Universidad Autónoma de San Luis Potośi.

Licenciatura en Física
de la Facultad de Ciencias.

Apuntes de "Introducción a la Programacion" usando Fortran 90.

Material didáctico.

Elaborado por Guillermo Iván Guerrero García.

1 de octubre de 2021

Índice general

Capítulo 1

Introducción al cómputo científico.

Objetivos del capítulo:

- El estudiante conocerá la evolución del uso de las computadoras y del lenguaje de programación Fortran 90 desde el siglo pasado hasta nuestros días.
- El estudiante identificará el uso de viñetas de colores verde y azul, donde se mencionan características importantes de Fotran 90 y se propoponen actividades de aprendizaje.
- El estudiante conocerá el contenido temático de los capítulos que constituyen estas notas.

En la física, varios autores han definido un triángulo en donde podemos encontrar en cada vértice a la física experimental, a la física teórica y a la física computacional. Los avances y descubrimientos en un vértice requieren la validación por parte de alguno de los otros dos vértices, promoviendo así el desarrollo de nuevos métodos y técnicas en un ciclo virtuoso. La física computacional es un área joven dentro de la física, la cual ha tenido un vertiginoso desarrollo y un impacto gigantesco en nuestra sociedad desde el siglo pasado hasta nuestros días. Aunque la forma en que se realizan cálculos numéricos ha evolucionado de forma dramática, desde el uso de bulbos y tarjetas perforadas durante la segunda guerra mundial hasta el advenimiento de computadoras con veloces procesadores de silicio en las últimas décadas, es notable observar como la lógica subyacente en la resolución numérica de ecuaciones o en métodos de simulación de muchas partículas (como el de Monte Carlo, en la dinámica molecular y/o browniana, etc.) se ha conservado en esencia hasta nuestros días. Eso nos habla de la existencia de conceptos e ideas básicas en programación que todo estudiante de ciencias debería conocer.

Uno de los lenguajes que ha sido muy utilizado en la investigación científica es el lenguaje de programación Fortran. El nombre de Fortran proviene del inglés "Formula Translation" y fue creado en los años 1950s en IBM. Posteriormente, Fortran ha ido evolucionando soportando el paradigma de programación estructurada (Fortran 77), programación modular usando vectores (Fortran 90), programación orientada a objetos (Fortran 2003), programación concurrente (Fortran 2008), incorporando incluso capacidades nativas de cómputo paralelo (Coarray Fortran 2018). El año de la versión de Fortran indica el año en que fue publicado el standard que incluye las funciones que deben estar implementadas.

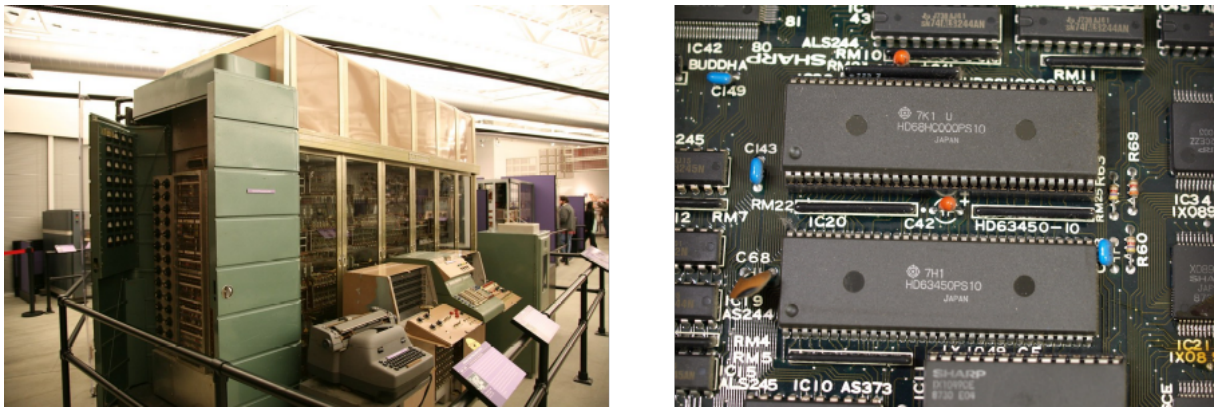

Figura 1.1: A la izquierda: computadora de mediados del siglo pasado. “Computer History Museum”, autor: Scott Beale, bajo licencia de Creative Commons BY-NC 2.0. A la derecha: Procesador de silicio de una computadora contemporánea, “Sharp X68000 Personal Computer Teardown”, autor: eevblog, bajo licencia de Creative Commons BY 2.0.

El estandard de Fortran 2003 esta completamente implementado en una gran variedad de compiladores, mientras que Fortran 2018 aún continua incorporándose en diversos compiladores libres y comerciales actualmente.

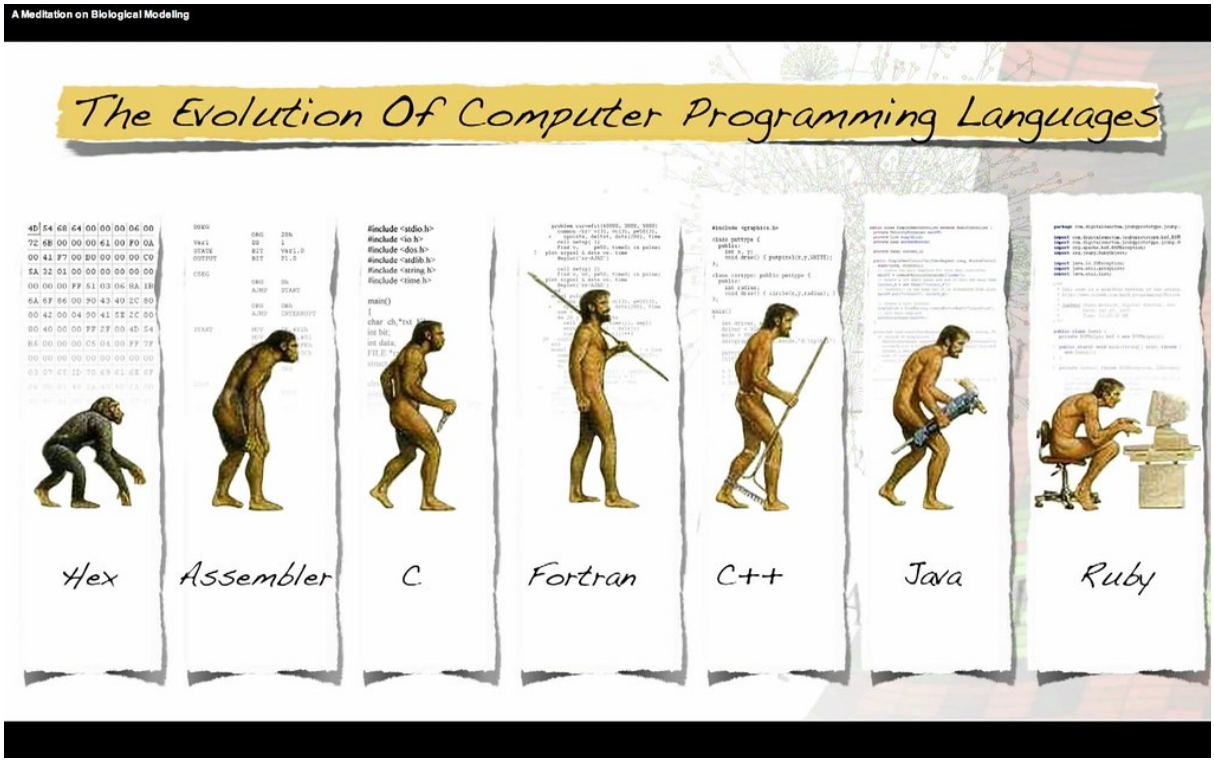

Figura 1.2: Algunos lenguajes de programación. “The Evolution of Computer Programming Languages #C #Fortran #Java #Ruby”, autor: dullhunk, bajo licencia de Creative Commons BY 2.0.

A la fecha existen numerosos libros de texto que cubren de manera detallada y minu-

ciosa las multiples características del lenguaje de programacion Fortran. Por lo anterior, el propósito de estos apuntes no es repetir de manera enciclopédica y sistemática la sintaxis de comandos en Fortran, sino proporcionar más bien una selección minimalista de códigos que ilustren importantes y útiles conceptos básicos de programación, con comentarios puntuales relacionados con su implementación en Fortran. Estos comentarios se mencionaran en viñetas en azul:

Aquí mencionamos una caraterística importante de Fortran 90.

En otras palabras, estos apuntes tienen el propósito de ser un tutorial que le permita a un estudiante de ciencias, ajeno a la programación y al lenguaje Fortran 90, poder tener los conocimientos básicos para comenzar a desarrollar sus propios códigos y poder buscar la información relevante que le permita realizar implementaciones más sofisticadas.

En los códigos de ejemplo incluidos se utilizan colores para diferenciar las palabras reservadas que denotan comandos intrínsecos de Fortran 90.

Algunas actividades propuestas para los estudiantes se mencionarán en viñetas en verde:

Aquí mencionamos una actividad que puede realizar el estudiante.

