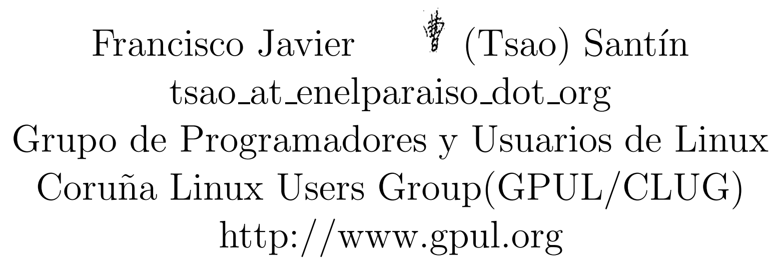
# Notas de Fortran 77 aplicado a Computación Numérica
# Versión 0.3

Francisco Javier (Tsao) Santín
tsao_at_enelparaiso_dot_org
Grupo de Programadores y Usuarios de Linux
Coruña Linux Users Group(GPUL/CLUG)
http://www.gpul.org


20 de Diciembre de 2001

Copyright ©2001 Francisco Javier Tsao Santín. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

# Índice general

**1. Operaciones básicas** **5**

**2. Un poco sobre arrays, un poco de estructuras iterativas, y una pizca de estructuras de control...** **8**

**3. I/O** **16**

**4. Algunos consejos para diseñar algoritmos C.N.** **24**

**5. Modularización** **34**

**6. Consejillos finales para depurar programas** **45**

**7. GNU Free Documentation License** **50**

1. APPLICABILITY AND DEFINITIONS . . . . . . . . . . . . . . . 51
2. VERBATIM COPYING . . . . . . . . . . . . . . . . . . . . . . . 53
3. COPYING IN QUANTITY . . . . . . . . . . . . . . . . . . . . . 53
4. MODIFICATIONS . . . . . . . . . . . . . . . . . . . . . . . . . 54
5. COMBINING DOCUMENTS . . . . . . . . . . . . . . . . . . . . 56
6. COLLECTIONS OF DOCUMENTS . . . . . . . . . . . . . . . . 57
7. AGGREGATION WITH INDEPENDENT WORKS . . . . . . . 57
8. TRANSLATION . . . . . . . . . . . . . . . . . . . . . . . . . . 57
9. TERMINATION . . . . . . . . . . . . . . . . . . . . . . . . . . 58
10. FUTURE REVISIONS OF THIS LICENSE . . . . . . . . . . . 58

## Position Paper

El Fortran es, como su propio nombre, FORmula TRANSlator, indica, un lenguaje de programación imperativa esencialmente enfocado a la realización de cálculos matemáticos. Es el lenguaje en activo más antiguo (tiene un año más que Lisp, que apareció en 1958). Su importante arraigo entre la comunidad científica, es la principal causa de su posición destacada en áreas de ciencia e ingeniería, hasta el punto de que se han realizado importantes mejoras en el muy obsoleto estándard F77 hasta convertirlo en uno de los importantes lenguajes para procesamiento en paralelo, como los Fortran 95 o HPF High Performance Fortran, pasando por una especie de fusión entre el viejo 77 y el Lenguaje C, conocida como Fortran 90.

Este documento son unos pequeños apuntes para un cursillo acelerado de iniciación a Fortran 77, pensados esencialmente para su aplicación inmediata en un curso básico de Cálculo Numérico. NO pretenden sustituir en absoluto a ningún libro. Mi intención primordial es facilitar una toma de contacto con la programación para estudiantes que la han tenido ya pero les *queda muy lejos* (suele pasar) o ésta ha resultado más bien frustrante. Por lo tanto, parto de que al lector *le suena* un poco todo esto; además de practicar los ejemplos que se muestran y se proponen, se debe ampliar el conocimiento de todas las extensiones del lenguaje acompañandose de un buen libro, para lo que adjunto una pequeña bibliografía que en su momento me fue útil, y que a los que, como yo, son alumnos de la Escuela de Ingenieros de Caminos de Coruña, les resultará ya familiar (además, conviene tenerlo para comprobar que no me equivoco en algún punto del documento, sea en la forma, o lo que es más peligroso, en el fondo).

El primer capítulo está dedicado a afinar algunos aspectos muy elementales sobre operaciones a las que no se suele prestar demasiada atención. En el segundo introduzco formaciones para poder empezar a hacer algunos ejemplos interesantes; además trato las estructuras básicas de la programación estructurada. El tercer capítulo machaco las posibilidades para entrada y salida de datos. El cuarto lo dedico a practicar un poco con la sesera para diseñar algoritmos de problemas comunes en C.N., y el quinto está destinado a plantear la organización global de un programa; estos dos recomiendo que se lean a la vez, están en este orden porque me *dio el punto* de escribirlos así pero podrían ir perfectamente al revés: los quinto y sexto (depuración de

errores)son temas que acaban con las expectativas del documento y permiten empezar a trabajar en serio, pero el cuarto es un tema al que se debe dedicar más tiempo, pero más espaciado para mejor asimilación, no tiene importantes conceptos y de lo que se trata es de entrenarse, para lo que propongo algunos ejemplos al cual más duro.

Podría haberme esforzado un poco más, pero personalmente creo que F77 está bien para cosas muy básicas, o para salir al paso de necesidades inmediatas que no se puedan cubrir con una calculadora programable; prefiero dedicar mi tiempo a aprender otros lenguajes más modernos y *divertidos*, como C++, CAML, Perl o incluso Ensambaldor...

# Capítulo 1

# Operaciones básicas

Bien, empezemos. Supongo que ya sabemos las normas de escritura de código fuente (lo de primera columna para marcar comentarios con una `c` o un `!`,segunda a quinta para etiquetas, sexta para indicar continuación de línea, con un punto o asterisco ("asteroide", como le llamó alguno) y de la 7 a la 72 para el código propiamente dicho),los tipos de datos: `integer, real, character`... Lo primero que recomiendo es:

```
program nombre_programa
implicit none
```

al principio de cada módulo

¿Por qué eliminar la declaración implícita de variables? Muy sencillo: porque aunque puede resultar muy cómoda, por tener que escribir poco y especialmente en módulos de programa con muchas variables, a la hora de depurar el programa nos puede complicar bastante la vida, ya que, por ejemplo, si tenemos una variable `inode`, y resulta que nos equivocamos al teclear y escribimos `lnode`, el compilador no detectará el error, y a la hora de operar, la máquina tomará en general 0 si no le damos otro valor previo, lo cual sólo lo detectaremos inspeccionando el código, o ejecutando el programa, lo que a veces es más difícil de ver, a no ser que sea un divisor, entonces tendremos un mensaje del estilo `Aritmetic trap error.Divide by zero` (en sistemas más amables, la máquina puede llegar a digerirlo y empezar a soltar Inf. entre los resultados). Sí recomiendo, por contra, procurar mantener las habituales costumbres a la hora de nombrar las variables: nombres que *digan algo*, empezar las variables enteras por $i-p$ para distinguirlas más fácilmente de las reales...¡Ah!, y no nos olvidemos de declarar todas las variables al principio del módulo, nada de andar haciendo declaraciones en mitad del código.

Ya sé que esto parece una estupidez, pero no sería la primera vez que lo veo hacer.

