Capítulo 2

Análisis Numérico y Programación

El análisis del comportamiento de los métodos numéricos cobra importancia al resolver problemas reales de Ingeniería. La solución de estos problemas requiere generalmente del cálculo reiterado de cientos o miles de iteraciones, lo que evidentemente es inviable de realizar a mano. Por esta razón, para poder comprender el funcionamiento interno de los métodos comprobar sus ventajas y desventajas es necesario poseer conocimientos básicos de programación en algún lenguaje imperativo de alto nivel.

En el caso particular de este curso, hemos elegido utilizar el lenguaje FORTRAN por varios motivos. En primer lugar porque se trata de un lenguaje especialmente desarrollado para cálculos numéricos, ya que a diferencia de otros lenguajes de programación, FORTRAN posee tipos de datos intrínsecos para números complejos y está optimizado para realizar operaciones de álgebra lineal con arreglos en forma nativa.

En segundo lugar, dado que la finalidad de esta asignatura no es enseñar a programar, aprovecharemos los conocimientos de programación en lenguaje PASCAL adquiridos asignaturas previas, ya que ambos lenguajes responden al mismo paradigma de programación y sólo se diferencian en cuestiones sintácticas menores.

Y por último, porque al tratarse de un lenguaje de programación estándar, es posible encontrar versiones de diferentes fabricantes y para diferentes sistemas operativos, desde computadoras personales hasta mainframes y desde sistemas privativos bajo licencia hasta software libre y gratuito.

2.1. Introducción a FORTRAN

La unión de las palabras **FOR**mula **TRAN**slator dan nombre a un lenguaje de programación especializado para la implementación de algoritmos y cálculos numéricos, desarrollado por un equipo de científicos, matemáticos e ingenieros, liderado

por John Backus, ganador en 1977 del TURING AWARD, considerado como el premio nobel de la informática, por sus trabajos sobre sistemas de programación de alto nivel y particularmente por el desarrollo del lenguaje de programación FORTRAN.

En 1966 FORTRAN surgió como el primer lenguaje estandarizado, conocido en ese entonces como FORTRAN66, el cuál años más tarde se transformó en FORTRAN77 al cumplir con las normas establecidas por ANSI (American National Standards Institute) y por ISO (International Organization for Standardization). A partir de ese momento, el desarrollo de nuevas versiones de FORTRAN fue aportando mejoras y nueva funcionalidad versión tras versión. En este sentido, a pesar de la aparición de nuevos lenguajes de programación como BASIC, PASCAL, C, C++, Java, C#, etc. FORTRAN sigue siendo considerado como un lenguaje especializado de alto desempeño para cálculos científicos y de Ingeniería.

La versión más reciente de FORTRAN disponible actualmente corresponde al estándar ISO/IEC 1539-1:2010, más conocida como FORTRAN 2008, mientras que la útima versión, actualmente en desarrollo, es conocida informalmente como FORTRAN 2015. Esta versión prevé la inclusión de mejoras en la interoperabilidad entre FORTRAN y el lenguaje C (ISO/IEC TS 29113) y características adicionales relacionadas con el procesamiento en paralelo (ISO/IEC TS 18508).

2.1.1. Tipos de Datos

Los tipos de datos intrínsecos son los definidos por el lenguaje mientras que los tipos de datos derivados son específicos para la aplicación y son definidos por el programador. Los tipos derivados pueden ser declarados, se les puede asignar valores y poseen atributos. Los tipos escalares son aquellos que no son estructurados.

2.1.2. Variables

En la declaración de una variable, el nombre de esa variable está asociado con un único lugar de almacenamiento. Las variables se clasifican según el tipo de dato (como lo son las constantes) al que pertenecen. El tipo de dato de una variable indica el tipo de dato que contiene, incluyendo su precisión, e implica los requerimientos de almacenamiento. Cuando un dato de cualquier tipo es asignado a una variable, este es convertido (de ser necesario) al tipo de dato de la variable.

Ejemplo:

Si **b** es declarada como tipo entera y se le asigna el valor **2.5**, el valor almacenado en **b** será igual a **2**, y no aparecerá ningún error de compilación.

Una variable se define cuando se le da un valor. Una variable puede ser definida antes de la ejecución de un programa en una sentencia DATA o en una sentencia de declaración. Durante la ejecución de un programa, las variables pueden ser definidas o redefinidas en sentencias de asignación o de E/S, o quedar indefinidas (por ejemplo, cuando ocurre un error de E/S). Cuando una variable está indefinida, su valor es impredecible como también todas las variables ligadas por almacenamiento asociado.

El tipo de dato de una variable puede ser explícitamente declarado en una sentencia de declaración de tipo. Sino se declara ningún tipo, la variable tiene un tipo de dato implícito basado en reglas de asignación de tipos predefinidos en una sentencia IMPLICIT. Una declaración explícita de tipo de dato toma precedencia sobre cualquier tipo implícito. El tipo implícito especificado en una sentencia IMPLICIT toma precedencia sobre las reglas de asignación de tipo.

Se recomienda que en cualquier programa FORTRAN y al principio del mismo se use la siguiente sentencia para anular cualquier especificación de tipo implícito:

Usando la sentencia **IMPLICIT NONE** cualquier aparición de variables cuyo tipo no haya sido declarado explícitamente será señalada como un error por el compilador.

2.1.3. Declaración de Datos

2.1.4. Tipos de datos intrínsecos

Enteros

INTEGER [([KIND =] valor-kind)] [[lista-atributos]::] lista-entidades

Reales

REAL [([KIND =] valor-kind)] [[lista-atributos]::] lista-entidades

Complejos

COMPLEX [([KIND =] valor-kind)] [[lista-atributos]::] lista-entidades

Lógicos

LOGICAL [([KIND =] valor-kind)] [[lista-atributos]::] lista-entidades

2.2. Formato de los Programas

Un programa en FORTRAN consta de las siguientes secciones:

[PROGRAM nombre del programa]
[sección de especificación]
[sección ejecutable]
[sección de sub-programas internos]
END [PROGRAM [nombre de programa]]

Un programa extremadamente corto, que por cierto no hace nada, podría ser:

```
PROGRAM prueba
END
```

Donde **prueba** es el nombre del programa. La sentencia **END** es la única parte obligatoria de un programa. Si la sentencia **END** incluye el nombre del programa, éste debe ser el mismo que se ha colocado a continuación de la sentencia **PROGRAM**.

El nombre del programa principal es considerado un nombre global, junto con el nombre de los procedimientos externos y los bloques comunes. Los nombres globales deben ser únicos dentro de un programa.

2.3. Estructuras de Control de Decisión

FORTRAN ejecuta secuencialmente las sentencias de un programa, de la primera a la última. Esta secuencia puede modificarse por medio de la ejecución de bloques de código pertenecientes a procedimientos y funciones o transfiriendo el control de ejecución hacia otras sentencias del programa. Las estructuras de control de decisión nos permiten ejecutar diferentes bloques de código según se cumpla o no una determinada condición. La sentencia que implementa la estructura de control de decisión en FORTRAN es la sentencia IF. Dicha sentencia admite tres formatos distintos, según se muestra a continuación.

2.3.1. Sentencia IF

En el caso más simple, tenemos la sentencia **IF**, utilizada para ejecutar un bloque de código si el resultado de la condición es verdadero. En caso contrario, la ejecución prosigue, sin ejecutar el mismo. En la figura 2.1 podemos observar el

diagrama de flujo correspondiente a la estructura mencionada.

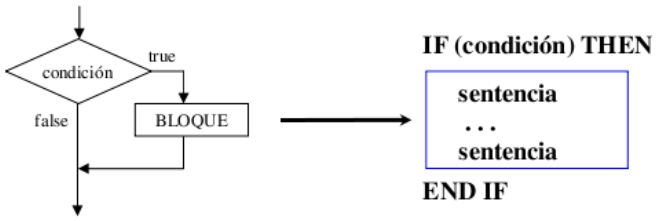

Figura 2.1: Estructura IF.

A continuación vemos una posible implementación:

```
IF (a < b) THEN
   aux = a
   a = b
   b = aux
END IF
```

NOTA: Se debe respetar la sintaxis previa (IF (condicion) THEN en el mismo renglón y se cierra la estructura con END IF).