Los apuntes estan estructurados de la siguiente manera: en el capítulo 2 se discuten elementos básicos para programar en Fortran 90, que van desde la definición general de lo que es un algoritmo hasta la exposición de diferentes alternativas para instalar un compilador de Fortran 90 en una computadora personal. En el capítulo 3 se abordan las funciones básicas de fortran incluyendo la escritura y lectura de archivos. En el capítulo 4 se tratan las estructuras de decisión basadas en las instrucciones IF-ELSE y SELECT-CASE, discutiendo al final el uso de control de iteración estructurado basado en ciclos DO. En el capítulo 5 se comenta el uso de vectores. En el capítulo 6 se describe el uso de subrutinas y funciones definidas por el usuario. Finalmente, en el capítulo 7 se discuten buenas prácticas para el desarrollo de aplicaciones.

Actividades:

1. Investigue cuales son los lenguajes de programación más utilizados para resolver de manera numérica ecuaciones algebraicas, diferenciales, e integrales, y realizar simulaciones moleculares de muchas partículas.
2. Investigue cuales son las diferencias y semejanzas entre los siguientes lenguajes de programación C, C++, Python y Fortran.
3. Investigue el orden de velocidad (averigüe que lenguaje es más rápido), si se desea resolver de manera numérica ecuaciones algebraicas, diferenciales, e integrales, y realizar simulaciones moleculares de muchas partículas.

Bibliografía

1. Mark Jones Lorenzo, *Abstracting Away the Machine: The History of the FORTRAN*

Programming Language (FORmula TRANslation), SE Books, Philadelphia, Pittsburgh, 2019.

2. Sitio web donde se encuentra información sobre el estado del arte en Fortran. https://fortran-lang.org/

3. Michael Metcalf y John Reid, *Fortran 90/95 explained*, Second edition, Oxford University Press, 1999.

4. Stephen J. Chapman, *Fortran 90/95 for Scientists and Engineers*, First edition, McGraw-Hill, 1997.

Capítulo 2

Elementos básicos para programar en Fortran 90.

Objetivos del capítulo:

- El estudiante conocerá lo que es un algortimo general de programación.
- El estudiante conocerá los principales sistemas operativos y compiladores de Fortran 90 gratuitos para estos sistemas.
- El estudiante conocerá los elementos básicos que le permitan instalar una máquina virtual y el sistema operativo Linux.
- El estudiante conocerá elementos básicos que le permitan instalar los compiladores gratuitos de Fortran 90 de Silver Frost y gfortran en los sistemas operativos Windows, Linux y MacOS.

2.1. Algoritmos

Todos sabemos como realizar tareas, como preparar el desayuno, cruzar la calle, ir al supermercado, hacer limpieza en casa, etc. Muchas de estas tareas las aprendimos a prueba y error, y pudieron incluir la intervención de alguna persona que nos enseño–mediante instrucciones y ejemplos–la forma en que debían realizarse dichas tareas. En este sentido, un algoritmo puede considerarse de manera coloquial como una *receta*, o un conjunto de pasos a seguir, para realizar una tarea arbitraria. Cuando nosotros separamos en pasos la manera en que se realiza una tarea, aquirimos una mayor conciencia de lo que se requiere para poder realizar dicha tarea. Ese es el primer paso que debemos considerar al escribir un programa, sin importar el lenguaje de programación. Dicho programa le dirá a la computadora que operaciones debe realizar.

Una parte fundamental de todo algoritmo es el uso adecuado de decisiones lógicas que permitan su correcta implementación y funcionamiento. En este sentido, existen dos maneras ampliamente utilizadas para visualizar un algoritmo. Uno es mediante el uso de diagramas de flujo y el otro es a través del uso de pseudo-códigos. Los diagramas de flujo utilizan figuras geométricas para representar operaciones diversas que se realizan en un programa como el inicio y fin del programa, lectura o escritura de datos, toma de decisiones, realización de iteraciones, etc. Esta alternativa es muy visual. Por otra parte, un pseudo-codigo

es mas parecido a una receta de cocina donde se enuncian los pasos que deben seguirse usando un lenguaje coloquial. Uno de los propósitos principales de un pseudo-código es que sea lo suficientemente general para que pueda ser implementado, independientemente del lenguaje de programación elegido. El pseudo-código de programas sencillos es muy parecido a la sintaxis de un programa en Fortran 90, donde se hace uso extensivo del idioma inglés en la definición de los nombres de los comandos o funciones intrínsecas de este lenguaje de programción.

Una vez que se ha definido un diagrama de flujo o pseudo-código de un algoritmo, es muy deseable encapsular cada uno de estos pasos en una caja negra que tenga variables de entrada y variables de salida. A este enfoque se le conoce como programación estructurada. Como veremos más adelante, las subrutinas constituyen la base de la implementación de códigos modulares basados en procedimentos. Una ventaja importante de este enfoque modular es que permite aislar la depuración de cada parte de un programa, lo cual es muy útil para detectar y corregir los errores lógicos y de sintaxis que se presenten. Una vez que una subrutina ha sido validada, puede re-usarse con mayor confianza en otro programa o subrutina.

2.2. Compiladores.

Un sistema operativo es un programa o *software* que permite al usuario poder interaccionar y controlar dispositivos perifericos de una computadora como el monitor, el teclado, el ratón, etc., así como comunicarse con otros programas. Actualmente, el sistema operativo de computadoras personales más utilizado por cuestiones comerciales es Windows de Microsoft. En segundo lugar esta el sistema operativo de Apple, MacOS. En último lugar estan los sistemas operativos basados en linux. Windows es un sistema operativo ineficiente que además tiene la necesidad de consumir recursos importantes de la computadora para correr programas adicionales como los antivirus, lo cual lo hace aún más ineficiente. MacOS es un sistema operativo propietario basado en el sistema operativo UNIX, el cual fue desarrollado e implementado en 1969 en los laboratorios Bell de ATT en los Estados Unidos por Ken Thompson, Dennis Ritchie, Douglas McIlroy, y Joe Ossanna. Linux es el primer sistema operativo libre implementado en los 1990s principalmente por Linus Torvalds. La resolución de ecuaciones numéricas y simulaciones de sistemas de muchas partículas que se realizan en centros de supercómputo a nivel mundial se realiza en servidores basados esencialmente en Linux y en menor medida en Unix.

Aquellos usuarios que sólo tengan acceso a una computadora personal con sistema operativo Windows pueden instalar el compilador de Silver Frost:

https://www.silverfrost.com/11/ftn95_overview.aspx

descargando la version personal gratuita para propositos de evaluación:

https://www.silverfrost.com/32/ftn95/ftn95_personal_edition.aspx

Este compilador de Fortran 90 y Fortran 95 incluye el depurador de programas Plato.

Los usuarios de Linux o MacOS pueden instalar el compilador gfortran desde una terminal remota. En este caso, es necesario usar un editor de textos que no introduzca formato (plain text editor) para editar el código fuente de un programa en Fortran 90 que será

compilado. Algunos ejemplos de este tipo de editores son xemacs, emacs, vim, etc. en linux, Open Text Edit en MacOS, y notepad en Windows.

Una alternativa para que los usuarios de Windows experimenten el uso del sistema operativo Linux es mediante el uso de una maquina virtual. Este programa es un emulador de una computadora física que corre sobre un sistema operativo específico, de tal manera que es posible instalar un sistema operativo que corre sobre otro sistema operativo. Así, por ejemplo, es posible disponer de un sistema operativo Linux virtual corriendo sobre un sistema operativo Windows nativo o viceversa. Una opción específica de una maquina virtual es virtualbox de Oracle:

https://www.virtualbox.org/

Linux tiene diferentes *sabores* o implementaciones. Algunos de los más conocidos son Fedora, Ubuntu, Debian, Linux Mint, etc. Una vez que la maquina virtual ha sido instalada es necesario descargar el archivo iso correspondiente a la versión de linux que desea instalarse. Algunas versiones muy ligeras son las versiones lxde y xcfe:

https://spins.fedoraproject.org/lxde/download/index.html

https://www.xfce.org/download

Una ventaja de estas versiones es que son más ligeras al correr sobre otro sistema operativo como Windows, requiriendo menos recursos para funcionar correctamente.

En el caso de Silver Frost, existe un tutorial de Fortran 90 usando Plato:

https://www.fortrantutorial.com/

Con respecto a Linux o Unix, existen diferentes compiladores gratuitos y comerciales que pueden instalarse.

En Fedora, la instalación de gfortran puede realizarse mediante el uso de la siguiente instrucción como superusuario:

[localhost]# dnf install gcc-gfortran

En Ubuntu, la instalación de gfortran puede realizarse mediante el uso de la siguiente instrucción como superusuario:

[localhost]# apt-get install gfortran

Para correr un programa usando gfortran desde una terminal o consola solo basta usar:

[localhost]$ gfortran fuente.f90

lo cual generará un ejecutable a.out, el cual se corre como:

[localhost]$ ./a.out

Es posible generar un ejecutable con un nombre específico con la siguiente opción:

[localhost]$ gfortran -o ejecutable.exe fuente.f90

el cual se corre como:

[localhost]$ ./ejecutable.exe

Actividades:

1. Investigue cual es la diferencia entre el paradigma de programación estructurada y la programación orientada a objetos.
2. Investigue cuales son los programas equivalentes a los que usualmente se utilizan en Windows para editar textos y presentaciones, reproducir sonidos, visualizar videos, etc., en la plataforma Linux y/o MacOS. Por ejemplo, en Linux es posible utilizar Libre Office en lugar de Microsoft Office, etc.
3. Investigue que tipo de programas o software no existen en Linux y/o MacOS, que son de uso científico y que si existen en Windows.
4. Investigue cual es la diferencia entre Fedora y Centos.
5. Investigue cual es la diferencia entre Debian y Ubuntu.
6. Investigue que otros compiladores gratuitos, además de gfortran y del de Silver Frost, es posible encontrar para Linux y MacOs, incluyendo su grado de confiabilidad.