Con el progreso de las computadoras puede parecer, y de hecho en la mayoría de los casos lo es, un problema menor el de la memoria. Sin embargo es bastante sano y elegante no usar más de lo que se necesita, así que no es bueno utilizar un `real*16` si el problema no va a exigir una precisión determinada.

Otro punto importante es la **consistencia** a la hora de asignar y operar. Ejecutemos un ejemplo:

```
      program ejemplo_1
      integer a,b
      real*8 c1,c2,d,e,f1,f2,g,h,i1,i2
      open(unit=10,file='ejemplo1.res',status='new')
      a=1
      b=3
      c1=a/b
      c2=1/3
      d=1.
      e=3.
      f1=d/e
      f2=1./3.
      g=.1d+01
      h=.3d+01
      i1=g/h
      i2=.1d+01/.3d+01
      write(10,10)'c1=',c1,'c2=',c2
      write(10,10)'f1=',f1,'f2=',f2
      write(10,10)'i1=',i1,'i2=',i2
 10   format(a3,1x,d20.13,1x,a3,1x,d20.13)
      end
```

Aparentemente,`c1,c2,f1,f2,i1,i2` deberían almacenar el mismo resultado. Sin embargo, la realidad es bien diferente:

```
c1=  0.0000000000000E+00 c2=  0.0000000000000E+00
f1=  0.3333333333333E+00 f2=  0.3333333432674E+00
```

```
i1=  0.3333333333333E+00 i2=  0.3333333333333E+00
```

¿Que es lo que ha pasado?:
· `c1` divide dos variables enteras, y al igual que `c2`, que divide dos enteros, el resultado será sólo la parte entera
· `f1` da el resultado correcto, ya que aunque la asignación de valores a `d` y `e` es de reales en simple precisión (también se puede escribir .1e+01), la máquina los convierte al almacenarlos en variables en doble precisión, mientras que `f2` hace las operaciones en simple precisión y al convertir el resultado para almacenarlo en doble precisión rellena con basura
· finalmente `i1` y `i2` son las expresiones correctas y tendremos en consecuencia los resultados que queremos.

Si nos vemos obligados a mezclar en una expresión variables o constantes debemos emplear las funciones de F77 para convertirlas adecuadamente, así `dfloat()` convierte a doble flotante,`int()` convierte a entero, truncando por la coma si es un decimal. Hay una buena lista de estas funciones, para más detalles y mejor uso, consultar la bibliografía.

Por último, recomendar inicializar las variables a 0, ya que aunque en general los sistemas ya lo hacen automáticamante, a veces pueden no hacerlo, con lo que conlleva.

# Capítulo 2

# Un poco sobre arrays, un poco de estructuras iterativas, y una pizca de estructuras de control...

Los arrays, o formaciones (en castellano), o vectores y matrices (en Caminos), tras su apariencia un tanto mística para algunos, no son, en el fondo, más que un montón de variables que se llaman con un mismo nombre, de cara al usuario, y se apiñan en la misma zona de memoria, de cara a la máquina. Ésto último, que podría parecernos poco útil, tiene su importancia, especialmente cuando el tamaño del problema alcanza dimensiones considerables.

Cada variable del array se determina por un grupo de números que determinan su posición. El tamaño de grupo es lógicamente, el número de dimensiones del array. Por otra parte, la máquina ordena las variables en su memoria de forma lineal, marcando distintas variables sobre la formación para agilizar la búsqueda por la memoria, partiendo de ellas. Así pues, en una matriz 2D, en F77, los elementos se ordenan por columnas, esto es:

| a11 | a21 | a31 | ... | an1 | a12 | a22 | ... |
|---|---|---|---|---|---|---|---|

Como veremos más tarde, cuando un módulo envía a otro un array lo hace enviándole la dirección de memoria del primer elemento, y de ahí se va al primer elemento de la columna a la que pertenece el elemento al que que-

remos acceder. F77 utiliza esta forma de comunicación para todo, variables e incluso constantes, denominada *por referencia*, y permitiendo así el acceso directo a la zona ocupada por el elemento,permitiendo su modificación. Esto tiene sus ventajas, pero también sus obvios riesgos. Otros lenguajes permiten el paso de argumentos *por valor*, esto es, enviando el valor de la variable pero no permiten acceso a ella, evitando posibles efectos colaterales. Volveremos más tarde sobre esto. Ahora, ejecutemos un ejemplo:

```
      program modulo
      integer n,i
      parameter (nmax=4)
      real* v,c
      dimension v(nmax)
      n=3
      c=0.d+00
      v(1)=.1d+01
      v(2)=.2d+01
      v(3)=.3d+01
c...ahora introducimos los valores del array en el propio codigo; en el
c...siguiente parte aprenderemos a sistematizarlo leyendo por teclado o
c...fichero de datos

      do i=1,n
        c=c+v(i)*v(i)
      enddo

      write(6,10)'c=',c
 10   format(a2,d15.7)
      end
```

Y el resultado será:

```
c=  0.1400000E+02
```

¿Qué cosas podemos comentar de aquí? Veamos. En primer lugar, la declaración de variables. Sabemos que el producto escalar va a necesitar un vector de `n` componentes, y lo hemos particularizado para 3. Podríamos olvidarnos

del `parameter` pero entonces tendremos que sustituir allá donde ponga `nmax` su valor. Por otra parte, podemos indicar otra alternativa a la declaración del vector:

```
real dimension v(nmax)
```

Es en bastantes ocasiones útil controlar la numeración de las formaciones: si queremos indicar que el array comienza en otro número que no sea el 1 (por defecto) entonces se lo debemos indicar así: `dimension v(numeroinicio: numerofinal)` ¡Cuidado ahora con los índices de las expresiones de los algoritmos!

No sólo es importante, sino que es obligatorio indicar el número de componentes al principio del programa principal, ya sea de forma directa o con un parameter previo. Todo lo contrario que en las subrutinas, si indicamos una dimensión exacta, duplicaremos el uso de memoria. Pero sobre eso volveremos en otro capítulo, también he metido los datos del problema en el código mediante asignaciones para dar a fondo la entrada y salida de datos más tarde.

En todos los programas en los que intervengan formaciones se debe utilizar una estructura de control (las vemos unos párrafos más adelante) para para evitar que se le pida a un programa que haga problemas mayores de lo que puede, en caso contrario, si el usuario, o también cuando exista un error de programación en el que se busque un elemento en una formación que no existe, entonces pasará que al ejecutarse el programa empezará por buscar el elemento fuera de su página de memoria. Puede *piratearlo* sin que el kernel del sistema se entere, con lo que cogerá lo primero que encuentre y lo devolverá, o sí lo detectará y matará el proceso devolviendo un error del estilo `virtual memory address violation` (violación de acceso a memoria virtual), caso de Open-VMS, o el mítico `segmentation fault` de los Unix.