Una variante de la sentencia IF, es la sentencia IF.. ELSE. En este caso, si el resultado de la condición es verdadero se ejecuta un bloque de código, y si el resultado es falso, se ejecutará otro bloque de código diferente. En la figura 2.2 podemos observar el diagrama de flujo correspondiente a la estructura mencionada.

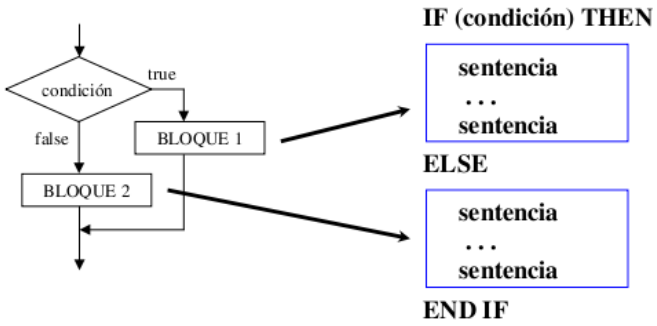

Figura 2.2: Estructura IF-ELSE.

A continuación vemos una posible implementación:

```
IF (leftCornerX < 0) THEN
   leftCorner = 0
ELSE
   aux = leftCornerX
   leftCornerX = rightCornerX
   rightCornerX = aux
END IF
```

Por último, tenemos la posibilidad de anidar consultas a varias condiciones en una misma sentencia. Esto se logra por medio de la sentencia IF... ELSE IF. Esto permite ejecutar diferentes bloques de código para cada resultado verdadero de cada condición y un solo bloque de código final para el caso en que todos los resultados de las condiciones resulten falsos. En la figura 2.3 podemos observar el diagrama de flujo correspondiente a la estructura mencionada.

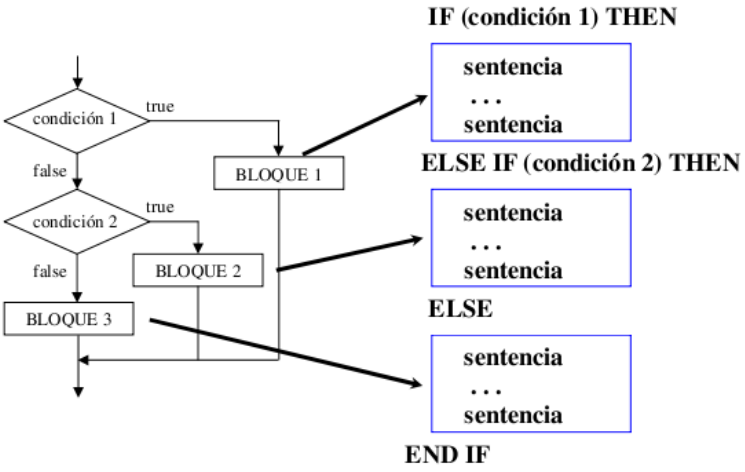

Figura 2.3: Estructura IF-ELSE-IF.

Es importante aclarar que una sentencia IF puede contener varias sentencias ELSE IF, pero no puede contener mas de una sentencia ELSE.

Una posible implementación, podría ser:

```
IF (kWatts < 50) THEN
  costo = 30
ELSE IF (kWatts < 100) THEN
  costo = 20 + 0.5*kWatts
ELSE IF (kWatts < 150) THEN
  costo = 15 + 0.3*kWatts
ELSE IF (kWatts < 200) THEN
  costo =  5 + 0.2*kWatts
ELSE
  costo =  0.15*kWatts
END IF
```

A continuación presentamos un programa para calcular el argumento de un número complejo, como un ejemplo de utilización de las estructuras IF.

```
PROGRAM arg_comp
IMPLICIT NONE

! Calculo del argumento de un numero complejo
REAL, PARAMETER:: Pi = 3.141592654

! Declaracion de variables
REAL re, im

! Seccion ejecutable
PRINT *,'Ingrese la parte real: '
READ *, re
PRINT *,'Ingrese la parte imaginaria: '
READ *,im

IF(im == 0) THEN
  IF (re > 0) THEN
    PRINT *,'Argumento : ', 0
  ELSE IF (re < 0) THEN
    PRINT *,'Argumento : ', Pi
  ELSE
    PRINT *,' Argumento Indeterminado '
  END IF
ELSE
  IF (re == 0) THEN
    IF (im > 0) THEN
      PRINT *,'Argumento : ', Pi/2
    ELSE
      PRINT *,'Argumento : ', -Pi/2
    END IF
  ELSE IF (re > 0) THEN
    PRINT *,'Argumento : ', ATAN(im/re)
  ELSE IF (im > 0) THEN
    PRINT *,'Argumento : ', (ATAN(im/re) + Pi)
  ELSE
    PRINT *,'Argumento : ', (ATAN(im/re) - Pi)
  END IF
END IF

END PROGRAM arg_comp
```

En FORTRAN existe además una sentencia denominada IF lógico, que permite controlar la ejecución de una sola sentencia, pero no se trata de una estructura. Por ejemplo:

```
IF (p > max) max = p
```

2.3.2. Sentencia SELECT CASE

La sentencia SELECT CASE ofrece una estructura de control de decisión similar a un conjunto de IF anidados, derivando la ejecución hacia diferentes bloques de código, de acuerdo a cada una de las opciones presentes. En la figura 2.4 podemos observar el diagrama de flujo correspondiente a la estructura mencionada.

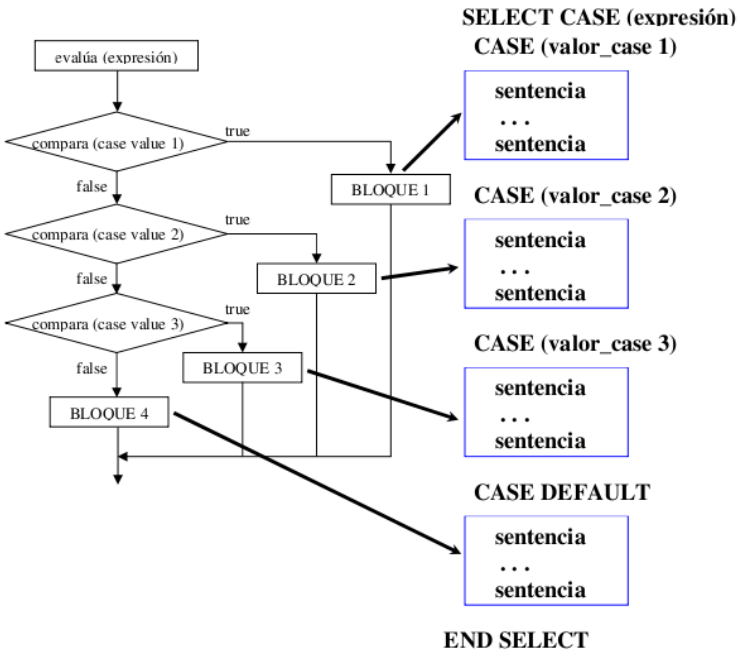

Figura 2.4: Estructura SELECT CASE.

Si bien la estructura SELECT CASE presenta un comportamiento similar a un conjunto de IF anidados, existen ciertas diferencias como veremos a continuación. Una vez que la expresión es evaluada, el resultado es denominado índice CASE. Los valores de los selectores **case** pueden tomar cuatro formas diferentes:

- (valor-case)

- (rango-inferior :)
- (: rango-superior)
- (rango-inferior : rango-superior)

Cada valor case debe ser del mismo tipo de datos que la expresión. Los tipos de datos permitidos son: *integer*, *character* y *logical*. Para los caracteres, los valores CASE no necesitan tener la misma longitud que la expresion.

Para el caso de los valores **case**, el bloque de código a ejecutar es aquel que es idéntico al resultado de la expresión. En el caso del rango inferior el resultado de la comparación es verdadero si el valor de la expresión es mayor o igual al rango inferior. En el caso del rango superior el resultado de la comparación es verdadero si el valor de la expresión es menor o igual al rango superior. Los rangos de valores no están permitidos para los tipos de dato *logical*.

