due to these drawbacks, common blocks are rarely used in modern FORTRAN programming.

6.2.11 Passing Subprogram Names as Parameters

FORTRAN 77 allows passing subprogram names as parameters, enabling powerful generic programming capabilities.

Example: Root-Finding Subroutine

```
1     SUBROUTINE ZERO(F, A, B, MAXIT, X, IER)
2     C Input parameters:
3     C    F: Name of function whose root is sought
4     C    A: Lower bound of interval containing root
5     C    B: Upper bound of interval
6     C    MAXIT: Maximum allowed iterations
7     C Output parameters:
8     C    X: Found root value
9     C    IER: Exit indicator (0=success, 1=too many iterations)
10    XA = A
```

```
    XB = B
    T = (F(A) - F(B))/2.
    ...
    RETURN
    END
```

Usage:

```
    EXTERNAL INVOL, FONC
    CALL ZERO(INVOL, XI, X2, MAX, SOLU, IER)
    CALL ZERO(FONC, X, Y, MAX, SOLU, IER)

    INTRINSIC SIN, TAN
    CALL ZERO(SIN, -PI, PI, MAXIT, RACINE, IER)
    CALL ZERO(TAN, X, Y, MAXIT, RACINE, IER)
```

Key points:

- Use `EXTERNAL` for user-defined functions/subroutines
- Use `INTRINSIC` for standard functions
- Enables writing generic numerical algorithms
- Powerful technique for mathematical applications

Conclusion

FORTRAN 77 maintains its vital role in scientific computing, combining numerical efficiency with timeless programming principles. This course establishes a structured pathway from fundamental syntax to advanced algorithmic implementation, equipping you with essential skills for computational science. Through this material, you'll develop crucial competencies that extend beyond FORTRAN-specific knowledge:

- **Algorithmic Thinking**: Transforming mathematical concepts into executable code
- **Computational Precision**: Mastering numerical methods at the hardware level
- **Performance Optimization**: Writing efficient, resource-conscious programs
- **Legacy Code Literacy**: Navigating and maintaining critical scientific codebases

The practical exercises and examples serve as foundational elements for real-world scientific programming. Whether you continue with modern Fortran variants or transition to other technical languages, the rigorous approach cultivated here will provide enduring advantages in computational fields. FORTRAN 77 stands as both a historical milestone and a living tool - its core principles continue to influence contemporary scientific computing. The skills you develop will form a robust foundation for tackling complex computational challenges across disciplines.

Appendix

Appendix A - Fortran 77 Instructions

Basic Instructions

Instruction	Description
ASSIGN	Assigns a label value to an integer
BACKSPACE	Positions file pointer to the previous record
BLOCK DATA	Identifies a data block subroutine for initializing variables and arrays
CALL	Calls and executes a subroutine
CHARACTER	Declaration for alphanumeric variables
CLOSE	File closure
COMMON	Global variables, shared among multiple modules
COMPLEX	Declaration for complex variables
CONTINUE	Obsolete instruction, no effect
DATA	Variable initialization
DIMENSION	Array declaration

Control Structures

Instruction	Description
DO	For loop
DO WHILE	While loop (F90)
ELSE	Else part of IF...THEN...ELSE structure
END DO	End of DO or WHILE loop (F90)
END	End of module (program, subroutine,...)
END IF	End of IF construct
EXIT	Premature exit from a DO loop
IF	Alternative IF structure

Declarations and Types

Instruction	Description
DOUBLE	Double precision declaration
EXTERNAL	Identifies a name as a subroutine or function
IMPLICIT	Assigns an implicit type to certain variables
INTEGER	Declaration for integer variables
INTRINSIC	Declaration for intrinsic functions
LOGICAL	Declaration for logical variables
REAL	Real type declaration

Input/Output

Instruction	Description
FORMAT	Read or write format
INQUIRE	File properties examination
OPEN	File opening
PRINT	Screen output
READ	Read operation
WRITE	Write operation
REWIND	Points to beginning of file

Subroutines and Control

Instruction	Description
FUNCTION	Function-type subroutine
GOTO	Jump statement
PARAMETER	Assigns a name to a constant
PAUSE	Temporary program halt
PROGRAM	Program beginning
RETURN	Return from subroutine or function
SAVE	Maintains variables in subroutines
STOP	Program termination
SUBROUTINE	Subroutine name

Appendix B - Intrinsic Functions

Mathematical Functions

Function	Description	Input Type	Output Type
COSH(X)	Hyperbolic cosine	real, double precision, complex	same as input
SINH(X)	Hyperbolic sine	real, double precision, complex	same as input
TANH(X)	Hyperbolic tangent	real, double precision, complex	same as input
ABS(X)	Absolute value (real) or modulus (complex)	real, double precision, complex	real, double precision
MAX(X1,...,XN)	Maximum value	integer, real, double precision	same as input
MIN(X1,...,XN)	Minimum value	integer, real, double precision	same as input

Type Conversion Functions

Function	Description	Input Type	Output Type
INT(X)	Truncation	real, double precision	integer
FLOAT(X) or REAL(X)	Convert integer to single precision real	integer	real
DBLE(X)	Convert to double precision	integer, real	double precision
NINT(X)	Round to nearest integer	real, double precision	integer

Other Functions

Function	Description	Input Type	Output Type
MOD(Y, X)	Remainder of division y/x	integer, integer	integer
CONJG(z)	Complex conjugate	complex, double precision	same as input
SIGN(X1, X2)	Returns sign of X1*X2	integer, real, double precision	same as input
CHAR(I)	Returns ASCII character for code I	integer	character
ICHAR(C)	Returns ASCII code of character C	character	integer
LEN(C)	Returns length of string C	character string	integer

References

Books

- Dubois, P. (1984). *Introduction to Fortran Through Examples*. Éditions Technip.
- Etter, D. M. (1993). *Structured FORTRAN 77 for Engineers and Scientists*. Benjamin/Cummings.
- Fuller, W. R. (1977). *FORTRAN Programming: A Supplement for Calculus Courses*. Springer-Verlag.
- Monro, D. M. (1982). *Fortran 77*. Edward Arnold.
- Kupferschmid, M. (2009). *Classical Fortran: Programming for Engineering and Scientific Applications*. CRC Press.
- Lignelet, P. (1985). *Practical Fortran 77*. Masson.

Online Documentation

- Stanford University Tutorial: https://web.stanford.edu/class/me200c/tutorial_77/
- IDRIS Fortran Training: https://www.idris.fr/formations/fortran/fortran-77
- Fortran 77 Wikibook: https://en.wikibooks.org/wiki/Fortran_77_Tutorial
- University of Leicester Professional Fortran 77: https://www.star.le.ac.uk/~cgp/prof77.html
- University of Strathclyde Course: http://www.strath.ac.uk/CC/Courses/fortran.html
- SoftaDesign Manuals: http://www.softadesign.org/manuels/fortran.html