Capítulo 1

Subrutinas

1.1. Manejo de Cálculos Repetitivos

En muchos programas, en particular en algunos muy largos y complejos donde varias veces se rehacen los mismos cálculos una y otra vez podrían utilizarse funciones externas para simplificar el código y volverlo más simple y compacto ¿Pero qué pasa si lo que se quiere calcular no tiene estructura de función? Es decir, necesito un subprograma parecido a como trabaja la función externa, pero que a diferencia de la función me devuelva varios resultados.
Para este tipo de trabajo se diseñaron las **Subrutinas**, que tienen las siguientes propiedades y formas de uso:

- Se escriben en forma externa al programa, tal como vimos lo hacen las funciones externas.
- Intercambian con el programa principal una lista de variables. A esas variables se las llama argumentos.
- Los argumentos son variables que se envían del programa a la subrutina y cuando esta termina son devueltos. Sólo esa lista es intercambiada. No se define cuales de estos argumentos son datos que ingresan información o cuales son los que devuelven los resultados[1]. Es decir, los argumentos son una lista de variables que se intercambia con la subrutina, donde cualquiera de estas variables puede ser modificada o no.
- Las subrutinas son llamadas del programa principal a partir de la orden **CALL Nombre de la subroutina(lista de variables)**. No son parte de un cálculo o una asignación, como las funciones.
- Las subrutinas se escriben al final de programa principal en el mismo archivo o en archivos separados que se compilan junto al principal. Se debe indicar como primer sentencia la orden: **SUBROUTINE Nombre de la subrutina(lista de variables)**.
- Los argumentos se copian a variables en el espacio de memoria asignado a la subrutina y cuando esta termina se copian de nuevo a la memoria del programa. Las variables internas de la subrutina desaparecen en el momento que esta termina (veremos en este capítulos una forma de evitar esto).

[1] En Fortran 90/95 es posible indicar variables que van ser modificadas por la subrutina, variables que no lo serán (inmutables), e incluso en variables que sólo tendrán los resultados de su ejecución. Pero esta facilidad es opcional y debe ser indicada explícitamente en el códifgo

- Al igual que en las funciones externas, es necesario que las variables en los argumentos de llamada de la subrutina sean del mismo tipo y dimensión en el programa y en la subrutina. Por eso, el orden de cada variable en los argumentos debe ser el mismo en el programa principal y en la subrutina.
- La subrutina retorna al programa cuando ejecuta la sentencia **RETURN**. Este retorno se hace exactamente al lugar donde fue llamada. Un programa puede llamar a una subrutina todas las veces que sea necesario.
- La última sentencia de una subrutina es al igual que cualquier otro programa, es la orden **END**.
- Una subrutina puede llamar a otra subrutina y cuando termina devuelve el control a la que la llamó en el punto que la llamó.
- Una subrutina puede llamarse a sí misma. Esta propiedad se llama **recursividad** y discutiremos el funcionamiento de esta propiedad con detalle más adelante.
- Si en el programa existe una llamada a una subrutina en particular pero esta no es encontrada por el compilador se señalará como error. No se indica como error la existencia de una subrutina que no es usada por el programa principal u otras subrutinas.
- Las subrutinas pueden no estar escritas en el mismo lenguaje que el programa que las llama, pero sí se deberán respetar el tipo de variable, su dimensión y el orden de los argumentos en la llamada.
- Las subrutinas pueden precompilarse en grupos y estos archivos se los denomina Librerías. Los sistemas operativos tienen muchas librerías que son accesibles desde los programas, incluso algunas de estas, las consideradas esenciales están cargadas en la memoria ram. Por ejemplo, existen librerías para que los programas Fortran puedan hacer dibujos (PGPLOT, etc) o hacer cálculos vectoriales (BLAS, etc).