Si alguna de las comparaciones resulta verdadera, solo se ejecuta un bloque de código, es decir, no se produce ejecución en cascada, por lo que no es necesario explicitar la salida de un bloque.

La sentencia CASE DEFAULT es opcional, la ejecución del bloque de código correspondiente a la misma se produce si los resultados de todas las comparaciones resultaron falsos. Si esto sucede y no existe ninguna sentencia CASE DEFAULT, la ejecución continúa fuera de la estructura CASE.

A continuación implementaremos uno de los ejemplos anteriores, esta vez por medio de la estructura CASE

```
SELECT CASE (kWatts)
  CASE (:49)
    costo = 30
  CASE (50:99)
    costo = 20 + 0.5*kWatts
  CASE (100:149)
    costo = 15 + 0.3*kWatts
  CASE (150:199)
    costo =  5 + 0.2*kWatts
  CASE DEFAULT
    costo =  0.15*kWatts
END SELECT
```

2.4. Estructuras de Repetición

2.4.1. Ciclo DO

La estructura DO permite ejecutar un bloque de código una cierta cantidad de veces. Para aquellos con experiencia en otros lenguajes de programación, recono-

cerán inmediatamente que se trata de una estructura similar al FOR de PASCAL ó C.

El DO de FORTRAN al igual que el FOR de C ó PASCAL, posee una variable que controla la ejecución del ciclo. Esta variable permite que el bloque de código se ejecute mientras el valor de la misma varía desde un valor inicial hasta un valor final, de acuerdo a un incremento determinado, el cual puede ser positivo o negativo. Si bien por compatibilidad con versiones anteriores de FORTRAN, se permite la utilización de un valor real para el incremento, esta característica actualmente se considera obsoleta y se sugiere utilizar solamente incrementos enteros. Si se omite el incremento, se adopta por defecto el valor 1.

En la figura 2.5 podemos observar el diagrama de flujo correspondiente a la estructura mencionada.

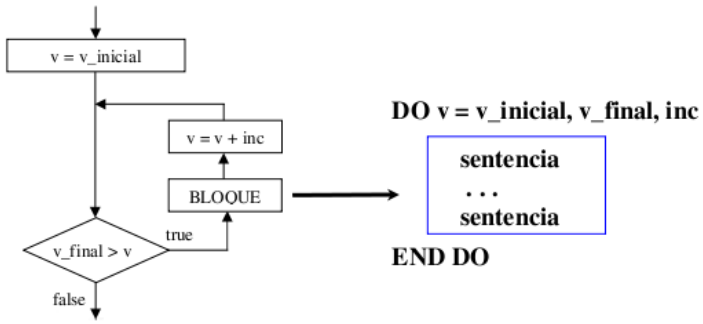

Figura 2.5: Ciclo DO.

La estructura DO puede contener a otras estructuras DO, IF ó SELECT CASE, siempre y cuando las mismas queden completamente contenidas dentro del bloque DO END DO. A continuación vemos una implementación de la estructura DO:

```
DO fila = 1, maxFilas
  DO col = 1, maxCols
    mat( fila , col ) = mat( fila , col ) / scale
  END DO
END DO
```

Si bien es posible alterar la ejecución normal de la estructura DO por medio de las sentencias CYCLE y EXIT, no se aconseja la utilización de las mismas, ya que no son necesarias, solo existen por compatibilidad con FORTRAN77.

2.4.2. Ciclo DO WHILE

La estructura DO WHILE permite ejecutar un bloque de código mientras se cumpla una determinada condición. Como puede apreciarse, se trata de una estructura bastante similar a la estructura DO ya vista, pero más flexible, pues no

es necesario conocer de antemano cuántas veces deberá ejecutarse el ciclo. En la figura 2.6 podemos observar el diagrama de flujo correspondiente a la estructura mencionada.

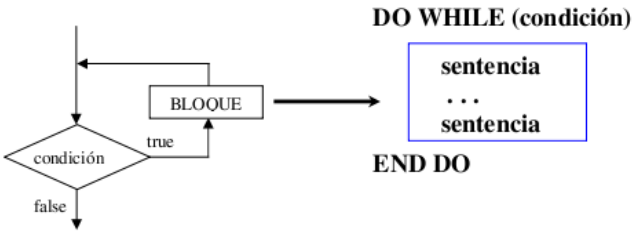

Figura 2.6: Ciclo DO.

Como puede apreciarse en el diagrama, el bloque se ejecuta hasta que la condición se vuelve falsa. La estructura DO WHILE no posee ningún contador de ciclos interno, por lo que el código que pueda hacer cambiar el resultado de la condición deberá incluirse dentro del bloque ejecutable.

```
DO WHILE (sigue /= 'n')
  WRITE (*, 'Desea␣continuar␣?␣␣s/n␣')
  READ (*, '') sigue
END DO
```

Nota: Los caracteres ␣ representan espacios.

2.5. Arreglos (Arrays)

Cuando se trabaja con muchas variables de un determinado tipo, es frecuente que las mismas se almacenen en un arreglo. Podría decirse que un arreglo es una variable que posee al menos una dimensión (vector, lista, etc.). Un arreglo puede ser estático ó dinámico. Si es estático, la reserva de la cantidad de memoria necesaria para almacenar los datos se realiza en tiempo de compilación, en ese caso su tamaño no puede modifcarse ni liberarse hasta la finalización del programa. Si es dinámico la cantidad de memoria para almacenar datos en el arreglo puede modificarse durante la ejecución del programa.

2.5.1. Especificación de Arreglos

El número de dimensiones de un arreglo es denominado rango (rank) y el número total de elementos es el tamaño (size) del arreglo. La forma (shape) de un arreglo está determinada por su rango y la extensión de cada dimensión. Por ejemplo:

```
REAL matriz(10, 3, 2)
```

es un arreglo de rango 3, que tiene un tamaño de 60 elementos (10*3*2) y cuya forma es diez, tres, dos, es decir (10,3,2). El número total de elementos en una dimensión en particular es la extensión (extent) de la dimensión. Cada dimensión puede especificarse con un límite inferior (lower bound) y un límite superior (upper bound), separados por dos puntos. Por ejemplo:

```
REAL matriz(0:9, -1:1, 4:5)
```

El número menor en la especificación de la dimensión es el límite inferior y el número mayor es el límite superior. El límite inferior puede ser cualquier valor positivo, negativo o cero y se omite se adoptará 1 como valor por defecto. Un arreglo puede especificarse con una declaración de tipo, una sentencia DIMENSION ó una sentencia ALLOCATABLE. Las siguientes son declaraciones válidas de arreglos

```
REAL A                          ! Declaracion de TIPO.
DIMENSION A(10, 2, 3)}          ! Declaracion de DIMENSION.
REAL, DIMENSION (2, 5) :: A     ! Declaracion de tipo y dimension
```

Un escalar es un tipo de dato simple, tal como un número o un tipo derivado, en cambio, un arreglo es una colección de escalares. Cada escalar dentro de un arreglo es denominado un elemento. Un escalar posee rango cero y un arreglo con al menos una dimensión posee rango uno. Todos los elementos de un array deben tener el mismo tipo, por ejemplo, todos deben ser INTEGER(2), o REAL(8), o cadenas de caracteres de la misma longitud o todos del mismo tipo derivado. Para referenciar un arreglo completo se utiliza el nombre, por ejemplo :

```
REAL A (10), B(10),C(10)

A = 3.0         ! Establece el valor de todos los elementos de A igual a 3.0.

B = SQRT(A)     ! Establece el valor de todos los elementos de A igual a
                ! la raiz cuadrada de 3.0.

C = A + B       ! Suma los vectores A y B, y almacena el resultado en C.
```

Cuando se declara un arreglo puede no especificarse una o más de sus propiedades (tamaño y forma) para que sean determinadas posteriormente durante la ejecución del programa. Sin embargo, el rango (número de dimensiones) de un arreglo debe especificarse siempre.