Centrémonos en esta otra parte del programa:

```
do i=1,n c=c+v(i)*v(i) enddo
```

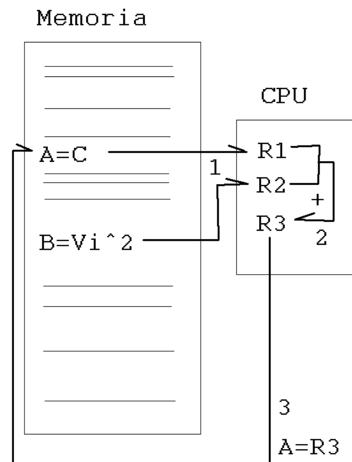
Esto es lo que se conoce como una estructura iterativa. `i` es el *contador*, y los números o variables separados por comas son el primer y el último valor del contador durante la iteración. Por defecto, el salto es 1, pero si se anade un tercer valor,éste será interpretado como salto. Si además es negativo, en-

tonces la iteración irá hacia atrás. Es importante controlar bien estos valores ya que si con el salto no se puede llegar a la condición de parada, el último valor, el programa al ejecutarse seguirá haciendo operaciones hasta el infinito o hasta que produzca un overflow en alguna variable[1].

Fijémonos en este trozo de código:

```
c=c+v(i)*v(i)
```

Parece un poco inconsistente, cuanto menos llamativo tener a ambos lados `c`. Esto se produce porque el almacenamiento de datos y resultados se produce en la memoria mientras las operaciones se realizan en el procesador. Aclaremos un poco esto:




Al llegar a esta instrucción, (1) el procesador llama a la memoria y solicita los valores de A y B, los guarda en sus propios registros R1 y R2, (2) realiza la operación, la almacena en otro registro R3, y (3) devuelve éste valor a la posición de memoria a la que se le asigne, en este caso a la misma A, por lo que en este caso concreto, se van acumulando los valores durante la iteración en ella.

[1]Nota para los usuarios de Open-VMS sobre el Vax de la Escuela: pulsar control-y y se parará el proceso

Que por qué no utilizo `v(i)**2` en vez de `v(i)*v(i)`? Porque carga innecesariamente de trabajo la máquina, ya que esta función de exponenciación emplea elementos de Cálculo Numérico para obtener el resultado, siendo, por tanto, menos óptimo para estos casos.

Por cierto, según unos compiladores u otros la declaración del contador puede ser desde necesaria hasta prohibida, por ejemplo, con `implicit none` es necesario declarar el contador, para el compilador de F77 de Digital(ahora de Compaq) Open VMS, sin embargo el de Microsoft (ahora, del Anticristo) protesta.

Otro ejemplo interesante:producto de una matriz a(mxn) por un vector v(n)

```
do i=1,m
   do j=1,n
      c(i)=c(i)+a(i,j)*v(j)
   enddo
enddo
```

Esto es una anidación de bucles. A algunas personas se les puede atragantar ver tanto índice junto (como en los tiempos lejanos del Álgebra Tensorial), lo importante es tener calma y, de la misma forma que lo hace la máquina, debemos seguir las variaciones del contador de uno en uno, y desde las últimas operaciones hasta las principales. Así pensamos:

$$c_i = \sum_{j=1}^{n} a_{i,j} * v_j \tag{2.1}$$

Y después pensamos en que debemos de calcular *cada* $c_i$.

Tenemos otra expresión para hacer bucles, es lo que se conoce como *do implícito*. En este último caso:
`c(i)=c(i)+a(i,j)*v(j),j=1,n`

Otra estructura iterativa típica que no está implementada en varios compiladores de F77 es la tipo *do...while*. Esto se puede suplir con una chapucilla:

```
      i=valorinicial
10
      ... expresion dependiente o no, de i

      if...condicion de parada
           si se cumple, continua el programa
           si no se cumple:
             i=i+salto
             goto 10
      endif
```

Este tipo de estructuras es especialmente útil por ejemplo en algoritmos iterativos que se detienen cuando la solución de un determinado problema alcanza una precisión determinada. Y aquí nos encontramos con otra de las grandes tipologías de la programación, las estructuras de control.

Las estructuras de control tienen la pinta:

```
if((condicion1).operadorbooleano.(condicion2)...)then
   ...codigo a ejecutar
else
   ...codigo a ejecutar en caso de que no se cumpla
      la condicion(opcional)
endif
```

donde `condicioni` puede ser una expresion `variableoconstante.cuantificador.variableoconstante2`, o más condiciones relacionadas por operadores booleanos anidadas. Los cuantificadores son:`eq`(*equal*,igual que), `gt`(*grower than*, mayor que),`le`(*lower than*, menor o igual que)...y `eqv` y `neqv` para comparar variables o constantes booleanas (.true. y .false., vaya), aunque en Open-VMS *traga* con `eq` y `neq`; y los booleanos son `and`, `or`,`not`. Podemos hacer *baterías* de `if ... endif` o anidados, según convenga; para indicar la complementaria de la condición utilizamos `else`, en cualquier caso, como al terminar un bucle, se cumpla o no la condición, el programa sigue su ejecución al acabar ésta.

Un ejemplo: cálculo del límite de $f(n) = 1/n$:

```
      program limite
      integer n
```

```
c...n sera el contador de iteraciones
      real*8 limite1,limite2,deltal
      n=1
      limite1=.1d+01/dfloat(n)

c...observa el dfloat, convierte a doble precision el entero

      n=2

 10   limite2=.1d+01/dfloat(n)
c...utilizamos dos variables para comparar en cada iteracion

        deltal=abs(limite2-limite1)
c...la funcion abs como ya te imaginas devuelve el valor absoluto

c...podriamos saltarnos directamente la variable deltal e incluir la
c...operacion dentro del if; tambien podriamos indicar la precision
c...con un parameter definido al principio del programa
      if(deltal.gt.0.1d-5)then
          limite1=limite2
          n=n+1
          goto 10
c...si no hay convergencia, se almacena el nuevo valor como antiguo y se
c...vuelve a calcular, sumando 1 al contador
      else
          write(6,15)n,deltal,limite2
 15       format('convergencia en',i5,' iteraciones, error',d15.7,
     .    ',limite',d15.7)
c...en caso contrario, se da como bueno el valor y se imprime
      endif

      end
```

Y este es el resultado:

```
convergencia en 1001 iteraciones, error  0.9990010E-06,limite  0.9990010E-03
```

*Problema 2.1: Escribir un programa para obtener el producto de dos matrices(hazlo con un ejemplo pequeño, e imprime el resultado con* `write(6,*)a(1,1)` *etc, en el siguiente capítulo aprenderemos a mostrar los resultados mejor)*

*Problema 2.2: Escribir un programa que evalúe la suma de alguna serie convergente que se te ocurra*

# Capítulo 3

# I/O

Bueno, ya va siendo hora de hacer programillas decentes, tomando datos de un posible usuario y devolviéndolos con una agradable presencia...

F77 tiene dos formas de manejar datos y resultados: libre y con formato. Pero además de saber el cómo, debemos pensar también en dónde.