Bibliografía

1. Arun Kumar, *Oracle VirtualBox Administration: A beginners guide to virtualization!*, Independently published, 2019.
2. James Bernstein, *VirtualBox Made Easy: Virtualize Your Environment with Ease (Computers Made Easy)*, Independently published, 2020.
3. Sitio web donde se encuentra información sobre como compilar gfortran en diferentes plataformas. https://fortran-lang.org/learn/os_setup/install_gfortran
4. Wale Soyinka, *Linux Administration: A Beginner's Guide, Eighth Edition*, McGraw-Hill Education, 2020.

Capítulo 3

Funciones intrínsecas.

Objetivos del capítulo:

- El estudiante conocerá la estructura básica de un programa simple de Fortran 90.
- El estudiante conocerá diversos comandos o funciones intrínsecas de Fortran 90, incluyendo funciones matemáticas y de formato.
- El estudiante conocerá como se definen números reales y enteros de doble precisión.

Uno de los programas más simples en Fortran 90 es el siguiente:

```
PROGRAM saludos

PRINT *,'Saludos desde San Luis Potosi'

END PROGRAM
```

Este programa imprime la frase "Saludos desde San Luis Potosi" en pantalla usando el comando PRINT. EL asterisco es necesario para poder imprimir la cadena de caracteres en pantalla.

Un programa que realiza la suma de dos números es el siguiente:

```
PROGRAM suma

!  Proposito:
!  Ilustrar algunas caracteristicas de Fortran 90.
!

! Declaracion de variables
INTEGER :: a, b, c             ! Declaracion de variables
   enteras

! Obtencion de los valores de entrada
PRINT *,'Este programa suma dos numeros'
```

```
WRITE (*,*) 'Ingrese: a'
READ (*,*) a
WRITE (*,*) 'Ingrese: b'
READ (*,*) b


! Suma de los numeros
c = a + b

! Escribe el resultado
WRITE (*,*) 'Resultado = ', c

END PROGRAM
```

En este programa,

- Las variables a, b, c se definen como números enteros.
- El signo de admiración se utiliza para escribir comentarios que documenten el programa, pero que no realizan ninguna función.
- El comando WRITE se utiliza para escribir una cadena de caracteres y una de las variables enteras.
- El comando READ se utiliza para leer dos variables enteras.

Note que a diferencia del comando PRINT, los comandos WRITE y READ usan dos asteriscos (*,*). El primer asterisco se usa para indicar que la entrada proviene del teclado (al modificar el comando WRITE) y la salida va direccionada a la pantalla (al modificar al comando READ). El segundo asterisco indica que no hay un formato específico en la escritura o lectura de los datos usando WRITE o READ, respectivamente. Ejemplos del uso de otros modificadores en las posiciones de los asteriscos se verá más adelante.

Fortran 90 no hace distinción entre mayúsculas y minúsculas. Es decir, los siguientes comandos son equivalentes:

- PRINT
- Print
- print

Lo mismo ocurre con las siguientes variables:

- DATO
- Dato
- dato

En el siguiente ejemplo se realiza la conversión de grados Fahrenheit a Kelvin:

```
PROGRAM conversion

!  Proposito:
!    Convertir una temperatura en grados Fahrenheit a
!    grados Kelvin.

IMPLICIT NONE   ! Le indica al compilador que todas las
   variables
! deben estar declaradas con algun tipo de dato

! Declaracion de las variables donde se almacenaran las
   temperaturas
REAL :: temp_f       ! Temperatura en grados Fahrenheit
REAL :: temp_k       ! Temperatura en grados Kelvin

! Se pide que se ingrese el valor de la temperatura en grados
   Fahrenheit
WRITE (*,*) 'Ingrese el valor de la temperatura en grados
   Fahrenheit: '
READ  (*,*) temp_f

! Conversion a grados Kelvin
temp_k = (5. / 9.) * (temp_f - 32.) + 273.15

! Se escribe el resultado en pantalla
WRITE (*,*) temp_f, ' grados Fahrenheit = ', temp_k, ' Kelvin'

END PROGRAM
```

En este programa la instrucción IMPLICIT NONE se utiliza para indicar al compilador que debe verificar que todas las variables hayan sido declaradas con un tipo específico.

En Fortran 90 existen los siguientes tipos de variables:

- INTEGER (variable de tipo entero)
- REAL (variable de tipo real)
- COMPLEX (variable de tipo complejo con dos componentes)
- LOGICAL (variable de tipo lógico o booleano)
- CHARACTER (variable de tipo cadena o caracter)

El valor máximo de un número entero, real o complejo depende del numero de bytes con que sea definido.

En general, para cálculos numéricos se recomiendo usar precisión doble para evitar errores de truncamiento o redondeo. Esto puede lograrse usando la sintaxis REAL*8.

Si se emplea la sintaxis INTEGER*8, es posible definir un entero de 8 bytes.

Existen formas más sofisticadas para definir el valor máximo que un número puede tomar en función del número de bits con sea definido usando el comando KIND, lo cual esta fuera del alcance de estas notas, pero puede consultarse en el material citado en la bibliografía.

A continuación se proporcionan algunos ejemplos del uso de número reales de doble precisión y enteros grandes:

```
PROGRAM enteros
IMPLICIT NONE

!Entero de 2 bytes
INTEGER*2 :: b2

!Entero de 4 bytes
INTEGER*4 :: b4

!Entero de 8 bytes
INTEGER*8 :: b8

!default integer
INTEGER :: default

PRINT *, HUGE(b2)
PRINT *, HUGE(b4)
PRINT *, HUGE(b8)
PRINT *, HUGE(default)

END PROGRAM enteros
```

Salida del programa:

```
[localhost]$ ./a.out
32767
2147483647
9223372036854775807
2147483647
```

```
PROGRAM reales
IMPLICIT NONE
```

```
!Entero de 4 bytes
REAL*4 :: b4

!Entero de 8 bytes
REAL*8 :: b8

!default integer
REAL :: default

PRINT *, HUGE(b4)
PRINT *, HUGE(b8)
PRINT *, HUGE(default)

END PROGRAM reales
```

Salida del programa:

[localhost]$ ./a.out
3.40282347E+38
1.7976931348623157E+308
3.40282347E+38

Fortran cuenta con muchas funciones matemáticas intrinsecas, es decir, funciones que ya estan implementadas y que solo es necesario llamar. Algunos ejemplos de estas funciones son:

- ABS(x) (valor absoluto)
- SQRT(x) (raíz cuadrada)
- SIN(x) (función seno)
- COS(x) (función coseno)
- TAN(x) (función tangente)
- ASIN(x) (función inversa del seno)
- ACOS(x) (función inversa del coseno)
- ATAN(x) (función inversa de la tangente)
- EXP(x) (función exponencial)
- LOG(x) (función logaritmo natural)

En Fortran 90, para elevar un numero a una potencia es necesario usar el operador doble asterisco **. Por ejemplo, si queremos calcular la raíz cuadrada de un número es posible usar:

REAL :: numero = 16.0

PRINT *, SQRT(numero)

o

PRINT *, numero**(0.5)

Note que en este caso se inicializó la variable "numero" desde su definición. También es importante considerar que es una buena práctica definir siempre los exponentes como numeros reales para evitar problemas de redondeo al usar numero enteros.

En el siguiente programa se reemplaza el segundo asterisco en el parentesis (*,*) por el número entero "etiqueta", de tal manera que se escribe (*,etiqueta). La variable "etiqueta" indica el formato de las variables que escribe el comando WRITE mediante el comando FORMAT.

```
PROGRAM tabla
!
!  Proposito
!    Ilustrar el uso de formato de salida usando WRITE
!
IMPLICIT NONE


INTEGER :: i=4          ! Variable i
INTEGER :: cuadrado       ! Cuadrado de i
REAL    :: raiz  ! Raiz cuadrada de i
INTEGER :: cubo         ! El cubo de i

! Escriba el titulo de la tabla
WRITE (*,100)
100 FORMAT ('12',T3, 'Tabla de raices cuadradas, cuadrados, y
   cubos')

! Escribir el encabezado
WRITE (*,110)
110 FORMAT ('123','Numero',T13,'Raiz
   cuadrada',T29,'Cuadrado',T39,'Cubo')
WRITE (*,120)
120 FORMAT
   (1X,T4,'======',T13,'===========',T29,'======',T39,'====')

! Generar los valores de salida
   raiz = SQRT ( REAL(i) )
   cuadrado = REAL(i)**2.0
   cubo = REAL(i)**3.0
   WRITE (*,130) i, raiz, cuadrado, cubo
   130 FORMAT (T4, I4, T13, F10.6, T27, I6, T37, I6)
```

```
END PROGRAM
```

En el ejemplo anterior, "Ti" quiere decir que la siguiente variable se situa en la columna "i". Puede usarse tambien "Xi", pero en este caso "i" indica el numero de espacios que habrá después de la posición de la variable anterior. Note el uso del comando REAL para convertir una variable entera a una variale real para evitar errores de redondeo o truncamiento.