2.5.2. Arreglos estáticos

Un arreglo estático o con formato explícito tiene una extensión especificada para cada una de sus dimensiones. Opcionalmente puede tener especificados o no los límites inferiores de alguna o de todas sus dimensiones. Por ejemplo:

```
INTEGER matrizA( 10, 10, 10)
INTEGER matrizB( -3:6, 4:13, 0:9 )
```

En ambos casos se trata de arreglos con formato explícito de rango 3 y tamaño 1000 que poseen la misma forma (10,10,10). En el primer caso, no se han declarado los límites inferiores, por lo que se asume 1 como valor por defecto. En el segundo caso puede apreciarse que los límites inferiores pueden adoptar valores positivos, negativos o cero.

2.5.3. Arreglos dinámicos

En ciertas ocasiones, el tamaño de un arreglo sólo se conoce en tiempo de ejecución. Otras veces, se necesita trabajar con arreglos auxiliares, los cuales podríamos eliminar cuando no sean necesarios. En todos estos casos, declarar los arreglos en forma estática, significa reservar una cantidad innecesaria de memoria. Para evitar esto, es que se recurre a los arreglos dinámicos.

Los arreglos dinámicos se declaran sin especificar su tamaño, agregando el modificador ALLOCATABLE en la declaración. Una vez que se conoce el tamaño requerido se reserva el espacio necesario por medio de la instrucción ALLOCATE. Cuando ya no se necesita más el arreglo, se puede liberar el espacio de memoria reservado, por medio de la instrucción DEALLOCATE.

Por ejemplo:

```
REAL(8), PARAMETER :: PI=3.141592654
REAL(8), ALLOCATABLE :: A(:,:), B(:)
INTEGER i,j,n

  READ (*,*), n
  ALLOCATE(A(n,n), B(n))

  DO i=1, n
     B(i)= SIN((i-1)*PI)
     DO j=1, n
        A(i,j) = SQRT(i+j)
     END DO
  END DO

  DEALLOCATE(A, B)
```

En el fragmento de código anterior se observa que la cantidad de filas y columnas del arreglo A y la cantidad de elementos del arreglo B, se ingresan en tiempo de ejecución por medio de la instrucción READ. Una vez conocido el tamaño de los arreglos, se reserva la memoria necesaria para los mismos con ALLOCATE. Y cuando ya no se precisa, se libera el espacio por medio de DEALLOCATE.

2.5.4. Referenciar elementos de un Arreglo

Los elementos de un arreglo son referenciados por medio de subíndices. Por ejemplo,

```
REAL A(12, 8, 4)

A(1, 1, 1) = 0
```

asigna el valor cero al primer elemento del arreglo A.

En memoria, el elemento **n+1** se almacena a continuación del elemento **n**. Es decir los elementos están organizados en una secuencia lineal, incluso en el caso de los arreglos multidimensionales, debido a que la memoria de la computadora tiene una sola dimensión. El subíndice que se encuentra más a la izquierda es el que se incrementa primero (por columnas), luego el subíndice que le sigue hacia la derecha, y así sucesivamente hasta el último subíndice. Por ejemplo, los ocho elementos del arreglo A(2, 2, 2), se almacenan en memoria en el siguiente orden:

A(1, 1, 1); A(2, 1, 1); A(1, 2, 1); A(2, 2, 1); A(1, 1, 2); A(2, 1, 2); A(1, 2, 2); A(2, 2, 2)

Los subíndices de un arreglo deben estar separados por comas. Un subíndice es una constante, variable o expresión entera, que puede ser positiva, negativa o cero. El valor de la misma debe encontrarse dentro de los límites de la dimensión a la que hace referencia. El número de expresiones de subíndices debe ser igual al número de dimensiones de la declaración del arreglo. Es posible utilizar funciones y elementos de arreglos como subíndices. Por ejemplo:

```
REAL A(3, 3), B(3, 3), C(89), R

B(2, 2) = 4.5                    ! Asigna el valor 4.5 al elemento B(2, 2)

R = 7.1

C(INT(R)*2 + 1) = 2.0            ! Asigna el valor 2.0 al elemento C(15)

A(1,2) = B(INT(C(15)), INT(SQRT(R))) ! Asigna el contenido del elemento
                                     ! A(1,2) al elemento B(2,2)
```

Anteriormente hemos visto como referenciar un elemento de un arreglo, sin embargo, FORTRAN permite referenciar conjuntos de elementos de un arreglo, con lo cual se simplifican enormemente ciertas operaciones con arreglos y las hacen más eficientes.

A continuación se muestran algunos ejemplos:

```
REAL A(10, 10)
INTEGER i,j

  DO i=1, 10
    A(i,:)=A(i,:)/A(i,i)   ! Divide a todos los elementos de la fila i
                           ! por el elemento A(i,i)
    A(i,4:8)=A(i,4:8)*2    ! Multiplica por 2 desde el 4to. al 8vo. de
                           ! la fila i
    A(i,:4)=A(i,:4)/3.0    ! Divide por 3.0 desde el 1er. al 4to. de
                           ! la fila i
  END DO

  A(:,9) = A(:,9) + 1      ! Suma 1 al 9no. elemento de cada fila
```

2.6. Utilización de Archivos

Los archivos permiten almacenar datos para que puedan ser utilizados por otros programas o simplemente para resguardarlos una vez finalizado el programa que los generó y posteriormente recuperarlos fácilmente. En FORTRAN, como en otros lenguajes de programación es posible trabajar con diferentes tipos de archivos, de texto, de registros, con acceso secuencial, acceso directo o por clave. Por el momento, veremos solamente el caso de los archivos de texto con formato, ya que son muy sencillos de generar y recuperar, y muchos programas son capaces de importarlos con facilidad, además de brindarnos la posibilidad de crearlos, examinarlos y modificarlos con un simple editor de textos.

Para comenzar a trabajar con un archivo es necesario abrirlo con la instrucción OPEN.

OPEN ([UNIT=] unidad [, ACCESS=acceso] [, ACTION=accion]
[, DELIM=delim] [, ERR=err] [, FILE=archivo]
[, STATUS=estado])

El primer parámetro de la instrucción OPEN, UNIT es el número de unidad que especifica el dispositivo lógico asociado (conectado) a un archivo externo físico (existente en un medio externo). Este especificador es un número entero no negativo o un asterisco. Sin embargo, es preciso recordar que generalmente el valor 5 está reservado para el teclado, el valor 6 está reservado para la pantalla y elvalor 0 está reservado para el kernel del sistema operativo. En consecuencia puede utilizarse cualquier valor entero no negativo diferente de 0, 5 ó 6. La utilización de un asterisco significa que las operaciones de lectura y/o grabación se harán sobre los dispositivos estándar de entrada-salida.

Es posible acceder a los archivos en forma secuencial, directa ó en una modalidad que permite agregar información al final del mismo, esto se especifica por medio del parámetro ACCESS y sus posibles valores son: SEQUENTIAL, DIRECT y APPEND, respectivamente.

Un archivo puede utilizarse para lectura, grabación ó para lectura-grabación, esto se especifica por medio del parámetro ACTION, siendo sus posibles valores READ, WRITE ó READWRITE.

Con el parámetro FILE se especifica el nombre del archivo. Si solo se especifica el nombre, se asumirá que el archivo se encuentra en el directorio local, de lo contrario será necesario especificar toda la trayectoria (*path*).

En ciertas ocasiones se utilizan archivos temporarios, por lo que no es preciso que los mismos permanezcan una vez que el programa ha finalizado, para ello se utiliza el parámetro STATUS. Si un archivo se abre con STATUS=SCRATCH, el mismo se borrará después del CLOSE. En cambio si se utiliza STATUS=REPLACE, el archivo será creado si no existe y si existe se lo reemplazará por uno nuevo.

Por ejemplo:

```
OPEN (2, FILE = 'miArchivo', ACCESS = 'SEQUENTIAL', STATUS = 'REPLACE')
```

2.6.1. Grabación y lectura desde archivos

FORTRAN permite interactúar con diversos dispositivos, ya sea de entrada como el teclado, de salida como la pantalla o la impresora y de entrada-salida como los archivos externos. Esta interacción se realiza principalmente por medio de las instrucciones READ y WRITE.