1.1.1. Uso de Subrutinas

Veamos como funciona en un caso real, donde el programa Prueba llama a la subrutina TEST

```
Program Prueba
REAL*4 C(100)
Integer B
⋮
CALL TEST(A, B, C)
⋮
END

SUBROUTINE TEST(X,Y,Z)
REAL*4 Z(100)
Integer Y
⋮
RETURN
END
```

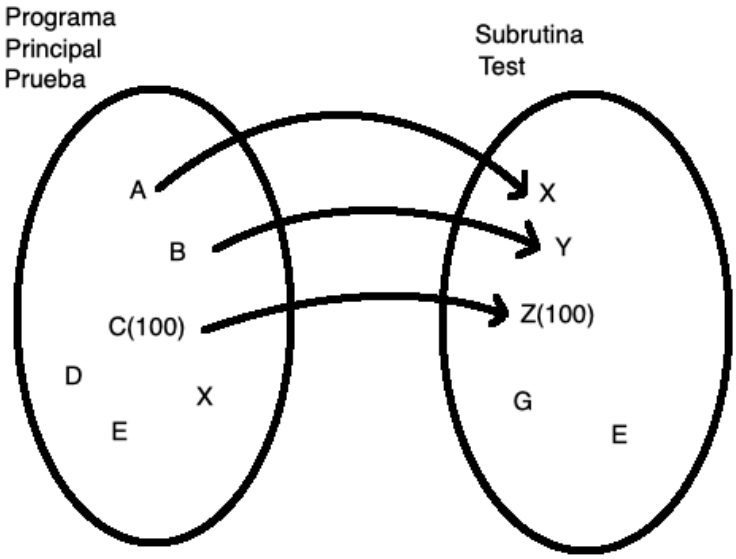

Figura 1.1. Esquema de como las variables de los argumentos de la llamada a una subrutina son reasignados a las variables de esta. Note que cada variable debe ser del mismo tipo y dimensionalidad que en el programa principal. La variable E de la subrutina desconoce que existe una variable también llamada E en el programa principal

En la figura 1.1 podemos ver un esquema de como las variables son transferidas del programa principal a la subrutina, para el caso del ejemplo que acabamos de ver. Notar que sólo A, B y C que son los argumentos de la llamada a la subrutina en el programa Prueba y son las únicas variables que son copiadas a la subrutina. Otras variables del programa como D, E y X no son transferidas y su existencia será desconocida por la subrutina. Las variables A, B y C son transferidas a un nuevo espacio de memoria como las variables X, Y y Z. Como B es una variable entera, sólo puede ser transferida a otra variable entera, por eso Y es definida como entera también en la subrutina. Y en el caso del vector C de 100 elementos tiene que ser recibida por una variable de igual dimensión y tipo, en este caso Z que fue definida correctamente para el caso en la subrutina Test. Note que el orden en los argumentos en el programa principal es concordante con la variable elegida en la subrutina (de igual tipo y dimensionalidad).

Cuando una subrutina se encuentra con la sentencia RETURN termina su trabajo y retorna las variables que están en los argumentos al programa principal, volviendo a reasignar los valores. En este caso las variables X, Y y Z se copian a A, B, y C. Las variables G y E de la subrutina test se pierden cuando esta termina y se las considera variables temporales[2]. En cambio en el programa Prueba las variables D, E y X nunca se enteraron de la llamada a Test, a pesar de que en la subrutina había dos variables con igual nombre (X y E).
Si, en cambio, es posible hacer lo siguiente:

CALL TEST(A+2.5, B, C)

En este ejemplo, se hace la cuenta A+2.5 y el resultado se envía a la subrutina y es asignado a la variable X. Pero en este caso en particular inhibimos a la variable A de recibir cualquier resultado

[2] A menos que se haya indicado alguna de ellas con el comando SAVE, que veremos más adelante

de producto de la ejecución de la subrutina.

Resumiendo, la subrutina cuando se ejecuta crea su propio universo de variables y sólo recibe datos por los argumentos de la llamada. Este universo de variables desaparece cuando la subrutina termina. Si la subrutina es vuelta a llamar el espacio de memoria se crea nuevamente, pero los datos en las variables anteriores no se conservan.

1.2. Ejemplos de Subrutinas

Como ejemplos realizaremos dos subrutinas que se encarguen de la transformación de coordenadas polares a cartesianas y viceversa, junto con un programa que las llama y realiza ambas transformaciones.

```
program polares
pi=atan(1)*4

write(*,*)'Ingrese X,Y'
read(*,*) X,Y

call pola(X,Y,r,theta)
write(*,*)'R =',r,'Theta =', theta/pi*180

call cartesianas(X,Y,r,theta)
write(*,*)'x =',X,'y =',Y

end

Subroutine pola(x,y,r,theta)
r = sqrt(x*x + y*y)
theta = atan2(y,x)
return
end

Subroutine cartesianas(x,y,r,theta)
x=r*cos(theta)
y=r*sin(theta)
return
end
```