Los comandos read y write van acompañados por un par de números. El primero indica unidad, mientras el segundo formato de lectura. Hay dos unidades que se suelen respetar normalmente, 5 para teclado y 6 para pantalla. Obviamente no vamos a leer datos de la pantalla ni vamos a enviar datos al teclado,así que ya sabemos a que comando corresponde cada una. Por otra parte si podemos extraer datos y enviar resultados a ficheros, que deberán de ser abiertos para trabajar con ellos y cerrados cuando no se precisen:

```
  ...
open(unit=numerounidad,name='nombredefichero',
   status='estadodefichero')
  ...
close(numerounidad)
```

Además de `name` también se puede utilizar `file`, uno de los dos está en vías de desaparición, pero ahora mismo no me acuerdo cual de los dos; el nombre puede darse como ahí arriba, con una constante de carácter, o con una variable del mismo tipo, y, por ejemplo, introducir el nombre a través de I/O; decir además que se puede indicar una extensión en el nombre, de

hecho al leer de uno se precisa si es que la tiene, típicas son .dat para datos y .res para resultados; por otra parte, si no se indica al crear uno para escribir sobre él algunos sistemas, como Open-VMS le añaden por defecto una extensión, normalmente .txt. El campo `status` puede ser `'old','new'`, o `'unknown'`. Para poder leer datos de un fichero, tendrá que haber sido previamente creado, y por lo tanto estará en `status='old'`, mientras que para escribir sobre un nuevo fichero tendremos `status='new'`. En el entorno de trabajo Open-VMS de nuestra Escuela el sistema crea un nuevo fichero con un número máyor en la versión si es que ya existe un fichero con ese nombre, habrá que tener cuidado con esto porque es relativamente fácil provocar un `disk quota excedeed` si no se hace una limpieza de vez en cuando; por otra parte, los que trabajamos con Linux nos encontramos con que si esto pasa, por defecto para el proceso, evitando sobreescribir.

Hay un campo más, `carriagecontrol`, una vez el profesor de C.N. explicó que tenía que ver con algo de las impresoras antiguas, pero ahora no lo recuerdo bien, y tampoco me apetece empezar a buscarlo, el que tenga interés, al libro, no me ha provocado especiales molestias...

¿Cuando utilizar teclado/ pantalla/ fichero? Si sólo hay dos o tres datos de entrada y/o salida, parece recomendable teclado y pantalla respectívamente, para lectura y escritura, si no se nos obliga a lo contrario. Pero suele suceder, sobretodo en programas de C.N. que el volumen de datos a tratar es mayor, por lo que es obviamente aconsejable en estos casos el uso de ficheros.

Lo de los formatos suele traer de cabeza a los programadores más novatos, y en el fondo no es más que una *incómoda tontería*. Lo importante es tener claro lo que se está haciendo. Si utilizamos formato libre, esto es, v.g.`read(numerounidad,*)variable1, variable2...variablen` la máquina tomará cualquier cosa que le entre por esa unidad y lo intentará almacenar en esas n-variables. Para que todo vaya bien, lo que se recoge y el tipo de variable deberán corresponder, sino los resultados son inpredecibles, desde conversión peligrosa de datos hasta defunción del proceso. Para escritura no hay este problema, pero tampoco es recomendable porque los resultados en formato libre dan un aspecto un tanto heterogéneo, desordenado. Por ello, creo que es recomendable el uso de formatos definidos.

La instrucción `format` indica a la máquina, como su nombre indica, el aspecto que espera que tenga algo que está leyendo o que va a tener algo

que va a imprimir. `format` va precedido una etiqueta que es la misma que se indica en el segundo campo de un `read` o un `write`. Algunos programadores recomiendan colocar todos las líneas `format` juntas al final del módulo del programa al que pertenecen, sin embargo yo recomiendo ponerlas justo a continuación de la instrucción o bloque de instrucciones I/O por las que son llamados.`format` va acompañado de una serie de campos que en número como mínimo serán los mismos que las variables y/o constantes que acompañen a la instrucción I/O. Aquí vuelvo a insistir sobre el tema de la consistencia, el tipo de variable y lo que se expresa en el formato deben ser iguales, de lo contrario los efectos pueden ser peligrosos. Veamos un ejemplo:

```
      program texto
      integer i
      real r
      i=15
      r=.35e-4
      write(6,10)'esto es entero',i,'y esto real',r
 10   format('Hola ',a14,i3,a12,e15.7)
      end
```

Este es el resultado:

```
Hola esto es entero 15 y esto real  0.3500000E-04
```

Observemos algo importante: `'esto es entero'` tiene calidad de constante de texto, como indica el formato de texto `a14`, mientras que `'Hola '` es en sí formato.¿A qué viene esto? hay gente que se confunde y hace cosas del estilo:

```
      write(6,10)'esto es entero',i,'y esto real',r
 10   format(i3,e15.7)
```

esperando que el resultado sea:

```
esto es entero 15 y esto real  0.3500000E-04
```

sin embargo lo que realmente hace el programa es morirse, ya que intentará dar salida chapuceramente al primer valor de texto a través de un formato entero, y si lo consigue hará lo propio con el valor entero sobre un formato real, y si todavía no ha protestado, lo hará al no ser capaz de dar

salida a los dos valores que restan... Además de dar entrada/salida a las variables indicadas, hay una serie de etiquetas de formato que sirven para indicar la disposición de los datos: espaciados(v.g.`3x` significaría tres espacios), saltos de carro (/) que son insertados como cualquier campo del `format`. Ejemplo: Supongamos que tenemos este fichero de datos:

```
Valor de i
123
Valor de r
-.2360e3
```

No olvidemos incluir un salto de línea después del último dato, en caso contrario podemos tener problemas porque al leer el real y a continuación encontrarse con el carácter de final de fichero, puede quejarse. Un programa que lee este fichero correctamente:

```
      program texto2
      real r
      integer i
      open(unit=11,file='texto2.dat',status='old')
      open(unit=12,file='texto2.res',status='new')
      read(11,10)i,r
 10   format(/,i3,//,e15.7)
      write(12,20)r,i
 20   format('Esto es real',1x,e12.3,1x,'y esto entero',1x,i3)
      close(11)
      close(12)
      end
```

Los saltos de línea nos permiten saltar los comentarios `Valor de i`... y recoger sólo los datos que precisamos. Esto es especialmente útil, porque a la vez que el usuario recibe el programa, también se le puede proporcionar un fichero plantilla para datos. Es importante pensar bien la cantidad de saltos que vamos haciendo al leer, yo mismo me he equivocado al escribir este código, insertando un sólo salto después de `i3` cuando precisaba dos:uno para saltar desde `123` a `valor de r` y otro para saltar desde ahí hasta la línea donde leeremos el real. El fichero de resultados que generamos:

```
Esto es real   -0.236E+03 y esto entero 123
```

Es importante, especialmente de cara a la depuración de errores, imprimir todos los datos que el programa recibe para saber que han entrado correctamente.