WRITE ([UNIT=] unidad [, FMT=] fmt]
[, ERR=err]) lista de salida

```
PROGRAM grabaCoord

  IMPLICIT NONE

INTEGER n, i
REAL(8) x, y, z

!Abre un archivo, si no existe lo crea, si existe lo reemplaza
  OPEN(UNIT=2, FILE='coordenadas',STATUS='REPLACE')

  WRITE (*, '(A28)', ADVANCE='NO'),'Ingrese cantidad de puntos: '
  READ *, n

  DO  i=1, n
    WRITE (*, '(A2, I3, A4)', ADVANCE='NO') 'X(', i, ') = '
    READ *, x
    WRITE (*, '(A2, I3, A4)', ADVANCE='NO') 'Y(', i, ') = '
    READ *, y
    WRITE (*, '(A2, I3, A4)', ADVANCE='NO') 'Z(', i, ') = '
    READ *, z
! Graba las coordenadas en el archivo
    WRITE (2,'(3F15.5)') x, y, z
  END DO
```

```
! Cierra el archivo
  CLOSE (2, STATUS='KEEP')

END PROGRAM
```

Tambien podemos hacer el proceso inverso y leer los datos grabados. Para ello es conveniente utilizar el mismo formato usado en la grabación, de esta forma los datos serán interpretados correctamente.

```
PROGRAM leeCoord

  IMPLICIT NONE

INTEGER n, i
REAL(8) x, y, z

!Abre un archivo, si no existe lo crea, si existe lo reemplaza
  OPEN(UNIT=2, FILE='coordenadas')

  DO WHILE (.TRUE.)
     ! Lee las coordenadas desde el archivo
    READ (2,'(3F15.5)', END=99) x, y, z

     ! Imprime las coordenadas en pantalla
    WRITE (*,'(3F15.5)') x, y, z
  END DO

! Esta sentencia se ejecuta cuando encuentra el final del archivo
99 CONTINUE

! Cierra el archivo
  CLOSE (2, STATUS='KEEP')

END PROGRAM
```

2.7. Declaración de Funciones

Debido a que las funciones y subrutinas son bastante similares, se suele describir las características comunes, denominándolas indistintamente como sub-programas. La mayoría de las siguientes características, son también aplicables a subprogramas externos, a menos que se diga expresamente lo contrario. Los subprogramas internos se ubican entre las sentencias CONTAINS y END del programa principal. Los sub-programas son muy parecidos al programa principal, excepto por la sentencia CONTAINS y END, es decir los sub-programas no pueden contener sub-programas, por lo que no pueden contener sentencias CONTAINS. La sintaxis general de una función interna es:

FUNCTION Nombre([lista de argumentos])
[sentencias de declaración]
[sentencias ejecutables]
END FUNCTION [Nombre]

La sentencia **FUNCTION Nombre([lista de argumentos])** es denominada sentencia función, o encabezamiento, o declaración. Note que si el programa principal tiene una sentencia IMPLICIT NONE (y es una buena práctica de programación que la tenga) el nombre de la función y sus argumentos deben declararse con sus respectivos tipos. Aunque esto puede hacerse en el programa principal, se recomienda que el nombre de la función y sus argumentos se declaren en el cuerpo mismo de la función. Debido a que se asocia un valor con el nombre de la función, este valor debe ser asignado al nombre de la función en el cuerpo de la misma. En este caso el nombre debe aparecer en el lado derecho de una expresión de asignación, como se aprecia en el siguiente ejemplo:

dist = SQRT(x*x + y*y)

Todo sub-programa interno tiene acceso directo a las variables declaradas en el programa principal, sin embargo no es una buena práctica de programación hacer uso de esta posibilidad, ya que suele conducir a programas poco modulares y puede producir errores muy difíciles de detectar. Por otra parte, si una variable es redeclarada en un sub-programa, el mismo no tendrá acceso a la variable del mismo nombre declarada en el programa principal. Por lo tanto, no existe ningún problema en declarar una variable interna con el mismo nombre que una variable global, ya que el compilador las tratará como dos variables diferentes.

2.8. Especificación de parámetros de entrada - salida

La sentencia **INTENT** especifica el uso intencional de los parámetros formales dentro de un procedimiento. El uso del atributo INTENT protege de acciones no deseadas. Si no se especifica, el argumento queda sujeto a las limitaciones de la asociación de argumentos. Una sentencia **INTENT** solo puede aparecer en la sección de especificación de un subprograma o en el cuerpo de una interfaz. El tipo del argumento actual en la invocación debe corresponderse con el del parámetro formal. Si el argumento es **INTENT(IN)**, puede suministrarse una expresión o constante del tipo requerido; si este es **INTENT(OUT)** o **INTENT(INOUT)**, se debe suministrar una variable.

Sintaxis del atributo INTENT: **especif-tipo, INTENT (spec) :: nombre-variables**

spec Especifica como un procedimiento usa un argumento formal. Puede ser uno de los siguientes:

- **IN** El argumento formal es de entrada al procedimiento. No puede ser redefinido (i.e. aparecer en el miembro izquierdo de una asignación, o ser pasado como un argumento actual al procedimiento que lo redefine) o quedar indefinido mientras se ejecuta el procedimiento. Si se intenta redefinir dentro del procedimiento resultará un error de compilación.
- **OUT** El argumento formal es de salida del procedimiento. El procedimiento debe definir el argumento antes de usarlo. Cualquier argumento actual asociado con el argumento formal debe ser una variable (definible). El correspondiente argumento actual debe ser una variable y queda indefinido a la entrada del procedimiento porque la intención es usarlo solo para extraer información.
- **INOUT** El argumento formal puede recibir datos como devolver datos al programa que invoque al procedimiento. El correspondiente argumento actual debe ser una variable porque se espera que sea redefinido por el procedimiento.

nombre-variables Una o más variables, separadas por comas.

especif-tipo Cualquier tipo de especificación intrínseca o derivada.

Utilización del atributo INTENT:

```
SUBROUTINE EX (a, b)
INTENT (INOUT) :: a, b }
```

Utilización de la sentencia INTENT:

```
SUBROUTINE inversa(A, singular)
REAL, DIMENSION(:,:), INTENT (INOUT) :: A
LOGICAL;INTENT(OUT) :: singular}
```

2.8.1. Funciones, Subrutinas y Arreglos

Los límites superiores e inferiores de un arreglo, pueden especificarse por medio de variables, o expresiones en funciones y sub-rutinas. Los valores de los límites son determinados al evaluar las variables o expresiones en el momento de ingresar a la sub-rutina. Por lo tanto, si se producen cambios en las variables o expresiones dentro de la sub-rutina, los mismos no tienen influencia sobre los límites del arreglo. Por ejemplo:

```
SUBROUTINE miSub (N, R1, R2)
INTEGER N
REAL R1, R2
INTEGER A(N, 5), B(10*N)

  N = INT(R1) + INT(R2)

END SUBROUTINE miSub
```

Al invocar a la sub-rutina, se reserva el espacio para los arreglos A y B por medio del valor N, que es pasado a la misma como parámetro. Los cambios posteriores del valor N no tienen influencia en el espacio reservado para A y B. Si un arreglo con formato explícito utiliza variables o expresiones para determinar sus dimensiones, debe ser un parámetro formal, el resultado de una función o un arreglo automático. Un arreglo automático es un arreglo con formato explícito declarado en un procedimiento, no es un parámetro formal y sus dimensiones son expresiones variables. En el ejemplo anterior, los arreglos A y B son arreglos automáticos.

Las funciones y las sub-rutinas son fragmentos de código generados para realizar una determinada tarea, la cual es ejecutada varias veces en el programa, con lo cual se logra una mejor separación de las tareas, además de evitar la escritura redundante de código. Una de las principales diferencias entre una función y una sub-rutina, es que la función siempre devuelve un resultado. Por ejemplo, en la funcion *factorial(n)*, el resultado es un número entero.

```
FUNCTION factorial(n)
INTEGER factorial, n, aux

aux = 1

  DO i=2, n
    aux = i*aux
  END DO

  factorial = aux
END FUNCTION

END
```

Este resultado no necesariamente debe ser un escalar, sino que también puede ser un arreglo, como vemos en el siguiente ejemplo:

```
FUNCTION hilbert(n)
INTEGER i,j,n
REAL, DIMENSION(n,n) :: hilbert

DO i=1, n
  DO j=1, n
    hilbert(i, j)=1.0/(i+j+1)
  ENDDO
ENDDO
```

```
END FUNCTION
```

Nótese en el ejemplo anterior que en la división se escribió **1.0/(i+j+1)**, en vez de **1/(i+j+1)**. Se deja al lector inquieto que experimente con ambas versiones y saque sus propias conclusiones.