1.2.1. Programando con subrutinas

Las subrutinas traen aparejado muchas mejoras para resolver problemas complejos que suelen generar programas grandes y largos. Esto se debe a que permiten dividir el problema grande en muchos pequeños cuyo solución es más fácil de manejar y testear. Existen libros[3] y muchas páginas web con subrutinas ya programadas que abarcan casi todas las áreas de la matemática. Esto no sólo es cierto para algoritmos matemáticos simples, sino que también para los muy complejos. Desarrollar algoritmos para computadoras es toda una rama de la matemática contemporánea. El uso masivo de computadoras creó a su vez otros campos en el desarrollo de la matemática. En

[3] Veremos en este curso el libro Numerical Recipes, William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Este libro se puede consultar aquí

capítulos siguientes trataremos con ejemplo dos de estos campos, el de ordenar y el de generar modelos con números al azar.
¿Cuáles son las metas de estas nuevas áreas de la matemática asociada a la computación? En si, resolver los problemas con algoritmos novedosos o variantes que tengan las siguientes propiedades:

- Que realicen la menor cantidad de operaciones matemáticas. En si, menos operaciones implica menos tiempo de computación para resolver el problema y por lo tanto mayor eficiencia en su uso. También menos pérdida de precisión en los resultados.
- Establecer cuál es el número de estas operaciones para comparar con otros algoritmos.
- Establecer como la cantidad de operaciones crece cuando aumenta el tamaño del problema (ejemplo: Si mi problema tiene un orden de tamaño N ¿El tiempo que tarda va con N elevado alguna potencia?
- Si no es posible calcular la cantidad de operaciones ¿Es posible obtener un número promedio de estas?
- ¿Cuál es la precisión matemática del resultado? Hasta que punto puede crecer el problema y se obtendría una solución útil?

Por otro lado, hay otra serie de ventajas de usar en forma masiva subrutinas para resolver un problema, que también son dignas de mención:

- Permiten trabajar en colaboración con otros colegas sobre un mismo proyecto, ya que el trabajo puede dividirse en partes.
- Escribir utilizando subrutinas hace que se puede recuperar muy fácilmente el trabajo de otros proyectos anteriores ya terminados, en los cuales se han implementado algoritmos para realizar ciertos cálculos que ahora se tendrán que volver a hacer.
- Permiten llegado el caso de encontrar un algoritmo más eficiente, mejorar un programa con solo modificar la subrutina que lo controla.
- Con el uso de subrutinas para todos los algoritmos necesarios se reduce el programa principal a que este sea el que llama en orden a las subrutinas y en si, son las subrutinas que realizan el trabajo, donde cada una lo hace parcialmente, pero el conjunto de todas lo resuelve.

1.3. Función Externa y uso de funciones en una subrutina

Las funciones pueden definirse como **external** haciendo:

External nombre_de_la_función

Esta orden tiene dos formas de trabajo muy diferentes. La primera de ellas es la de reemplazar una función intrínseca como por ejemplo puede ser el cos(), por una creada por el propio usuario. Es decir, si se trabaja en una computadora cuyo compilador Fortran no tiene una función cos() que satisfaga los requerimientos necesarios (por ejemplo: precisión en los decimales). Esta puede ser reemplazada por una propia, programada por el usuario. Con poner la orden, tal como está en el ejemplo, ya no se llama a la función intrínseca sino a la construida por el usuario. Se podría evitar

hacer esto, por ejemplo, poniéndolo otro nombre, como COS_mio() pero había que editar todo el programa para buscar las llamadas de cos() y reemplazarlas por COS_mio(). En cambio, de esta manera, con EXTERNAL, no hay que hacer ningún reemplazo. El usuario es ahora el dueño del nombre cos() para su función.

La otra función de la orden EXTERNAL es declarar que una función puede ser parte de los argumentos en el llamado de una subrutina, y por lo tanto la subrutina queda habilitada para usar la función al recibirla en los argumentos como una variable mas.

```
PROGRAM AREA
EXTERNAL FUN
⋮
CALL RUNGE (FUN, LOW, HIGH, AREA2 )
⋮
END

FUNCTION FUN( X )
⋮
RETURN
END

SUBROUTINE RUNGE ( F, X0, X1, A )
⋮
RETURN
END
```