Vamos ahora que ya tenemos una ideilla de I/O a jugar un poco con algunos casos un poco más complejos. Pensemos en matrices. Supongamos este fichero de datos:

```
m
2
n
3
Matriz
.1e-1 .3e+0 .2e+0
.4e+1 .3e-2 .5e+1
```

Un código adecuado para leerlo:

```
      program texto3
      parameter(mmax=5,nmax=5)
      integer m,n
      real a
      dimension a(mmax,nmax)
      open(unit=11,file='texto3.dat',status='old')
      open(unit=12,file='texto3.res',status='new')
      read(11,10)m
      write(12,*)m
      read(11,15)n
      write(12,*)n
 10   format(/,i1)
 15   format(/,i1,/)
c...Hemos leido las dimensiones de la matriz

      do i=1,m
         read(11,20)(a(i,j),j=1,n)
 20      format(<n>e7.1)
```

```
      enddo
      do i=1,m
        do j=1,n
           write(12,30)i,j,a(i,j)
 30        format('a(',i2,',',i2,')=',e12.5)
        enddo
      enddo
      end
```

¿Qué cosas nos pueden hacer la puñeta en esta ocasión? Primero, el 15 format(/,i1,/), el segundo slash nos permite saltar Matriz, y como que cada vez que se hace un read hace un salto de línea automáticamente al acabarlo, ya empezamos a leer los elementos de la matriz a continuación. Lo segundo que nos la puede liar es no tener cuidado con el tamaño del formato de lo que estamos leyendo. El do implícito es la única forma para poder leer-escribir datos en línea, y el formato que debe acompañar es el de los n-valores que se imprimen en una línea , esto es:format(<n>e7.2), en vez de format(e7.2) como podríamos pensar. Recordar que el primer dígito después de la 'e (simple precisión), o la d (doble precisión), es el número total de digitos a leer, con signo y cero de mantisa, y letra, signo y valor del exponente, y el segundo el numero de dígitos de la mantisa. Cuidado con esto último, y con espacios entre valores al leer, y al escribir, que haya espacio para cero, signo, mantisa, etc según configuración.

Afortunadamente la mayoría de los problemas que nos encontramos en C.N. no tendremos que leer matrices así. Esto es una apreciación muy subjetiva: cuando trabajas con Open-VMS, los ficheros de texto tienen un límite de caracteres en horizontal, que no es mayor de un 6d15.7, con lo que si la matriz que intentamos introducir es muy grande en esa dimensión es inviable escribirla en el fichero de datos así; por otra parte, si como yo utilizas el compilador g77 del proyecto GNU[1] sobre Linux, puedes casi meter las columnas que te de la gana (yo he probado hace tiempo, si mal no recuerdo, hasta 50d15.7 o más), sin embargo no está implementado en el compliador el formato variable < >. Para imprimir sí que podría ser útil, ya que podríamos v.g. coger la tremenda matriz 50 puntos en x,200 en t, en un problema de diferencias finitas, imprimirla tal cual en un fichero, y tomarla cómodamente

[1] http://www.gnu.org

en una hoja de cálculo o similar para hecer gráficas. Para Linux hay la solución chapucera de meter una constante que vaya sobrada (`60d15.7` valdría en este último ejemplo), pero esto sólo es válido para escribir, porque al leer de esta manera también protesta. Intentar ejecutar esto en Open-VMS es encontrarse con la muerte automática del proceso [2].

En definitiva, si tienes que trabajar en Open-VMS, como buen estudiante de Caminos-UDC, no te queda más narices que plantear la recepción e impresión de de datos en n-columnas en vertical. Y de cualquier modo, el formato libre de lectura siempre es más fácil de hacer que funcione aunque sea menos elegante.

Volviendo al ejemplo original, daría este resultado:

```
 2
 3
a( 1, 1)= 0.10000E-01
a( 1, 2)= 0.30000E+00
a( 1, 3)= 0.20000E+00
a( 2, 1)= 0.40000E+01
a( 2, 2)= 0.30000E-02
a( 2, 3)= 0.50000E+01
```

*Problema 3.1: Escribir un programa que lea una matriz mxn en un fichero de resultados en el que se lea primero las dimensiones del problema y después las componentes de la matriz colocadas por columnas, v.g.:*

```
m
2
n
3
matriz
```

[2] ya se que recurro mucho a la palabra muerte, para el lector poco experimentado, aclaro que no tiene que ver con algún instinto asesino, sino con el comando de Unix `kill` que sirve para cortar un proceso

```
.230e+01
```
*(elemento 1,1)*

```
.562e-01
```
*(elemento 2,1)*

```
.283e-02
```
*(elemento 1,2)*

...

*Imprimir la matriz en ternas de columnas, la primera, la posición i que ocupa el elemento, la segunda la posición j, y la tercera el valor del elemento, expresando cada terna una columna de la matriz*

*Problema 3.2: Escribir un programa que lea una matriz de tal forma que primero lea dimensiones y el número de elementos distintos de cero, y a continuación dichos elementos, uno en cada línea, precedido por la posición que ocupan en la matriz, v.g.:*

```
m
2
n
3
elementos distintos de cero:
3
i j  a(i,j)
```

`3 2 .230e+01` *(elemento 3,2)*

`1 1 .562e-01` *(elemento 1,1)*

`1 2 .283e-02` *(elemento 1,2)*

*Imprimir la matriz con "pinta de matriz"*

# Capítulo 4

# Algunos consejos para diseñar algoritmos C.N.

Ahora ya nos hemos familiarizado con todo lo básico que necesitamos para ir haciendo cosas más serias. A continuación vamos a trabajar un poco el diseño de algoritmos, que es algo donde la gente suele flojear. En el caso de Caminos-UDC, si uno tiene unos cuantos libros y apuntes a mano le puede bastar para *sobrevivir* modificar alguno de los algoritmos que éstos contienen, sin embargo en el mundo real no todo está en los libros, y haciendo gala al nombre de nuestra carrera, tenemos que ingeniárnoslas para salir al paso (además, uno no sabe cuando se le va a *cruzar el cable* al profesor en un examen...)

Creo que un buen ejemplo para empezar es la resolución de un sistema de ecuaciones triangular inferior. Lo primero, es resolvera mano un problema pequeño, pero lo **suficientemente grande como para que incluya todos los condicionantes**. En este caso, es fácil, nos basta con una matriz 3x3.Supongamos $a_{ij}$ los elementos de la matriz, $v_i$ los del vector de incógnitas, y $b_i$ los del de términos independientes. Entonces tenemos:

$$a_{11} \cdot v_1 = b_1 \tag{4.1}$$

$$a_{21} \cdot v_1 + a_{22} \cdot v2 = b_2 \tag{4.2}$$

$$a_{31} \cdot v_2 + a_{32} \cdot v2 + a_{33} \cdot v3 = b_3 \tag{4.3}$$

de donde deducimos:

$$v_1 = \frac{b_1}{a_{11}} \tag{4.4}$$

$$v_2 = \frac{b_2 - a_{21} \cdot v_1}{a_{22}} \tag{4.5}$$

$$v_3 = \frac{b_3 - (a_{31} \cdot v_1 + a_{32} \cdot v_2)}{a_{33}} \tag{4.6}$$

A continuación podríamos dar una vuelta de tuerca más:

$$v_1 = \frac{b_1}{a_{11}} \tag{4.7}$$

$$v_2 = \frac{b_2 - a_{21} \cdot \frac{b_1}{a_{11}}}{a_{22}} \tag{4.8}$$

$$v_3 = \frac{b_3 - (a_{31} \cdot \frac{b_1}{a_{11}} + a_{32} \cdot \frac{b_2 - a_{21} \cdot \frac{b_1}{a_{11}}}{a_{22}})}{a_{33}} \tag{4.9}$$

como vemos, las expresiones se complican más. ¡Pero es que no es necesario! Si implementásemos (5),(6),(7) la máquina obtendría $v_1$, y con ese valor y los que ya tenemos que son dato le basta operar (6) para obtener $v_2$; y de ahí $v_3$ en (7).¡¡Que trabaje la máquina, narices!!