2.9. La sentencia CONTAINS

La sentencia **CONTAINS** marca el límite entre la sección ejecutable de un programa y cualquier subprograma interno que pueda contener. Separa los procedimientos internos de los procedimientos anfitriones, y separa la parte de especificación de un módulo de los procedimientos modulares. Si un programa contiene subprogramas internos, éstos deben definirse luego de la sentencia **CONTAINS**. A continuación de una sentencia **CONTAINS** pueden definirse una o varias funciones, pero los procedimientos internos no pueden contener sentencias **CONTAINS** en sí mismos. El siguiente ejemplo contiene la función interna *dist(x, y)*, que calcula la distancia al origen de un punto. El programa principal permite ingresar las coordenadas y luego imprime el resultado calculado por la función.

```
PROGRAM miPrograma

  IMPLICIT NONE

 ! Ejemplo de utilizacion de una funcion interna
REAL a, b

  READ *, a, b
  PRINT *, dist(a, b)

CONTAINS

FUNCTION dist(x, y)
REAL, INTENT(IN) :: x, y, dist

  dist = SQRT(x*x + y*y))

END FUNCTION dist

END
```

2.10. Variables Locales y Globales

A continuación presentaremos un ejemplo donde se aprecia claramente un problema que puede presentarse si no se declaran adecuadamente las variables locales

y globales. El siguiente programa contiene la función interna *Fact(n)*, que calcula el factorial de un número entero positivo. Sin embargo, el mismo no devuelve la salida esperada. Analicemos el código mas detenidamente, para averiguar que es lo que sucede.

```
PROGRAM Factorial

IMPLICIT NONE

INTEGER I

DO I = 1, 10
  PRINT*, I, Fact(I)
END DO

CONTAINS

FUNCTION Fact(n)
INTEGER Fact, n, aux

aux = 1

  DO i=2, n
    aux = i*aux
  END DO

  Fact = aux
END FUNCTION

END
```

El problema se debe a que la variable **i**, se ha definido en el programa principal por lo que se trata de una variable global. Sin embargo, esta misma variable es referenciada a su vez, dentro de la función *Fact*.

La primera vez que *Fact* es invocada, **i** tiene el valor **1**. Este valor es pasado a la función *Fact* en reemplazo del parámetro formal **n**. Dentro de la función *Fact*, se asigna a **i** el valor **2** en el ciclo **DO** y como este valor es mayor que **n**, el ciclo **DO** no es ejecutado. Sin embargo, el valor de **i** ha sido modificado a **2** y por lo tanto cuando la la ejecución retorna de la función *Fact* al programa principal, siendo éste el valor que se imprimirá.

Ahora el valor de **i** se incrementará a **3** y se efectuará la segunda invocación a *Fact*. De esta forma podemos ver que la función *Fact*, nunca se invocará para valores pares de **i**, como consecuencia de que **i** es tomada como variable global.

Este problema se soluciona fácilmente re-declarando la variable **i** como local, dentro de la función *Fact*. Como regla general es conveniente declarar todas las variables utilizadas en los sub-programas. Tenga en cuenta que no existe ningún problema en llamar **i** tanto a la variable local como a la global, ya que el compilador es capaz de determinar el ámbito de validez por su declaración. Es decir, si en un programa se declara una variable global llamada *velocidad* y esa variable se pasa como un parámetro a una función, es conveniente que la variable local de la función

también se llame *velocidad* y no *velocidad2*, ó *vv* u otro nombre ficticio, ya que lo único que se logra con esto es un código más confuso y menos entendible.

Si dentro de un sub-programa se necesita alguna información contenida en variables que se hallan definidas en el programa principal, la forma más segura de hacerlo es pasándola al sub-programa a través de sus parámetros formales. De esta forma se evitará la utilización inadvertida de variables globales en un contexto erróneo.

2.11. Invocación a programas externos

Muchas veces es necesario añadir cierta funcionalidad a los programas sin implementarla directamente, sino solicitándole a otros programas que la realicen. Por ejemplo, para borrar la pantalla, podemos invocar la función **clear** del sistema operativo, de la siguiente manera:

```
CALL SYSTEM ("clear")
```

También podemos invocar a otros programas de la misma forma. Por ejemplo, podemos ejecutar el programa **Gnuplot** desde un programa **FORTRAN**, de la siguiente manera.

```
CALL SYSTEM ("gnuplot")
```

De esta forma podemos utilizar el **Gnuplot** en forma interactiva y luego de finalizar (ingresando **quit** o **exit**), continuar con la ejecución del programa principal.

También es posible invocar a un programa enviándole un conjunto de parámetros para que realice una determinada tarea. Por ejemplo, el programa **Gnuplot** puede utilizarse en modo interactivo, como vimos anteriormente, ó en modo batch, es decir, ejecutando un conjunto de instrucciones (propias de Gnuplot) almacenadas en un archivo de texto *(script)*. Por medio de la siguiente sentencia se invoca al programa **Gnuplot** para que realice un gráfico, de acuerdo a lo establecido en un archivo denominado **'miScript.p'**. (el parámetro **-persist** es para impedir que la ventana del gráfico se cierre inmediatamente después de graficar)

```
CALL SYSTEM("gnuplot␣-persist␣'miScript.p'")
```

Nota: Los caracteres ␣ representan espacios.

2.12. Programación Modular en FORTRAN

Cuando los programas son simples e involucran poco código, es común colocar toda la funcionalidad dentro del programa principal. Sin embargo, si se desea realizar un programa de relativa complejidad, es mucho más conveniente estructurar la funcionalidad del mismo en varios procedimientos y funciones, los cuales pueden ser internos o externos. En la figura 2.7 podemos observar la estructura de un programa modular.

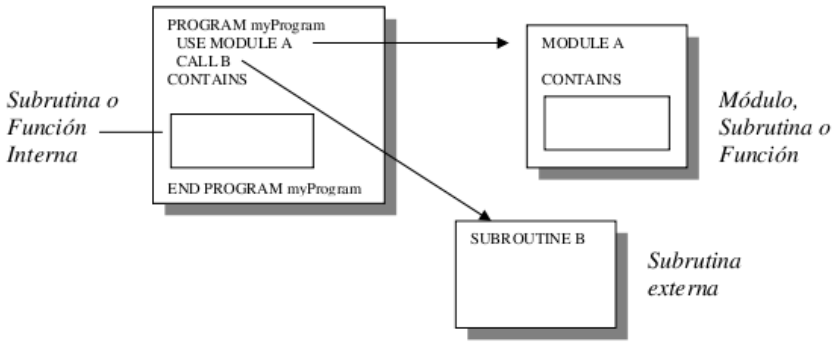

Figura 2.7: Programación modular.

El programa **myProgram** contiene subrutinas internas, invoca a la subrutina externa B, así como también a los procedimientos modulares contenidos en el Módulo A.

Cuando en FORTRAN hablamos de procedimientos en forma genérica nos referimos tanto a subrutinas como a funciones. En este contexto, podemos diferenciar cuatro tipos de procedimientos.

2.12.1. Procedimientos Externos

Los procedimientos externos son funciones o subrutinas que escribe el programador y que se encuentran fuera del programa principal. Los mismos pueden almacenarse en archivos fuente separados o pueden incluirse dentro del mismo archivo del programa principal, pero luego de la sentencia END. Los procedimientos externos pueden contener a su vez funciones y procedimientos internos, los que se situarán luego de una sentencia CONTAINS y antes del final del procedimiento.

2.12.2. Procedimientos Internos

Los procedimientos internos son funciones o subrutinas que se encuentran dentro del programa principal, luego de una sentencia CONTAINS. Dichas subrutinas

o funciones solo pueden ser invocadas por el programa que las contiene. Por otra parte los procedimientos internos son iguales a los procedimientos externos excepto por: Los nombres de los procedimientos internos son locales. No globales. No se puede utilizar un procedimiento interno como argumento actual al invocar a otro procedimiento.