Salida del programa:

```
[localhost]$ ./a.out
12Tabla de raices cuadradas, cuadrados, y cubos
123Numero Raiz cuadrada Cuadrado Cubo
====== =========== ====== ====

4 2.000000 16 64
```

Actividades:

1. Investigue que es un error de truncamiento.
2. Investigue como se ven afectados los errores de truncamiento usando variables de tipo REAL y REAL*8.
3. Investigue el uso de la instrucción KIND para determinar el tamaño o número de bytes asociados a una variable real o entera, así como para establecer la precisión de estas variables en Fortran 90.

Bibliografía

1. Michael Metcalf y John Reid, *Fortran 90/95 explained*, Second edition, Oxford University Press, 1999.
2. Stephen J. Chapman, *Fortran 90/95 for Scientists and Engineers*, First edition, McGraw-Hill, 1997.

Capítulo 4

Estructuras de decisión y control de iteración estructurado

Objetivos del capítulo:

- El estudiante conocerá las estructuras de decisión IF-ELSE y SELECT-CASE.
- El estudiante conocerá la estructura de control iterativo basada en el ciclo DO con iterador
- El estudiante conocerá como imprimir tablas usando ciclos DO y modificadores de formato.

Una de las instrucciones más simples para tomar decisiones en un programa de Fortran 90 es mediante el comando IF-ELSE. Por otra parte, un ciclo es una instrucción que repite un conjunto de instrucciones hasta que se satisface una condicón de término. Esto se ilustra mediante el siguiente programa.

```
PROGRAM media_y_desviacion_estandar
!
!  Proposito: calcular la media y desviacion estandar
!  de un conjunto de numeros arbitrario
IMPLICIT NONE
! Declaracion de las variables
INTEGER :: n = 0      ! Numero de datos.
REAL :: std_dev = 0. ! Desviacion estandar de los datos de
   entrada.
REAL :: sum_x = 0.    ! Suma de los datos de entrada.
REAL :: sum_x2 = 0.   ! Suma del cuadrado de los datos de
   entrada.
REAL :: x = 0.        ! Valor de entrada.
REAL :: x_bar         ! Valor medio de los datos de entrada.

!  Realizacion de un ciclo DO mientras haya datos que ingresar
```

```
PRINT *,'Cuando ya no desee ingresar mas datos, escriba un
   numero negativo'
DO
   ! Ingresar datos
   WRITE (*,*) 'Ingrese un numero: '
   READ  (*,*) x
   WRITE (*,*) 'El numero ingresado es', x

   ! Verificacion de la condicion de termino del ciclo
   IF ( x < 0 ) EXIT

   ! Si el numero no es negativo se continua con la acumulacion
   n      = n + 1
   sum_x  = sum_x + x
   sum_x2 = sum_x2 + x**2
END DO

!  Verificacion de que hay suficientes datos
IF ( n < 2 ) THEN ! No hay el minimo numero de datos necesarios

   WRITE (*,*) 'Error: al menos se requiere ingresar dos datos'

ELSE ! Si hay suficientes datos por lo que se procede al
! calculo de la media y la desviacion estandar

   x_bar = sum_x / REAL(n)
   std_dev = SQRT( (REAL(n) * sum_x2 - sum_x**2) /
      (REAL(n)*REAL(n-1)))

   ! Reportar los resultados
   WRITE (*,*) 'La media del conjunto de datos es:', x_bar
   WRITE (*,*) 'La desviacion estandar es:   ', std_dev
   WRITE (*,*) 'El numero de datos ingresados es:', n

END IF

END PROGRAM
```

Salida del programa:

[localhost]$ ./a.out
Cuando ya no desee ingresar mas datos, escriba un numero negativo
Ingrese un numero:
10
El numero ingresado es 10.0000000
Ingrese un numero:
20

El numero ingresado es 20.0000000
Ingrese un numero:
30
El numero ingresado es 30.0000000
Ingrese un numero:
40
El numero ingresado es 40.0000000
Ingrese un numero:
50
El numero ingresado es 50.0000000
Ingrese un numero:
60
El numero ingresado es 60.0000000
Ingrese un numero:
70
El numero ingresado es 70.0000000
Ingrese un numero:
80
El numero ingresado es 80.0000000
Ingrese un numero:
90
El numero ingresado es 90.0000000
Ingrese un numero:
100
El numero ingresado es 100.000000
Ingrese un numero:
-2
El numero ingresado es -2.00000000
La media del conjunto de datos es: 55.0000000
La desviacion estandar es: 30.2765045
El numero de datos ingresados es: 10

Una secuencia de comandos IF puede usarse para generar IFs anidados:

IF condicion1 THEN
operacion1
ELSE IF condicion2 THEN
operacion2
ELSE IF condicion3 THEN
operacion3
ELSE
operacion
END IF

Una alternativa para IFs anidados es la utilización de la instrucción SELECT CASE usando la siguiente sintaxis:

SELECT CASE (numero)
CASE (1)

operacion1
CASE (2)
operacion2
CASE (3)
operacion3
CASE default
operacion
END SELECT

Esto se ilustra en el siguiente programa.

```
PROGRAM equivalencia_if_anidados_select_case
  IMPLICIT NONE

  INTEGER :: trimestre

  PRINT *,'Ingrese el numero de un trimestre entre 1 y 4'
  READ (*,*) trimestre

  IF (trimestre==1) THEN
     PRINT *,'Escogio el primer trimestre usando IFs anidados'
  ELSE IF (trimestre==2) THEN
     PRINT *,'Escogio el segundo  trimestre usando IFs
        anidados'
  ELSE IF (trimestre==3) THEN
     PRINT *,'Escogio el tercer trimestre usando IFs anidados'
  ELSE IF (trimestre==4) THEN
     PRINT *,'Escogio el cuarto trimestre usando IFs anidados'
  ELSE
     PRINT *,'El trimestre escogido no existe usando IFs
        anidados'
  END IF

   SELECT CASE (trimestre)
   CASE (1)
     PRINT *,'Escogio el primer trimestre usando SELECT CASE'
   CASE (2)
     PRINT *,'Escogio el segundo trimestre usando SELECT CASE'
   CASE (3)
     PRINT *,'Escogio el tercer trimestre usando SELECT CASE'
   CASE (4)
     PRINT *,'Escogio el cuarto trimestre usando SELECT CASE'
   CASE default
      PRINT *,'El trimestre escogido no existe usando SELECT
         CASE'
   END SELECT
```

```
END PROGRAM
```

Salida del programa:

```
[localhost]$ ./a.out
Ingrese el numero de un trimestre entre 1 y 4
1
Escogio el primer trimestre usando IFs anidados
Escogio el primer trimestre usando SELECT CASE

[localhost]$ ./a.out
Ingrese el numero de un trimestre entre 1 y 4
2
Escogio el segundo trimestre usando IFs anidados
Escogio el segundo trimestre usando SELECT CASE

[localhost]$ ./a.out
Ingrese el numero de un trimestre entre 1 y 4
3
Escogio el tercer trimestre usando IFs anidados
Escogio el tercer trimestre usando SELECT CASE

[localhost]$ ./a.out
Ingrese el numero de un trimestre entre 1 y 4
4
Escogio el cuarto trimestre usando IFs anidados
Escogio el cuarto trimestre usando SELECT CASE

[localhost]$ ./a.out
Ingrese el numero de un trimestre entre 1 y 4
5
El trimestre escogido no existe usando IFs anidados
El trimestre escogido no existe usando SELECT CASE

[localhost]$ ./a.out
Ingrese el numero de un trimestre entre 1 y 4
0
El trimestre escogido no existe usando IFs anidados
El trimestre escogido no existe usando SELECT CASE
```

También es posible usar una variable contador para controlar el número de iteraciones de un ciclo DO como se ilustra en el siguiente programa.

```
PROGRAM tabla2
!
!  Proposito
!    Ilustrar el uso de un contador en un ciclo DO
!
IMPLICIT NONE


INTEGER :: i          ! Variable i
INTEGER :: cuadrado   ! Cuadrado de i
REAL    :: raiz       ! Raiz cuadrada de i
INTEGER :: cubo       ! El cubo de i

! Escriba el titulo de la tabla
WRITE (*,100)
100 FORMAT ('12',T3, 'Tabla de raices cuadradas, cuadrados, y
   cubos')

! Escribir el encabezado
WRITE (*,110)
110 FORMAT ('123','Numero',T13,'Raiz
   cuadrada',T29,'Cuadrado',T39,'Cubo')
WRITE (*,120)
120 FORMAT
   (1X,T4,'======',T13,'===========',T29,'======',T39,'====')

! Generar los valores de salida en un ciclo DO
DO i=1,10
   raiz = SQRT ( REAL(i) )
   cuadrado = REAL(i)**2.0
   cubo = REAL(i)**3.0
   WRITE (*,130) i, raiz, cuadrado, cubo
   130 FORMAT (T4, I4, T13, F10.6, T27, I6, T37, I6)
END DO

END PROGRAM
```

Salida del programa:

```
[localhost]$ ./a.out
12Tabla de raices cuadradas, cuadrados, y cubos
123Numero Raiz cuadrada Cuadrado Cubo
====== =========== ====== ====

1 1.000000 1 1
2 1.414214 4 8
3 1.732051 9 27
```

```
4 2.000000 16 64
5 2.236068 25 125
6 2.449490 36 216
7 2.645751 49 343
8 2.828427 64 512
9 3.000000 81 729
10 3.162278 100 1000
```

Actividades:

1. Investigue el uso y sintaxis de la instrucción DO WHILE en Fortran 90.
2. Investigue que es un ciclo infinito, y determine si este puede aparecer en un ciclo DO con iterador (como el que se vio en el presente capítulo), así como en un ciclo DO WHILE.
3. Investigue si existe algún escenario en donde un ciclo DO WHILE sea superior a un ciclo DO con iterador (como el que se vio en este capítulo).