Pero además no vamos a ir implementando línea a línea, imaginémonos que la matriz fuera 100x100, sería intratable. Entonces debemos desprender del ejemplo (de eso se trata) una expresión o conjunto de expresiones básicas que hagan variar los subíndices para obtener las incógnitas. En este caso observamos que las incógnitas $v_i$ son fracciones cuyo denominador es el elemento $a_{ii}$ correspondiente, el numerador es una diferencia entre el elemento $b_i$ correspondiente y un sumatorio: en el primer caso, es 0 directamente, por lo que debemos programarlo aparte; y a partir de $v_2$, es $\sum_{j=1}^{i-1} a_{ij} \cdot vj$. En definitiva, la expresión queda así:

$$v_1 = \frac{b_1}{a_{11}} \tag{4.10}$$

$$v_i = \frac{b_i - \sum_{j=1}^{i-1}(a_{ij} \cdot v_j)}{a_{ii}} \tag{4.11}$$

Programamos el primer elemento para no crear una inconsistencia en los extremos del sumatorio, siendo menor el final que el inicial.

Ya tenemos lo más importante, el algoritmo de resolución, pero en *abstracto*. Ahora tendremos que particularizarlo para el almacenamiento adecuado

para una matriz de estas características, ya que es obvio que no vamos a almacenar todos los ceros. Una forma típica es almacenando la matriz en un vector por filas: $k = (k1, k2, ...kn) = (a11, a21, a22, a31, ...)$. Según esto, un elemento $a_{ij}$ se encontraría en el elemento $k_p$, con $p = 1+2+...+(i-1)+j = (\sum_{n=1}^{i-1} n) + j$, y de nuevo, con el caso particular $a_{11} = k_1$. Esto, expresado en código Fortran, quedaría más o menos así:

```
      program sis_tr_inf
      integer nmax,i,j,p,pmax
      parameter(nmax=10)
      real*8 v,b,k,sum
      dimension v(nmax),b(nmax),k(55)
      open(unit=11,file='trianga.dat',status='old')
      open(unit=12,file='triangb.dat',status='old')
c...almacenamos la matriz en un fichero y el vector de terminos
c...independientes en otro
      open(unit=13,file='triang.res',status='new')
      read(11,10)n
 10   format(/,i2,/)
c...leemos la dimension del problema
      if(n.gt.nmax)then
        write(6,*)'memoria insuficiente'
        stop
      endif

      pmax=0
      do i=1,n
         pmax=pmax+i
      enddo
c...calculamos el numero de componentes que vamos a leer

      do i=1,pmax
         read(11,20)k(i)
      enddo
      do i=1,n
         read(12,20)b(i)
      enddo
 20   format(d15.7)
```

```
c...comenzamos a obtener los resultados

      v(1)=b(1)/k(1)
      do i=2,n
         p=0
         do j=1,i-1
           p=p+j
         enddo
c...obtenemos la posicion del ultimo elemento de k antes de la fila i
c...correspondiente

         sum=.0d+00
         do j=1,i-1
           sum=sum+k(p+j)*v(j)
         enddo
c...operamos con un bucle el sumatorio antes de restatrselo a b(i)

         v(i)=(b(i)-sum)/k(p+i)
      enddo

      do i=1,n
        write(13,30)i,v(i)
 30     format('v(',i2,')=',d10.3)
      enddo
      end
```

Con estos ficheros de datos:
trianga.dat

```
n
3
a
-.10d+0
-.23d+0
 .15d+0
 .82d-0
-.35d+0
```

```
 .47d-0
```

triangb.dat

```
.47d+0
.33d+0
.13d-0
```

tendremos estos resultados:
triang.res

```
v( 1)=-0.470E+01
v( 2)=-0.501E+01
v( 3)= 0.475E+01
```

Un error frecuente a la hora de testar programas o bloques de programas es el meter datos a lo loco. Esto, además de generar problemas de convergencia en métodos iterativos, de diferencias, etc., ya de por si pueden provocar que la matriz esté mal condicionada con lo que pueden dar resultados raros estando bien programado. Por ello es importante procurar testar con matrices bien condicionadas y datos con sentido físico.

*Ejercicio 4.1: Escribir un programa que resuelva un sistema triangular en banda superior, para un ancho de banda variable. Por supuesto, no se almacenarán los ceros por debajo de la diagonal principal ni por encima de la última diagonal no nula, con lo que el almacenamiento será el habitual en estos casos:una matriz 3xn en la que se almacena una diagonal por columna. Obsérvese que hay dos pautas de comportamiento, la de las filas iniciales-centrales, y la de las filas finales*

Otro ejemplillo, este bastante más frecuente que el resuelto anterior: Sea un sistema en banda simétrico. Podemos descomponer la matriz del sistema A en dos matrices L y su traspuesta, de tal manera que:

$A \cdot v = b$
$(L \cdot L^t) \cdot v = b$
$L^t \cdot v = x$
$L \cdot x = b$
Resolveremos pues, $x$ en primer lugar, y con éste tendremos $v$. Pero no me preocupa la solución del sistema, porque ya hemos empezado a prepararla haciendo el ejercicio 4.1¿¿¿verdad??? No, en lo que nos vamos a fijar es en el algoritmo de factorización, en este caso conocido como factorización de Cholesky.

Tomemos el ejemplo de una matriz tridiagonal, por supuesto, simétrica, ya que de lo contrario no habría solución real. La factorización de Cholesky, al igual que las LU, o las LDU sabemos que respetan el perfil de la matriz, por lo que sólo tenemos que calcular los elementos correspondientes, en este caso, a diagonal principal y a su inmediata inferior. Pongamos como ejemplo una matriz 4x4, en ella se ven bien las excepciones que tendremos que gestionar en el algoritmo. De $A = L \cdot L^t$, tenemos:

$$a_{11} = l_{11}^2 \tag{4.12}$$

$$a_{21} = l_{21} \cdot l_{11} \tag{4.13}$$

$$a_{22} = l_{21}^2 + l_{22}^2 \tag{4.14}$$

$$a_{32} = l_{22} \cdot l_{32} \tag{4.15}$$

$$a_{33} = l_{32}^2 + l_{33}^2 \tag{4.16}$$

$$a_{43} = l_{33} \cdot l_{43} \tag{4.17}$$

$$a_{44} = l_{43}^2 + l_{44}^2 \tag{4.18}$$

De estas ecuaciones deducimos:

$$l_{11} = \sqrt{a_{11}} \tag{4.19}$$

$$l_{21} = \frac{a_{21}}{l_{11}} \tag{4.20}$$

$$l_{22} = \sqrt{a_{22} - l_{21}^2} \tag{4.21}$$

$$l_{32} = \frac{a_{32}}{l_{22}} \tag{4.22}$$

$$l_{33} = \sqrt{a_{33} - l_{32}^2} \tag{4.23}$$

$$l_{43} = \frac{a_{43}}{l_{33}} \tag{4.24}$$

$$l_{44} = \sqrt{a_{44} - l_{43}^2} \tag{4.25}$$

De nuevo obsevamos que haciendo los cálculos con orden, calculando en este caso la triangular inferior, siendo consistente con los índices, se encadena una secuencia fácilmente progamable. Ahora no nos resulta complicado extraer las expresiones para programar. El primer elemento es una excepción:

$$l_{11} = \sqrt{a_{11}} \tag{4.26}$$

Después ($i > 1$) vemos que hay dos secuencias bien definidas, la de la diagonal inferior:

$$l_{i,i-1} = \frac{a_{i,i-1}}{l_{i-1,i-1}} \tag{4.27}$$

y la diagonal principal:

$$l_{i,i} = \sqrt{a_{i,i} - l_{i,i-1}^2} \tag{4.28}$$

No debería ser necesario decir, pero lo diré por si acaso, que estas expresiones (28) y (29) deben ir juntas en el mismo bucle, es que siempre hay un despistado que hace un bucle `i=2,n` con una expresión y a continuación hacen otro con la otra, con lo que la máquina se vuelve loca al no tener los datos en el orden encadenados adecuadamente.