2.12.3. Procedimientos Intrínsecos

Los procedimientos intrínsecos son funciones o subrutinas predefinidas por el lenguaje FORTRAN. Los procedimientos intrínsecos son automáticamente enlazados al programa. Entre otras cosas, los procedimientos intrínsecos realizan conversiones de datos y devuelven información sobre tipos de datos, ejecutan operaciones sobre datos numéricos y caracteres, controlan el final de los archivos y ejecutan operaciones de bit.

2.12.4. Procedimientos Modulares

Un procedimiento modular es un procedimiento declarado y definido dentro de un módulo, dentro de sentencias CONTAINS y END. Cualquier programa que incluya al módulo por medio de la sentencia USE puede acceder a los procedimiento públicos definidos en el módulo.

2.13. Declaración de Subrutinas

En FORTRAN se utiliza la sentencia SUBROUTINE para declarar a las subrutinas. A continuación se muestra la sintaxis correspondiente a la misma.

SUBROUTINE nombresub [([lista de argumentos])]
[sentencias de declaración]
[sentencias ejecutables]
END [SUBROUTINE [nombresub]]

nombresub Nombre de la subrutina, el cual puede ser global, y externo o interno a un procedimiento anfitrión. La sentencia final debe ser END [SUBROUTINE [nombresub]]

Lista de argumentos Uno o más argumentos formales, separados por comas. Un argumento formal puede ser el nombre de una variable convencional, una variable de tipo derivado, o una función intrínseca.

Si una subrutina es externa, **nombresub** es global y no puede ser utilizado por ninguna otra variable o subprograma. Si una subrutina es interna, **nombresub**

es local a la unidad del programa anfitrión. La lista de argumentos establece el número de argumentos para la subrutina. Tanto las subrutinas como las funciones pueden cambiar los valores de sus argumentos, y el programa que los invoca puede utilizar estos valores modificados.

Ejemplo de definición de subrutina simple:

```
SUBROUTINE Intercambio(a,b)
! Declaracion de argumentos
REAL a,b

! Declaracion de variables auxiliares
REAL auxi

! Cuerpo de la subrutina
  auxi=a
  a=b
  b=auxi

END SUBROUTINE Intercambio
```

Para invocar a una subrutina se utiliza la sentencia CALL. En una sentencia CALL, los argumentos actuales pasados deben concordar con los corespondientes argumentos formales en la sentencia SUBROUTINE en orden, en número, y en tipo. Si no existe una correspondencia exacta se produce un error conocido como referencia externa sin resolver. Una subrutina no devuelve directamente un valor. Sin embargo, los valores pueden ser devueltos a la unidad de programa que la llama a través de los argumentos o variables conocidas por dicha unidad. Cuando es invocada, una subrutina ejecuta el conjunto de acciones definido por sus sentencias ejecutables. La subrutina, entonces, retorna el control a la sentencia siguiente a la que la llamó. El alcance define la extensión en la cual una variable o nombre es conocido en un programa. El alcance de un nombre global es el programa completo incluyendo cualquier rutina externa.

```
PROGRAM MUESTRA
INTEGER n

  CALL sub A(n) !llamada a la subrutina
  PRINT *, n
END

SUBROUTINE sub A(a)
INTEGER a
      . . .
END SUBROUTINE
```

La variable **n** tiene como ámbito al programa principal solamente; la variable **a** tiene como ámbito a la subrutina solamente. Sin embargo, **n** y **a** se encuentran de alguna forma asociados. Esta asociación permite a los valores de **a** desde **sub_A** ser devueltos a **n** en el programa **MUESTRA**.

2.14. Creación y utilización de Módulos

A medida que nuestros programas crecen y se especializan, tener todo el código en un mismo archivo deja de ser conveniente, ya que hace mucho más difícil encontrar el lugar indicado en donde hacer una modificación, o depurar un error. Por otra parte, es común que al trabajar en un determinado dominio, se escriban varias funciones y sub-rutinas relacionadas entre sí. Si estas funciones y sub-rutinas, son invocadas por varios programas, lo mejor es agruparlas todas en un módulo, e incluir dicho módulo en los programas que las requieran.

Por ejemplo, supongamos que escribimos varias funciones y sub-rutinas para ingresar matrices, operar con ellas, calcular sus normas, intercambiar elementos, etc. Para incluirlas todas en un módulo, lo único que debemos hacer es crear un nuevo archivo, con la siguiente estructura:

[MODULE nombre del modulo]
[sección de especificación]
[sección de sub-programas internos]
END [MODULE [nombre de programa]]

Veamos un ejemplo de módulo:

```
MODULE matrixFunctions

  IMPLICIT NONE

  CONTAINS

SUBROUTINE creaMatrizIdentidad(orden, matriz)
!Crea una matriz identidad
REAL(8), ALLOCATABLE :: matriz(:,:)
INTEGER :: i, orden

IF (ALLOCATED(matriz)) THEN
  DEALLOCATE(matriz)
ENDIF

! Crea la matriz
ALLOCATE(matriz(orden,orden))
!Inicializa la matriz

matriz = 0
DO i=1, orden
   matriz(i,i) = 1
ENDDO


SUBROUTINE imprimeMatriz(a)
!Imprime la Matriz
REAL(8) a(:,:)
INTEGER i,j,cantFilas, cantCols
```

```
cantFilas = SIZE(a,DIM=1)
cantCols = SIZE(a,DIM=2)
DO i=1,cantFilas
   DO j=1,cantCols
      WRITE (*,'(F12.4)', ADVANCE='NO') a(i,j)
   ENDDO
   WRITE (*,*)
ENDDO

END MODULE
```

Para poder, utilizar las funciones del módulo es preciso compilarlo, de esta forma el compilador creará dos archivos, uno con extensión **.o** y otro con extensión **.mod**, éstos archivos se integrarán al programa principal al compilarlo y construir el programa ejecutable. Para poder utilizar las funciones y sub-rutinas del módulo que hemos creado, en el programa principal, es necesario declarar dicho módulo por medio de la sentencia **USE**.

```
PROGRAM matrices

!Modulos utilizados
USE matrixFunctions

IMPLICIT NONE

! Codigo del Programa principal
. . .
END PROGRAM
```

2.15. Utilización de Bibliotecas

Otra forma de reutilización de código, es la utilización de bibliotecas de funciones y sub-rutinas específicas. FORTRAN, es tal vez, uno de los lenguajes que posee la mayor cantidad de bibliotecas de funciones y sub-rutinas para la resolución de problemas relacionados con múltiples áreas de la matemática, la ciencia y la ingeniería. Algunas de estas bibliotecas son libres, gratuitas, de dominio público y de código abierto, otras posees licencias más restrictivas y un costo asociado, pero sea cual fuere el origen, nos permiten incorporar a nuestros programas mayor funcionalidad, necesidad de programar, agregando código existente, ya probado y de excelente calidad.

Por ejemplo, existen dos bibliotecas de código abierto muy difundidas llamadas BLAS (**B**asic **L**inear **A**lgebra **S**ubprograms) y LAPACK (**L**inear **A**lgebra **Pack**age) que tienen implementados una gran cantidad de métodos optimizados para resolver problemas de álgebra lineal. Se trata de funciones y sub-rutinas fuertemente probadas y ampliamente utilizadas en laboratorios de investigación y en

la resolución de complejos problemas de ingeniería.