Bibliografía

1. Michael Metcalf y John Reid, *Fortran 90/95 explained*, Second edition, Oxford University Press, 1999.
2. Stephen J. Chapman, *Fortran 90/95 for Scientists and Engineers*, First edition, McGraw-Hill, 1997.

Capítulo 5

Apertura de archivos para lectura y escritura.

Objetivos del capítulo:

- El estudiante conocerá diferentes maneras de utilizar archivos para leer y escribir datos.
- El estudiante conocera el uso de una variable de estado que le permita conocer si la apertura de un archivo y la lectura de datos fue exitosa o no.

Para abrir un archivo en Fortran 90 se utiliza el comando OPEN. Este comando tiene la siguiente sintaxis:

OPEN (UNIT=num, FILE='arch', STATUS='char1', ACTION='char2', IOSTAT=edo)

La opción "UNIT" asigna el archivo que va abrirse a la unidad de disco "num", donde "num" es un número entero o es una variable de tipo entero inicializada. La opción "FILE" registra que el nombre del archivo es 'arch'. También es posible usar una variable de tipo caracter que contenga el nombre del archivo de texto a leer.

La opción "STATUS" registra una cadena de caracteres 'char1', donde char1 puede tomar los siguientes nombres:

- NEW (se usa para crear un archivo nuevo que no existe en el directorio donde se ubica el ejecutable)
- OLD (se usa para leer un archivo que ya existe)
- REPLACE (se usa para crear un archivo que ya existe en el directorio donde se ubica el ejecutable sobre-escribiéndolo)
- SCRATCH (se usa para crear archivos temporales)
- UNKNOWN (se utiliza cuando el argumento de estado IOSTAT no esta presente)

La opción "ACTION" registra la acción que se va a realizar sobre el archivo que se abre mediante una cadena de caracteres 'char2', donde char2 puede tomar los siguientes nombres:

- READ (el archivo sólo se abre con permisos de lectura)
- WRITE (el archivo sólo se abre con permisos de escritura)
- READWRITE (el archivo se abre con permisos de lectura y escritura simultáneamente)

Una vez que el archivo ha dejado de usarse es importante cerrarlo con el comando CLOSE(num), donde 'num' es el número entero correspondiente al número de unidad de disco que se abrió.

La opción IOSTAT inicializa la variable entera "edo". Si el valor de la variable "edo" es igual a cero, el archivo se abrió exitosamente. De otro modo, hubo un error de lectura. Este puede haber ocurrido porque el archivo que se quiere abrir no existe en el directorio donde se encuentra el ejecutable o porque el archivo esta dañado.

En el siguiente programa se ilustra el uso de los comandos OPEN y CLOSE para leer un conjunto de datos de un archivo de texto cuyo número es desconocido.

```
PROGRAM lectura
!
!  Proposito: leer datos de un archivo de lectura.
!
IMPLICIT NONE

! Declaracion de variables
CHARACTER(len=20) :: archivo      ! Nombre del archivo que se
   va a abrir
INTEGER :: nvals = 0              ! Numero de valores leidos
INTEGER :: estado                 ! Estado de la lectura
REAL :: valor                     ! Valor leido

! Introducir el nombre del archivo
WRITE (*,*) 'Ingrese el nombre del archivo de menos de 20
   caracteres: '
READ  (*,*) archivo
WRITE (*,1000) archivo
1000 FORMAT (' ','El nombre del archivo a leer es: ', a20)

! Apertura del archivo de lectura
OPEN (UNIT=10, FILE=archivo, STATUS='OLD', ACTION='READ', &
      IOSTAT=estado )

IF ( estado == 0 ) THEN

! El archivo se abrio exitosamente, se procede a leer los datos
   DO
      READ (10,*,IOSTAT=estado) valor   ! Leer dato y ponerlo
         en la variable valor
      IF ( estado /= 0 ) THEN
         EXIT           ! Si el estado es diferente de cero
```

```
            salir del ciclo DO
      ELSE
      nvals = nvals + 1                  ! Si el estado es cero
         incrementar el contador
      WRITE (*,1010) nvals, valor        ! Imprimir el pantalla
         el contador y el dato leido
      1010 FORMAT (' ','Linea ', I6, ': Valor = ',F10.4 )
      END IF
   END DO
ELSE
   WRITE (*,1040) estado ! Reportar que hubo un error al leer
      el archivo
   1040 FORMAT (' ','Hubo un error al abrir el archivo: IOSTAT
      = ', I6 )
END IF

! Cerrar archivo
CLOSE ( UNIT=10 )

END PROGRAM
```

donde el archivo "entrada.dat" contiene los siguientes datos:

1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0

Salida del programa:

[localhost]$ ./a.out
Ingrese el nombre del archivo de menos de 20 caracteres:
entrada.dat
El nombre del archivo a leer es: entrada.dat
Linea 1: Valor = 1.0000
Linea 2: Valor = 2.0000
Linea 3: Valor = 3.0000
Linea 4: Valor = 4.0000
Linea 5: Valor = 5.0000
Linea 6: Valor = 6.0000

Linea 7: Valor = 7.0000
Linea 8: Valor = 8.0000
Linea 9: Valor = 9.0000
Linea 10: Valor = 10.0000

[localhost]$ ./a.out
Ingrese el nombre del archivo de menos de 20 caracteres:
entrada
El nombre del archivo a leer es: entrada
Hubo un error al abrir el archivo: IOSTAT = 2

En particular, nótese como se usa el valor de la variable "edo" para determinar si el archivo se pudo abrir exitosamente con el comando OPEN.

También es importante el uso de un parentesis con 3 argumentos para el comando READ (10,*,IOSTAT=edo). El número 10 indica el número de unidad de disco a la que fue asignada el archivo que se va a leer. En este caso la variable "edo" se usa para saber que todavía no se ha llegado al final del archivo.

En el caso del comando WRITE (arg1,arg2), arg1 indica el numero de unidad asociado al nombre del archivo usado en la opcion FILE del comando OPEN, mientras que arg2 puede usarse para darle un formato a las variables que habrán de escribirse en el archivo de texto de salida. El siguiente programa, que es una modificacion del programa anterior, ilustra esta característica.

```
PROGRAM lectura_escritura
!
!  Proposito: leer datos de un archivo de lectura y reportar
   la suma en
!  un archivo de salida.
IMPLICIT NONE

! Declaracion de variables
CHARACTER(len=20) :: archivo='entrada.dat' ! Nombre del
   archivo a leer
CHARACTER(len=20) :: salida='salida.dat' ! Nombre del archivo
   a escribir
INTEGER :: nvals = 0              ! Numero de valores leidos
INTEGER :: estado                 ! Estado de la lectura
REAL :: valor=0.0                 ! Valor leido
REAL :: suma=0.0                  ! Acumulador

! Apertura del archivo de lectura
OPEN (UNIT=10, FILE=archivo, STATUS='OLD', ACTION='READ', &
      IOSTAT=estado )

OPEN (UNIT=20, FILE=salida, STATUS='REPLACE', ACTION='WRITE')
```

```
IF ( estado == 0 ) THEN

! El archivo se abrio exitosamente, se procede a leer los datos
   DO
      READ (10,*,IOSTAT=estado) valor   ! Leer dato y ponerlo
         en la variable valor
      IF ( estado /= 0 ) THEN
         EXIT           ! Si el estado es diferente de cero
            salir del ciclo DO
      ELSE
      nvals = nvals + 1 ! Si el estado es cero incrementar el
         contador
      suma = suma + valor
      WRITE (*,1010) nvals, valor ! Imprimir en pantalla
      1010 FORMAT (' ','Linea ', I6, ': Valor = ',F10.4 )
      END IF
   END DO
ELSE
   WRITE (*,1040) estado ! Reportar que hubo un error al leer
      el archivo
   1040 FORMAT (' ','Hubo un error al abrir el archivo: IOSTAT
      = ', I6 )
END IF

WRITE (20,*) 'La suma de los datos leidos es: ',suma

! Cerrar archivo
CLOSE ( UNIT=10 )
CLOSE ( UNIT=20 )

END PROGRAM
```

La salida del archivo 'salida.dat' se obtiene con el comando "more" en linux :

[localhost]$ ./a.out
Linea 1: Valor = 1.0000
Linea 2: Valor = 2.0000
Linea 3: Valor = 3.0000
Linea 4: Valor = 4.0000
Linea 5: Valor = 5.0000
Linea 6: Valor = 6.0000
Linea 7: Valor = 7.0000
Linea 8: Valor = 8.0000
Linea 9: Valor = 9.0000
Linea 10: Valor = 10.0000

[localhost]$ more salida.dat
La suma de los datos leidos es: 55.0000000

Actividades:

1. Determine cuales son las ventajas o desventajas al inicializar los coeficientes o variables numéricas que requiere un programa mediante el uso de un archivo de lectura, en relación a ingresarlos interactivamente mediante el teclado.
2. Determine cuales son las ventajas o desventajas al escribir los valores de las variables numéricas que genera un programa en un archivo de salida, en relación a presentarlos directamente en pantalla.
3. Investigue si existe un método alternativo en Fortran 90, distinto al presentado en este capítulo, para leer todos los datos de un archivo de texto cuyo número total de datos es desconocido.