1.4. Sentencia Common - Include

Hasta ahora hemos visto que la única forma de transferir datos entre un programa y una subrutina es a través de los argumentos que se encuentra en la llamada a la subrutina. Pero forma tiene una limitación a pocas variables, y por ello no es el único método. Cuando las variables en los argumentos es muy grande se prefiere enviarlos a través de la sentencia COMMON usada en combinación con la orden INCLUDE.
Para utilizar este comando en la zona de definición de variables debo indicar el COMMON que defino con su nombre[4] y sus variables asociadas. La forma general de la definición sería:

```
COMMON /NOMBRE1/ Lista de variables
COMMON /NOMBRE2/ Lista de variables
```

Ejemplo:

```
COMMON /listado1/ A,B,C,IK,X(1000)
COMMON /listado2/ B1,B2,B3,B4
```

[4] Puede no tener nombre, pero entonces no puedo poner más de una de estas sentencias

y en las subrutinas tendría poner:

```
SUBROUTINE SUB1()
COMMON /listado1/ X,Y,Z,J,ES(1000)
⋮
RETURN
END
```

Vemos que esta subrutina recibe también como argumentos las variables del COMMON listado1 Pero la segunda podría ser asi:

```
SUBROUTINE SUB2()
COMMON /listado1/ X,Y,J,ES(1000)
COMMON /listado2/ B1,B2,B3,B4
⋮
RETURN
END
```

Es decir la SUB1 recibe los argumentos del primer COMMON, mientras la segunda SUB2 recibe los argumentos de ambos COMMONs: listado1 y listado2. Una tercera subrutina podría no recibir ninguno de los dos listados de variables, es decir, no tendría ninguna de las sentencias COMMON en su código.

La sentencia INCLUDE “nombre_de_un_archivo"hace que el compilador cargue y compile la secuencia de Fortran que está escrita en el archivo nombrado, en ese lugar del programa. Es una práctica común poner muchas de las definiciones de variables y COMMON´s en un archivo aparte y que este sea incluido por el compilador, tanto en el programa principal, como en las subrutinas. Esto permite que no haya diferencias entre las definiciones de las variables que se comparten como argumentos entre el programa principal y sus subprogramas. Los archivos que tienen esta información (y que va a ser incluida) suelen tener terminación “.h” como norma.

1.5. Recursión

Se denomina **recursión** cuando una subrutina se llama a si misma. Esto tiene sentido cuando un problema es posible reducirlo en un orden de complejidad pero sigue siendo el mismo problema. Esta situación se da en algoritmos tipo diagrama de árbol, que son muy usados, por ejemplo, en programas de cálculo de la evolución dinámica de las estrellas en una galaxia.

Para ver un ejemplo real y simple estudiaremos el factorial, que cumple con las propiedades que hemos descripto. Recordar que factorial de N, se puede escribir como N!=N(N-1)!. Es decir convierto el factorial de N en resolver ahora el factorial de (N-1)!. Y entonces podría repetir el procedimiento hasta que mi problema quede reducido al factorial de 1 que por definición 1!=1.

Por otro lado hay que recordar que cuando una subrutina es llamada, crea su propio universo de variables que no son ni las del programa principal, ni el de las otras subrutinas. Incluso, cuando una subrutina se llama así misma crea otra zona memoria para sus variables, que no se comparte contra su propia versión iniclal. Es decir la subrutina madre no comparte variables con ls subrutina

hija, salvo las que se reasignan porque están en los argumentos del llamado.

Podemos entonces hacer un programa muy compacto que resuelva el factorial usando una subrutina recursiva y sería así:

```
program factor
write(*,*)'Cual es el numero:'
read(*,*) n
call factorial(n,p)
write(*,*) p
end

Recursive Subroutine factorial(n,p)
if(n.gt.1) then
Call factorial(n-1,p)
    p=p*n
else
    p=1
endif
return
end
```

En este caso en particular para el compilador GFORTRAN debo indicar que la subrutina es recursiva con el aviso de RECURSIVE en el nombre de la subrutina, pero esto puede cambiar según el compilador Fortran que se use.

1.6. Save

La función SAVE se usa en la subrutinas de la siguiente manera :

SAVE lista de variables

Esta sentencia se agrega al principio de la subrutina e indica cuales variables se conserven cuando la subrutina se cierra y los valores guardados sirven para ser usados en el próximo llamado. Esto destruye el modelo del uso de memoria que hemos descriptos (es una forma antigua de programar en Fortran) y su uso inhibe la posibilidad de hacer recursiones. Esta sentencia vuelve a un uso de memoria más antiguo donde la subrutinas podían volver llamarse y las variables conservaban los valores de la llamada anterior.