En el sistema GNU/Linux que utilizaremos en el curso, estas librerías ya están instaladas, por lo que no necesitaremos más que invocar a las funciones y subrutinas directamente desde nuestros programas, ya que el compilador se encargará de buscar y enlazar el código necesario. Encontrará más información con respecto a estas librerías en: **http://www.netlib.org/lapack/** y **http://www.netlib.org/blas/**

```
PROGRAM triDiag
 ! Resuelve un sistema tri-diagonal

   IMPLICIT NONE
INTEGER, PARAMETER :: n=3
REAL(8), DIMENSION(n) :: lD, dD, uD, x
INTEGER INFO

   CALL ingVec(lD, "Diagonal Inferior     ")
   CALL ingVec(dD, "Diagonal Principal    ")
   CALL ingVec(uD, "Diagonal Superior     ")
   CALL ingVec(x, "Terminos Independientes")

   CALL DGTSV( n, 1, lD, dD, uD, x, n, INFO )

   PRINT *, "La solucion es: ", x

   CONTAINS

SUBROUTINE ingVec(vec, title)
 ! Ingresa vector
REAL(8) vec(:)
INTEGER i,n
CHARACTER*22 title

   CALL SYSTEM("clear")
   PRINT *, "Ingrese ", title

   n=SIZE(vec)
   DO i=1, n
     WRITE(*,'(A10, I2, A4)', ADVANCE='NO') "Elemento (", i, "): "
     READ (*,*) vec(i)
   ENDDO
END SUBROUTINE

END PROGRAM
```

2.16. Gráficos con Gnuplot

Frecuentemente en el Análisis Numérico es necesario graficar funciones o conjuntos de datos con la finalidad de elegir el método de resolución más adecuado o el intervalo de aplicación del mismo. **Gnuplot**, es un programa especializado en visualización de datos científicos en 2D y 3D, multiplataforma, cuya copia, distribución, ejecución y/o modificación es totalmente gratuita y legal.

2.16.1. Utilización en modo Interactivo

Para invocar al **Gnuplot** y trabajar en *modo interactivo*, simplemente abrimos una terminal y escribimos **gnuplot**. Una vez dentro del **Gnuplot** se podrán ingresar por teclado, los comandos necesarios para graficar.

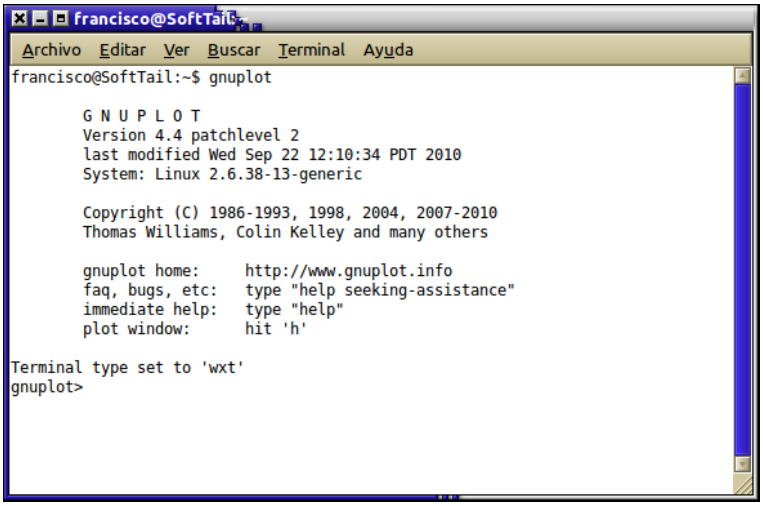

Figura 2.8: Ejecutando Gnuplot en una terminal.

La lista de comandos disponibles en **Gnuplot** puede obtenerse simplemente escribiendo la palabra **help**. Y si se desea información más detallada sobre un comando en particular, se debe escribir la palabra **help** y el comando a continuación.

```
gnuplot> help linetype
```

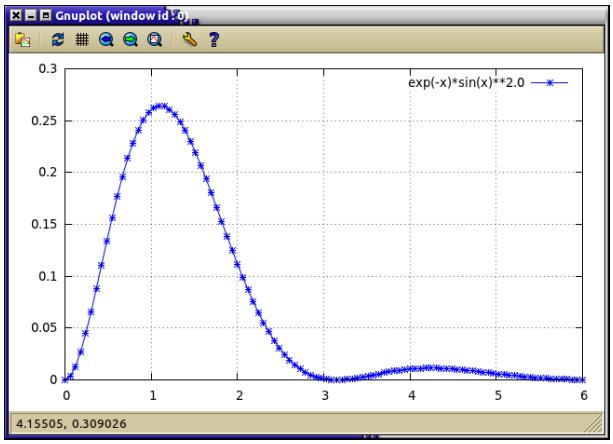

Figura 2.9: Gráfico obtenido con Gnuplot.

Para graficar una función se utiliza el comando **plot**, luego si se desea modificar ciertos aspectos del gráfico, tales como los límites, el tipo de línea, el estilo, etc. se pueden agregar los respectivos modificadores.
Por ejemplo, el gráfico de la 2.9 se obtiene a partir del siguiente conjunto de comandos **Gnuplot**.

```
gnuplot> set grid
gnuplot> set xrange [0:6]
gnuplot> plot exp(-x)*sin(x)**2.0 ls 3 with linespoints
```

Gnuplot no sólo grafica funciones, sino que también es posible graficar curvas a partir de puntos almacenados en un archivo de datos. Por ejemplo, supongamos que tenemos los valores de 200 puntos correspondientes a dos experimentos, almacenados de la siguiente forma, la primer columna corresponde a la *variable independiente*, la segunda columna corresponde al valor de la *función 1* y la tercer columna, corresponde a la *función 2*. A continuacion se muestran a modo de ejemplo algunos de esos valores.

```
  0.000000   0.000000   1.000000
  0.080000   0.079915   0.899387
  0.160000   0.159318   0.764651
  0.240000   0.237703   0.606295
  0.320000   0.314567   0.434485
  0.400000   0.389418   0.258827
  0.480000   0.461779   0.088124
  0.560000   0.531186  -0.069863
  ........   ........  .........
```

Para graficar dichos valores podemos escribir el siguiente comando.

```
plot  "valores.dat" using 1:2 title 'f1(x) = Sin(x)' with lines,\
      "valores.dat" using 1:3 title 'f2(x) = Cos(3*x)/(x+1)' with lines
```

2.16.2. Utilización en modo Batch

La utilización de **Gnuplot** en *modo interactivo* es una alternativa para obtener gráficos sencillos rápidamente, sin embargo, si se desea obtener gráficos complejos, de gran calidad, es necesario ajustar varios parámetros por lo que la cantidad de comandos aumenta notablemente. Para no tener que ingresar todos esos parámetros cada vez que se desea graficar, lo que se suele hacer es utilizar **Gnuplot** en *modo Batch*, y ejecutar varios comandos juntos almacenándolos en un *script*. Un *script* es un archivo de texto que contiene todos los comandos necesarios para generar un determinado gráfico, es decir, es como un pequeño programa cuyas instrucciones son interpretadas por el **Gnuplot**, línea por línea.

Por ejemplo, supongamos que queremos graficar los puntos almacenados en el archivo "valores.dat", como ya hicimos en el *modo interactivo*, pero ahora que-

remos mejorar nuestro gráfico, agregando información en los ejes, agregando una grilla, un título al gráfico, etc. Para no tener que escribir reiteradamente tantos comandos, simplemente los grabamos en un *script*. Otra ventaja de utilizar scripts es que podemos ejecutar un script **Gnuplot** directamente desde nuestro programa FORTRAN.

```
set autoscale
unset log
unset label
set xtic auto
set ytic auto
set grid
set title "Ejemplo Script"
set xlabel "x"
set ylabel "y"

plot  "valores.dat" using 1:2 title 'f1(x) = Sin(x)' with lines,\
      "valores.dat" using 1:3 title 'f2(x) = Cos(3*x)/(x+1)' with lines
```

Analicemos brevemente algunos de los comandos,

- **set autoscale** ajusta la escala automáticamente a los datos.
- **unset log** pasa a una escala no-logarítmica.
- **unset label** elimina las etiquetas del gráfico.
- **xtic auto**
- **set grid** agrega una grilla al gráfico.
- **set title** establece un título para el gráfico.
- **set xlabel** establece una etiqueta para el eje x.
- **plot** "valores.dat"**using** 1:2 grafica los valores que se encuentran en el archivo "valores.dat", tomando a los valores de la columna 1 como abscisas y a los de la columna 2 como ordenadas.

Encontrará más información sobre la utilización de **Gnuplot** en los siguientes enlaces:

www.gnuplot.info
www.gnuplot.vt.edu
t16web.lanl.gov/Kawano/gnuplot/index-e.html