Bibliografía

1. Michael Metcalf y John Reid, *Fortran 90/95 explained*, Second edition, Oxford University Press, 1999.
2. Stephen J. Chapman, *Fortran 90/95 for Scientists and Engineers*, First edition, McGraw-Hill, 1997.

Capítulo 6

Vectores.

- El estudiante conocerá los conceptos básicos que le permitan hacer uso de vectores, especificando la dimensión y rangos correspondientes.
- El estudiante utilizará vectores para ordenar un conjunto de números usando el método de selección del mínimo con intercambio.

En Fortran 90 es posible definir vectores como ocurre en álgebra lineal. El equivalente a la dimensión en álgebra lineal es el rango o rank en Fortran 90. Para cada rango, es necesario definir en Fortran 90 el número de valores que se almacenará en dicho rango mediante el comando DIMENSION, indicando el índice del valor inicial y el índice del valor final en el vector. También es necesario incluir el tipo de dato asociado a los valores almacenados en el vector. Para ilustrar lo anterior, considérese el siguiente código:

```
PROGRAM suma_vectores

  !PROPOSITO: Ilustrar la inicializacion de vectores
  ! y mostrar dos formas de sumarlos

IMPLICIT NONE

INTEGER :: i
REAL, DIMENSION(1:4) :: a = (/ 1., 2., 3., 4./)
REAL, DIMENSION(1:4) :: b = (/ 5., 6., 7., 8./)
REAL, DIMENSION(1:4) :: c, d

! Suma de los vectores elemento por elemento
DO i = 1, 4
   c(i) = a(i) + b(i)
END DO

! Suma de los vectores en bloque
d = a + b
```

```
! Se escriben los resultados en pantalla
WRITE (*,100) 'c', c
WRITE (*,100) 'd', d
100 FORMAT (' ',A,' = ',5(F6.1,1X))

END PROGRAM
```

En este ejemplo, se definen cuatro vectores: a, b, c, y d. Estos vectores tienen rango 1 y todos almacenan números de tipo real. En todos los casos, el índice de la primera componente es 1 y el índice de la última componente es 4. De manera análoga a las variables escalares numéricas reales y enteras, los vectores pueden inicializarse componente a componente. Lo interesante del ejemplo anterior es que muestra que dos vectores pueden sumarse componente a componente usando un ciclo DO, aunque también es posible sumarlos en bloque mediante el operador suma. La salida del programa muestra que ambas maneras de realizar la suma son equivalentes:

[localhost]$ ./a.out
c = 6.0 8.0 10.0 12.0
d = 6.0 8.0 10.0 12.0

También es posible declarar un vector de rango 1 de enteros como:

INTEGER, DIMENSION(1:4) :: vector_de_enteros = (/1,2,3,4/)

y un vector de cadenas de rango 1 como:

CHARACTER(len=9), DIMENSION(1:7) :: dia
dia(1)='lunes'
dia(2)='martes'
dia(3)='miercoles'
dia(4)='jueves'
dia(5)='viernes'
dia(6)='sabado'
dia(7)='domingo'

Es muy importante que el número total de componentes de un vector por rango sea suficiente para almacenar los datos que se necesitarán almacenar para evitar errores de desbordamiento. Es decir, un vector en Fortran no puede almacenar más valores que el número máximo definido por los índices inicial y final por rango mediante la siguiente formula:

Número máximo de componentes = índice final - índice inicial + 1

donde el índice final es mayor o igual al índice inicial.

Un vector de rango 2 puede usarse como un tipo de dato para representar tres matrices de 3×5 con componentes reales mediante la siguiente definición:

REAL, DIMENSION(1:3,1:5) :: m1,m2,m3

Si quisieramos conocer la suma de las matrices $m1 + m2$ podríamos realizar la suma correspondiente, componente a componente, mediante dos ciclos DO anidados. Una alternativa más simple para realizar esta tarea podría realizarse mediante el uso del operador suma:

m3 = m1 + m2

Esto es posible siempre y cuando las matrices sean conformes, es decir, siempre que tengan el mismo número de componentes en cada rango independientemente del valor del índice inicial y final.

Otra ventaja de los vectores en Fortran 90 es que al aplicar una función algebraica o trigonométrica a un vector, esto es equivalente a aplicar la misma función a cada componente del vector. Por ejemplo, las siguientes operaciones son equivalentes:

```
DO i=1,3
    DO j=1,5
       m3(i,j)=COS(m3(i,j))
    END DO
END DO
```

y

```
m3=COS(m3)
```

En Fortran 90 el rango máximo era originalmente 7. En la mayoría de los compiladores de Fortran se ha extendido este número en versiones más recientes. Por ejemplo, el máximo número de rangos en Fortran 2008 es 15 en gfortran, y 31 en el compilador de Intel.

6.1. Ordenamiento de un conjunto de números

Una aplicación o uso de los vectores en Fortran 90 es el ordenamiento de una secuencia de números mediante el algoritmo de selección del mínimo con intercambio. La idea básica consiste en considerar un conjunto de n números enteros. Se inicia con el elemento $i = 1$ y se compara con el menor valor de los siguientes $n - 1$ elementos. Si el menor elemento de los $n - 1$ elementos siguientes es menor que el elemento $i = 1$ entonces se intercambian ambos valores. Si eso no ocurre no hay intercambio. Posteriormente, se continua con $i = 2$ y se compara con el menor valor de los siguientes $n - 2$ elementos. Si el menor elemento de los $n - 2$ elementos siguientes es menor que el elemento $i = 2$ entonces se intercambian ambos valores. Si eso no ocurre no hay intercambio. Se continua así sucesivamente, hasta que $i = n - 1$. Si elemento n es menor que el elemento $i = n - 1$ entonces se intercambian ambos valores. Si eso no ocurre no hay intercambio.

De manera específica, en la figura 6.1 se describe la utilización del algoritmo de ordenamiento por selección del mínimo con intercambio para un conjunto de 5 números. En este caso, el primer elemento (10) se compara con con el mínimo de los siguientes cuatro elementos que resulta ser el segundo elemento (3). Como 3 es menor que 10 se realiza un intercambio entre el primer y segundo elemento. Posteriormente, se compara el segundo elemento (10) con el menor de los siguientes tres elementos que resulta ser el cuarto (4). Como 4 es menor que 10 se realiza un intercambio entre el segundo y el cuarto elemento. A continuación se compara el tercer elemento (6) con el menor de los siguientes dos elementos que resulta ser el quinto (9). Como 9 no es menor que 6 no se realiza intercambio. Finalmente, se compara el cuarto elemento (10) con el quinto. Como 9 es menor que 10 se realiza un intercambio y se finaliza el ordenamiento de los cinco números, de menor a mayor.

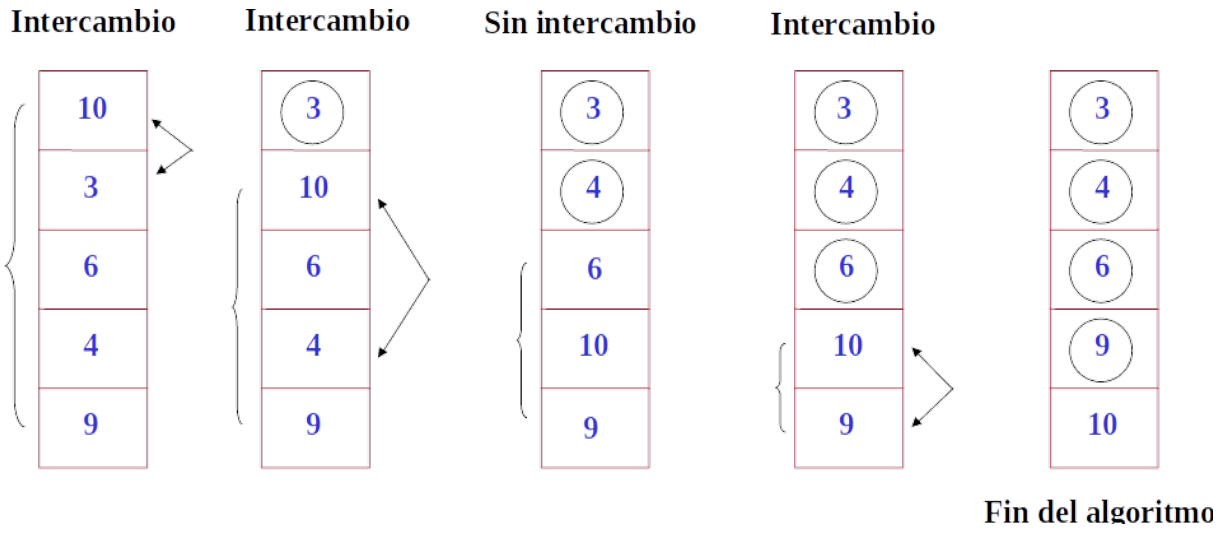

Figura 6.1: Esquema del algoritmo de ordenamiento mediante selección del mínimo con intercambio.