Ahora debemos particularizar las expresiones para un almacenamiento adecuado. Pensemos en que A y L están típicamente almacenadas en matrices (nx2). Mantenemos el orden $i$, pero según $j$ sea igual a $i-1$ o a $i$ lo almacenamos en la primera o en la segunda columna:

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & \dots & \dots & \dots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & \dots & \dots & \dots & 0 \\ 0 & a_{32} & a_{33} & a_34 & 0 & \dots & \dots & 0 \\ 0 & 0 & a_{43} & a_{44} & a45 & 0 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & a_{m-2,m-3} & a_{m-2,m-2} & a_{m-2,m-1} & 0 \\ 0 & \dots & \dots & \dots & 0 & a_{m-1,m-2} & a_{m-1,m-1} & a_{m_1,m} \\ 0 & \dots & \dots & \dots & 0 & 0 & a_{m,m-1} & a_{m,m} \end{bmatrix}$$

la almacenamos en:

$$\begin{bmatrix} 0 & a_{11} \\ a_{21} & a_{22} \\ a_{32} & a_{33} \\ \dots & \dots \\ a_{m-2,m-3} & a_{m-2,m-2} \\ a_{m-1,m-2} & a_{m-1,m-1} \\ a_{m,m-1} & a_{m,m} \end{bmatrix}$$

Así pues:

```
l(1,2)=sqrt(a(1,2))
do i=2,n
    l(i,1)=a(i,1)/l(i-1,2)
    l(i,2)=sqrt(a(i,2)-l(i,1)*(i,1))
enddo
```

Además, es bastante habitual que una vez factorizada la matriz no es necesario conservarla, y como una vez utilizado cada elemento ya no se necesita más para los siguientes operaciones, se puede almacenar L sobre A. Así quedaría entonces:

```
a(1,2)=sqrt(a(1,2))
do i=2,n
    a(i,1)=a(i,1)/a(i-1,2)
    a(i,2)=sqrt(a(i,2)-a(i,1)*(i,1))
enddo
```

Y ya lo tenemos, ahora no hay más que resolver los sistemas triangular inferior y superior para obtener la solución del problema. El programa completo, que sirve de comprobación de que el algoritmo está bien, queda propuesto.

Un último ejemplo, factorización de la matriz de un sistema en banda no simétrico con dos diagonales por debajo de la principal y una por encima, esto es, $A = L \cdot U$. Para que el número de ecuaciones e incógnitas se adecuado, todos los elementos de la diagonal principal de lamatriz U tendrán valor

1. Tomaremos como ejemplo una matriz 5x5. Así, realizando el producto e igualando:

$$a_{11} = l_{11} \mid a_{12} = l_{11} \cdot u_{12} \tag{4.29}$$

$$a_{21} = l_{21} \mid a_{22} = l_{21} \cdot u_{12} + l_{22} \mid a_{23} = l_{22} \cdot u_{23} \tag{4.30}$$

$$a_{31} = l_{31} \mid a_{32} = l_{31} \cdot u_{12} + l_{32} \mid a_{33} = l_{32} \cdot u_{23} + l_{33} \mid a_{34} = l_{33} \cdot u_{34} \tag{4.31}$$

$$a_{42} = l_{42} \mid a_{43} = l_{42} \cdot u_{23} + l_{43} \mid a_{44} = l_{43} \cdot u_{34} + l_{44} \mid a_{45} = l_{44} \cdot u_{45} \tag{4.32}$$

$$a_{53} = l_{53} \mid a_{54} = l_{54} \cdot u_{34} + l_{54} \mid a_{55} = l_{54} \cdot u_{45} + l_{55} \tag{4.33}$$

Parece claro que se producen excepciones en los dos primeros grupos de ecuaciones (30) y (31) y en el último(34), por lo que serán programados aparte, mientras que (32) y(33) parecen seguir una pauta clara en el movimiento de índices. No caigamos en la tentación de confiarnos en ver la excepción por encima de la primera línea en que se resuelve el mismo número de incógnitas que el resto de la faja central de la matriz, en este caso 4: en un caso general, como el del ejercicio 4.4 podemos tener excepciones dependiendo de las relaciones entre los anchos de banda horizontal y vertical, lo que detectaremos al ver que aparecen o desaparecen términos en las ecuaciones.

Resolvemos, pues:

$$l_{11} = a_{11} \mid u_{12} = \frac{a_{12}}{l_{11}} \tag{4.34}$$

$$l_{21} = a_{21} \mid l_{22} = a_{22} - l_{21} \cdot u_{12} \mid u_{23} = \frac{a_{23}}{l_{22}} \tag{4.35}$$

$$l_{31} = a_{31} \mid l_{32} = a_{32} - l_{31} \cdot u_{12} \mid l_{33} = a_{33} - l_{32} \cdot u_{23} \mid u_{34} = \frac{a_{34}}{l_{33}} \tag{4.36}$$

$$l_{42} = a_{42} \mid l_{43} = a_{43} - l_{42} \cdot u_{23} \mid l_{44} = a_{44} - l_{43} \cdot u_{34} \mid u_{45} = \frac{a_{45}}{l_{44}} \tag{4.37}$$

$$l_{53} = a_{53} \mid l_{54} = a_{54} - l_{53} \cdot u_{34} \mid l_{55} = a_{55} - l_{54} \cdot u_{45} \tag{4.38}$$

Ahora, lo programamos, almacenando sobre genéricas L y U:

```
...(las mismas operaciones de (35) y (36), tal cual)
do i=3,n-1
  l(i,i-2)=a(i,i-2)
  l(i,i-1)=a(i,i-1)-l(i,i-2)*u(i-2,i-1)
  l(i,i)=a(i,i)-l(i,i-1)*u(i-1,i)
  u(i,i+1)=a(i,i+1)/l(i,i)
enddo
...(las mismas operaciones de (39))
```

Afinamos un poco más: almacenamos sobre la propia matriz A(nx4):

```
...(las mismas operaciones de (35) y (36), tal cual)
do i=3,n-1
  do j=2,3
     a(i,j)=a(i,(j)-a(i,j-1)*a(i,j)
  enddo
  a(i,4)=a(i,4)/a(i,3)
enddo
...(las mismas operaciones de (39))
```

almacenando las bandas inferiores en la primera y segunda columna, la principal en la tercera y la superior en la cuarta. Hemos eliminado una asignación innecesaria, y reconvertido dos ecuaciones en un segundo bucle anidado.