El programa donde se implementa el algoritmo anterior en Fortran 90 se muestra a continuación:

```
PROGRAM ordena1
!
!  Proposito: leer un archivo de datos con n\'umeros que
!  son ordenados en orden ascendente usando el algortimo
!  de selecion del minimo con intercambio. Al final se
!  se escriben los datos ordenados en  un archivo de salida.
IMPLICIT NONE

! Definicion de parametros
INTEGER, PARAMETER :: max_size = 10

! Definicion de variables
REAL, DIMENSION(max_size) :: a    ! Vector de datos
LOGICAL :: exceed = .FALSE.       ! Variable logica que indica
   que se
                                  ! excedieron los limites del
                                     vector
INTEGER :: i                      ! Indice del ciclo DO
INTEGER :: iptr                   ! Indice del elemento mas
   chico
INTEGER :: j                      ! Indice del ciclo DO anidado
INTEGER :: nvals = 0              ! Numero de valores a ordenar
INTEGER :: status                 ! Variable de estado, es 0
   si hubo exito
REAL :: temp                      ! Variable temporal del
   intercambio

! Apertura del archivo numeros.dat
! El estado es OLD porque el archivo de datos debe
! existir en el directorio donde esta el ejecutable
OPEN ( UNIT=9, FILE='numeros.dat', STATUS='OLD',
   ACTION='READ', &
       IOSTAT=status )

! Verificar que se abrio exitosamente el archivo
IF ( status == 0 ) THEN       ! Apertura exitosa
   ! En este punto la apertura del archivo fue exitosa,
   ! por lo que se procede a leer los datos para ordenarlos
   ! y escribir los datos ordenados en un archivo de texto.
   DO
      READ (9, *, IOSTAT=status) temp      ! Leer dato
      IF ( status /= 0 ) EXIT              ! Salir si es el
         fin de los datos
      nvals = nvals + 1                    ! Incrementar el
         contador
        IF ( nvals <= max_size ) THEN  ! Verificar el valor
           del contador
           a(nvals) = temp             ! Si es menor que el
```

```
            numero maximo de
      ELSE                              ! componentes pasar su
         valor al vector
         exceed = .TRUE.                ! De otro modo indicar
            desbordamiento
      END IF
END DO

! Si hubo desbordamiento
IF ( exceed ) THEN
   WRITE (*,1010) nvals, max_size
   1010 FORMAT ('Se excedio el numero maximo de componentes
      &
                  del vector: ', I6, ' > ', I6 )
ELSE

   ! En este punto no se excedieron los limites del vector,
   ! es decir, no hubo desbordamiento
   DO i = 1, nvals-1

      ! Buscar el minimo desde a(i) hasta a(nvals)
      iptr = i
      DO j = i+1, nvals
         IF ( a(j) < a(iptr) ) THEN
            iptr = j
         END IF
      END DO

      ! iptr tiene el indice del minimo
      ! Si i es diferente de iptr intercambiar
      ! a(iptr) con a(i)
      IF ( i /= iptr ) THEN
         temp    = a(i)
         a(i)    = a(iptr)
         a(iptr) = temp
      END IF

   END DO

   ! Se procede ahora con la escritura de los
   ! datos ordenados en un archivo de salida
   WRITE (*,'(A)') ' Los valores ordenados se escribieron &
                     en el archivo ordenados.dat'
   OPEN ( UNIT=10, FILE='ordenados.dat', STATUS='REPLACE',
      ACTION='WRITE', &
    IOSTAT=status )
   WRITE (10,'(4X,F10.4)') ( a(i), i = 1, nvals )
END IF
```

```
ELSE

   ! Hubo un error en la apertura del archivo de lectura
   WRITE (*,1050) status
   1050 FORMAT (1X,'No se pudo abrir el archivo de entrada, &
                el estado es ', I6)

END IF

CLOSE(9)
CLOSE(10)

END PROGRAM
```

Note que en esta implementación la variable max_size limita el número máximo de datos a ordenar. También es importante señalar el uso del comando EXIT para salir de un ciclo DO si el valor de la variable status es diferente de cero, lo cual indica que se llegó al final del archivo. Los números se leen inicialmente del archivo 'numeros.dat' y los números ordenados se escriben en el archivo 'ordenados.dat':

[localhost]$more numeros.dat
10.0
3.0
6.0
4.0
9.0

[localhost]$ ./a.out
Los valores ordenados se escribieron en el archivo ordenados.dat

[localhost]$more ordenados.dat
3.0000
4.0000
6.0000
9.0000
10.0000

Actividades:

1. Investigue como funciona el algoritmo de ordenamiento de burbuja (bubble sort), impleméntelo y compare su velocidad con respecto al algoritmo de ordenamiento presentado en este capítulo.
2. Investigue el uso de la instrucción ALLOCATABLE cuando se definen variables de tipo vector en Fortran 90. Discuta las ventajas y desventajas asociadas al utilizar este atributo.

Bibliografía

1. Michael Metcalf y John Reid, *Fortran 90/95 explained*, Second edition, Oxford University Press, 1999.

2. Stephen J. Chapman, *Fortran 90/95 for Scientists and Engineers*, First edition, McGraw-Hill, 1997.

Capítulo 7

Funciones y subrutinas.

- El estudiante conocerá los conceptos básicos que le permitan hacer uso de funciones y subrutinas.
- El estudiante utilizará una subrutina para ordenar un conjunto de números usando el método de selección del mínimo con intercambio visto en el capítulo anterior.

En Fortran 90 es posible definir funciones específicas análogas a las funciones algebraicas y trigonométricas intrínsecas.

A continuación se proporciona el código de una función que evalua un polinomio cuadrático:

```
PROGRAM cuadratico1
  ! Proposito: llamar una funcion que evalua
  !            un polinomio cuadratico
IMPLICIT NONE

REAL :: cuadratico  ! Declaracion del tipo de dato de la
   funcion
REAL :: polinomio   ! Declaracion de una variable para el
   resultado
REAL :: a, b, c, x  ! Declaracion de variables locales
INTEGER :: status   !

OPEN ( UNIT=9, FILE='coeficientes.dat', STATUS='OLD',
   ACTION='READ', &
        IOSTAT=status )

READ (9,*) a,b,c,x
PRINT *,'Coeficientes: a,b,c',a,b,c
PRINT *,'Evaluar en: ',x

polinomio = cuadratico(x,a,b,c)
```

```
! Escribir el resultado en pantalla
WRITE (*,100) ' cuadratico(', x, ') = ', polinomio
100 FORMAT (A,F10.4,A,F12.4)

CLOSE(9)

END PROGRAM

FUNCTION cuadratico( x, a, b, c )
!  Proposito: evaluar un polinomio cuadratico
!  de la forma
!  cuadratico = a * x**2 + b * x + c
IMPLICIT NONE
REAL :: cuadratico
! Declaracion de los argumentos de entrada
REAL, INTENT(IN) :: x
REAL, INTENT(IN) :: a
REAL, INTENT(IN) :: b
REAL, INTENT(IN) :: c

! Evaluacion del polinomio
cuadratico = a * x**2 + b * x + c

END FUNCTION
```

En este ejemplo, el nombre de la función debe declararse con un tipo de dato. Note el uso de la instrucción INTENT(IN). Este comando indica que los argumentos de la función son de entrada, lo cual implica que los valores de estas variables son sólo de lectura dentro de la función y no se puede modificar su valor original. Una limitación importante de las funciones es que sólo se puede generar un resultado de salida para ser asignado a una variable numérica. La salida del programa anterior se proporciona a continuación:

[localhost]$ more coeficientes.dat
1.0 2.0 3.0 1.0

[localhost]$./a.out
Coeficientes: a,b,c 1.00000000 2.00000000 3.00000000
Evaluar en: 1.00000000
cuadratico(1.0000) = 6.0000

Cuando se necesitan calcular varios resultados diferentes (por ejemplo, donde la salida corresponda a diferentes números reales y/o enteros) al mismo tiempo, es posible usar subrutinas. El ejemplo anterior usando una subrutina se muestra a continuación:

```
PROGRAM cuadratico2
  ! Proposito: llamar una funcion que evalua
  !            un polinomio cuadratico
```

```
IMPLICIT NONE

REAL :: polinomio   ! Declaracion de una variable para el
   resultado
REAL :: a, b, c, x  ! Declaracion de variables locales
INTEGER :: status   !

OPEN ( UNIT=9, FILE='coeficientes.dat', STATUS='OLD',
   ACTION='READ', &
        IOSTAT=status )

READ (9,*) a,b,c,x
PRINT *,'Coeficientes: a,b,c',a,b,c
PRINT *,'Evaluar en: ',x

CALL cuadratico(a,b,c,x,polinomio)
! Escribir el resultado en pantalla
WRITE (*,100) ' cuadratico(', x, ') = ', polinomio
100 FORMAT (A,F10.4,A,F12.4)

CLOSE(9)

END PROGRAM

SUBROUTINE cuadratico( x, a, b, c, salida )
!  Proposito: evaluar un polinomio cuadratico
!  de la forma
!  cuadratico = a * x**2 + b * x + c
IMPLICIT NONE
! Declaracion de los argumentos de entrada
REAL, INTENT(IN) :: x
REAL, INTENT(IN) :: a
REAL, INTENT(IN) :: b
REAL, INTENT(IN) :: c
!Declaracion de variable de salida
REAL, INTENT(OUT) :: salida
! Evaluacion del polinomio
salida = a * x**2 + b * x + c
END SUBROUTINE
```

La salida de este programa se proporciona a continuación:

[localhost]$ more coeficientes.dat
1.0 2.0 3.0 1.0

[localhost]$ ./a.out
Coeficientes: a,b,c 1.00000000 2.00000000 3.00000000
Evaluar en: 1.00000000

cuadratico(1.0000) = 6.0000

En ambos casos, tanto la subrutina como la función deben incluirse afuera del programa principal hasta abajo. En el caso de la subrutina, se observa el uso de la instrucción INTENT(OUT). Este comando indica que la variable es de salida, lo cual implica que sólo se usará esta variable para escribir un valor numérico sin poder leer su contenido original. Una tercera posibilidad es INTENT(INOUT). Esto significa que la variable afectada por este comando puede usarse en modo de lectura o escritura.

Con la finalidad de ilustrar la utilidad del uso de subrutinas para modularizar los procedimientos que se realizan en un programa en Fortran 90, a continuación se proporciona el código del programa discutido en el capítulo anterior, el cual ordena una secuencia de números, haciendo uso de una subrutina:

```
PROGRAM ordena2
!
!  Proposito: leer un archivo de datos con numeros que
!  son ordenados en orden ascendente usando el algortimo
!  de selecion del minimo con intercambio. Al final se
!  se escriben los datos ordenados en  un archivo de salida
!  usando subrutinas
IMPLICIT NONE

! Definicion de parametros
INTEGER, PARAMETER :: max_size = 10

! Definicion de variables
REAL, DIMENSION(max_size) :: a,b  ! Vectores de datos de
   entrada y salida
LOGICAL :: exceed = .FALSE.       ! Variable logica que indica
   que se
                                  ! excedieron los limites del
                                     vector
INTEGER :: i                      ! Indice del ciclo DO
INTEGER :: iptr                   ! Indice del elemento mas
   chico
INTEGER :: nvals = 0              ! Numero de valores a ordenar
INTEGER :: status                 ! Variable de estado, es 0
   si hubo exito
REAL :: temp                      ! Variable temporal del
   intercambio

! Apertura del archivo numeros.dat
! El estado es OLD porque el archivo de datos debe
! existir en el directorio donde esta el ejecutable
OPEN ( UNIT=9, FILE='numeros.dat', STATUS='OLD',
   ACTION='READ', &
        IOSTAT=status )
```

```
! Verificar que se abrio exitosamente el archivo
IF ( status == 0 ) THEN       ! Apertura exitosa
   ! En este punto la apertura del archivo fue exitosa,
   ! por lo que se procede a leer los datos para ordenarlos
   ! y escribir los datos ordenados en un archivo de texto.
   DO
      READ (9, *, IOSTAT=status) temp      ! Leer dato
      IF ( status /= 0 ) EXIT              ! Salir si es el
         fin de los datos
      nvals = nvals + 1                    ! Incrementar el
         contador
        IF ( nvals <= max_size ) THEN  ! Verificar el valor
           del contador
           a(nvals) = temp             ! Si es menor que el
              numero maximo de
        ELSE                           ! componentes pasar su
           valor al vector
           exceed = .TRUE.             ! De otro modo indicar
              desbordamiento
        END IF
   END DO

   ! Si hubo desbordamiento
   IF ( exceed ) THEN
      WRITE (*,1010) nvals, max_size
      1010 FORMAT ('Se excedio el numero maximo de componentes
         &
                   del vector: ', I6, ' > ', I6 )
   ELSE

      ! No hubo desbordamiento
      ! Se procede a ordenar los datos
      CALL ordena(nvals,a,b)

      ! Se procede ahora con la escritura de los
      ! datos ordenados en un archivo de salida
      WRITE (*,'(A)') ' Los valores ordenados se escribieron &
                        en el archivo ordenados.dat'
      OPEN ( UNIT=10, FILE='ordenados.dat', STATUS='REPLACE',
         ACTION='WRITE', &
       IOSTAT=status )
      WRITE (10,'(4X,F10.4)') ( b(i), i = 1, nvals )

   END IF

ELSE
```

```
   ! Hubo un error en la apertura del archivo de lectura
   WRITE (*,1050) status
   1050 FORMAT (1X,'No se pudo abrir el archivo de entrada, &
                el estado es ', I6)

END IF

CLOSE(9)
CLOSE(10)

END PROGRAM

SUBROUTINE ordena(n,va,vb)
!
!  Proposito: ordenar el vector va que tiene n componentes
IMPLICIT NONE

! Declaracion del vector de entrada y de salida
INTEGER, INTENT(IN) :: n                ! Numero de datos
REAL, DIMENSION(1:n), INTENT(IN)  :: va  ! Vector a ordenar de
   entrada
REAL, DIMENSION(1:n), INTENT(OUT) :: vb  ! Vector ordenado de
   salida

! Variables locales
INTEGER :: i                        ! Indice del ciclo DO
INTEGER :: iptr                     ! Indice del elemento mas
   chico
INTEGER :: j                        ! Indice del ciclo DO anidado
REAL :: temp                        ! Variable temporal del
   intercambio
REAL, DIMENSION(1:n) :: vt  ! Vector temporal

vt = va

! Ordenamiento del vector
DO i = 1, n-1

   ! Encontrar el valor minimo desde vt(i) hasta vt(n)
   iptr = i
   DO j = i+1, n
      IF ( vt(j) < vt(iptr) ) THEN
         iptr = j
      END IF
   END DO

   ! iptr contiene el indice del minimo
   ! si i es diferente de iptr intercambiar
```

```
    ! vt(i) y vt (iptr)
    IF ( i /= iptr ) THEN
       temp       = vt(i)
       vt(i)      = vt(iptr)
       vt(iptr) = temp
    END IF

END DO

vb = vt ! Exportar el archivo ordenado

END SUBROUTINE ordena
```

Note como se encapsuló el procedimiento mediante el cual se realiza el ordenamiento de los números en la subrutina ordena(n,va,vb), de tal manera que la estructura del programa principal es ahora mas clara y concisa.

Actividades:

1. Elabore un programa en donde haga uso de una subrutina que utilice como variables de entrada los 3 coeficientes de un polinomio cuadrático igualado a cero y que genere i) una variable que indique si las raíces son reales o complejas, ii) una variable de salida indicando el número de raíces asociado al polinomio igualado a cero, y iii) las variables que contengan el valor numérico de las raíces.
2. Investigue las ventajas y desventajas de usar la instrucción INTENT(INOUT) al definir una variable en una subrutina, en lugar de usar dos variables, una con el modificador INTENT(IN) y otra con el modificador INTENT(OUT).

Bibliografía

1. Michael Metcalf y John Reid, *Fortran 90/95 explained*, Second edition, Oxford University Press, 1999.
2. Stephen J. Chapman, *Fortran 90/95 for Scientists and Engineers*, First edition, McGraw-Hill, 1997.

Capítulo 8

Buenas prácticas para el diseño de aplicaciones

- El estudiante conocerá las ventajas de la programación estructurada.
- El estudiante conocerá las ventajas del uso de módulos y su documentación para facilitar su posterior revisión, lectura y extensión.

La programación estructurada tiene como propósito principal modularizar todos los procedimientos con la finalidad de hacer más claro y compacto el código, mediante el uso de subrutinas. Estas subrutinas son mas fáciles de validar, y pueden usarse como bloques constructores para producir subrutinas y/o programas más complejos.

Otro aspecto importante es el diseño de programas con una perspectiva de arriba hacia abajo, es decir, una vez que se tiene clara la manera en que debe resolverse un problema, lo más conveniente es proponer la creación de un conjunto de módulos o subrutinas que realicen ciertas funciones, particularizando posteriormente el comportamiento detalladio de cada parte por separado.

El comentar tanto las variables como las acciones que se van realizando en cada módulo y en el programa principal es fundamental. Esto es particularmente efectivo para recordar que fue lo que hicimos, además de que facilita la lectura de nuestros programas cuando son revisados por otras personas.

Finalmente, en cómputo científico es crucial realizar pruebas exhaustivas para verificar que no hay errores lógicos y para validar los resultados numéricos obtenidos por nuestros programas. Una vez que el código ha sido validado, es posible realizar optimizaciones que permitan reducir el tiempo de ejecución, manteniendo al mismo tiempo una precisión adecuada para nuestras aplicaciones.

Actividades:

1. Investigue el uso de plataformas como GitHub https://github.com/ para la realización de programas de manera colaborativa, así como las herramientas que

proporciona este tipo de plataformas para documentar los programas generados.

Bibliografía

1. Michael Metcalf y John Reid, *Fortran 90/95 explained*, Second edition, Oxford University Press, 1999.
2. Stephen J. Chapman, *Fortran 90/95 for Scientists and Engineers*, First edition, McGraw-Hill, 1997.