*Ejercicio 4.2: Plantear la factorización LU de la matriz de un sistema de ecuaciones completa.*

*Ejercicio 4.3: Plantear la factorización de Cholesky de la matriz de un sistema de ecuaciones en sky-line.*

*Ejercicio 4.4: Plantear la factorización LDU de la matriz de un sistema de ecuaciones en banda no simétrico de anchos de banda variables v y h. Los elementos de la diagonal principal de las matrices L y U tienen valor 1, D es diagonal.*

# Capítulo 5

# Modularización

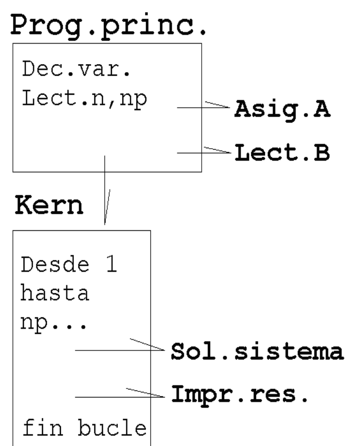
Hasta ahora hemos escrito programas relativamente sencillos. Programas que calculan productos de matrices, límites, sistemas... Pero todos ellos van encaminados a ser piezas de un puzzle mayor que van a ser los programas que nos servirán para resolver problemas mayores, como obtención de raíces de funciones, integración numérica, diferencias finitas, elementos finitos... y todo lo que se nos ocurra. Ahora ya es cuestión de cada uno fabricar las piezas que necesite y ensamblarlas adecuadamente.

Llegados a este punto, si todo lo que hemos hecho atrás es correcto, no deberíamos de tener demasiados problemas en construir programas más serios. Sólo tenemos que readaptar los módulos para formar parte de entidades mayores, lo que si se hace con cuidado no debe de ser ningún problema.

Pongamos por ejemplo el típico problema académico de diferencias finitas: obtenemos valores en un dominio de una ecuación diferencial haciendo una discretización de dicho dominio y proporcionando unas condiciones de contorno y/o iniciales. Al final, todo se reduce a resolver un sistema de ecuaciones cuyo término independiente va cambiando con cada paso en la discretización de tiempo. Y es ésto lo primero que tenemos que tener claro. Partiendo de esta idea, vamos dividiendo el programa en pedazos más pequeños: una entrada de datos, un módulo de construcción de la matriz del problema, otro de factorización si es ese el método empleado, un módulo de iteraciones de tiempo, que llamará en cada paso a otro que genere el nuevo término independiente, una salida de resultados... nos hacemos un esquema del programa, escribimos los módulos de *más bajo nivel*, esto es, i/o, factorización,... y luego, una vez que comprobamos que funcionan correctamente, los ensamblamos sobre los módulos que llaman a estos...

Empecemos, como siempre, con un ejemplo: vamos a resolver un sistema de ecuaciones para diferentes términos independientes. Pongamos que la matriz del sistema es triangular inferior en banda. El orden del problema, el número de problemas a resolver y los términos independientes se leerán en un fichero de datos, la matriz nos la inventaremos y asignaremos valores dentro del programa, para simplificar.

Se me ocurre este primer esquema, aunque como siempre, esto es algo personal y por tanto no tiene por qué ser el único. Con el tiempo y la práctica uno aprende a buscar soluciones no mejores ni peores, sino con uno u otro estilo. Pero lo que es importante, ante todo, es hacer un **programa legible**, esto es, módulos ni muy grandes como para que el depurarlos sea un lío, ni muy pequeños como para que cueste ver la estructura global del programa. A *freaks* como yo, con una más que deplorable vida social nos resulta divertido escribir un programa ahorrando el mayor número de líneas posible, pero es eso, un pasatiempo, pero cuando se va a usar en serio es mejor que cualquiera que lo coja pueda comprenderlo inmediatamente, a costa de extender el código fuente. A la vez, por supuesto, cuanto menor número de operaciones realice mejor, y eso sí es objetivo.




(Ya se que esto no se parece demasiado a los diagramas de flujo de los libros técnicos, pero para empezar esto bien vale...)

Bien, tenemos entonces un programa principal donde se declara la cantidad de memoria a usar en forma de variables simples y formaciones, y llama a una subrutina para leer los datos, y escribe los resultados que obtiene de una llamada a otra subrutina que conforma el cuerpo central del programa, donde se efectúan las operaciones, o al menos, donde se organizan. De ese bloque se harán llamadas al bloque de solución del sistema y al de impresión de la solución, una vez por sistema a resolver. Podríamos almacenar todas las soluciones e imprimirlas juntas fuera del bucle, pero sólo si es necesario, si no, estamos usando memoria innecesaria.

En F77 hay dos tipos de módulos, las subrutinas y las funciones. Las subrutinas son pedazos de programa que comparten una serie de variables; las funciones, como su propio nombre indica, son códigos que reciben un/os dato/s de entrada y devuelven un dato de salida. Ejemplos de funciones que ya suelen venir implementadas con los compiladores son `sin( ), dfloat()`,... y podemos crear las nuestras propias, no tenemos más que enlazarlas compiladas al código del programa en el *linkado*. Las subrutinas, como ya he dicho, comparten variables entre ellas, y recalco lo de comparten porque como ya indiqué en el capítulo 2, los pasos se hacen por referencia, al contrario que, v.g. los módulos del lenguaje Pascal, que pueden hacer pasos por valor exclusivamente.

El efecto colateral del paso por referencia es que un cambio en cualquier subrutina, tendrá consecuencias inmediatas sobre la variable en todo el programa. Por ello tendremos que tener cuidado con las operaciones que hacemos sobre ellas, reiniciarlas si es preciso, etc...

Parece de perogrullo, pero no me quedo tranquilo si no recuerdo que de una llamada a otra el cambio de nombre no importa, pero sí es fundamental el orden y que el tipo de variable declarado en una y otra sea el mismo. De esto último algunos compiladores no avisan por defecto. En cuanto a los arrays como veremos no importa la dimensión, ya que cuando se llama a la subrutina lo único que se pasa es la dirección de memoria del primer elemento, por defecto, o el que se indique. Al final del capítulo vermos como sacar partido de esta forma de comunicación.

También creo que debo comentar que existe una declaración `common` que evita la escritura de todo el lote de variables de una subrutina a la que se llama. La expresión `common /nombredegrupo/listavariables` sirve para declarar un grupo de variables que comparten todas las subrutinas en las que se inserta. Además indica al compilador que estas variables permanezcan lo

mejor disponibles posible, para acceder inmediatamente a la memoria caché. La verdad no es que haya utilizado esto mucho, la única ventaja que veo es la de la mejora de velocidad de acceso desde el procesador, porque me parece más limpio escribir todos los argumentos.

Empecemos con el programa principal. No importa escribir todas las variables en la declaración de entrada, se van añadiendo según hagan falta,lo importante es que estén cuando se compile:

```
      program sol_sis
      integer nmax,n,npmax,np
      parameter(nmax=10,npmax=4)
      real a,v,b
      dimension a(nmax,2),v(nmax),b(nmax,npmax)
c...npmax es el numero de problemas a resolver, nmax la dimension
c...del problema;v es el vector solucion, que sera reescrito cada vez
c...que se resuelve un sistema, b es una matriz que guarda todos los
c...terminos independientes de los problemas a resolver

      open(unit=11,file='solsis.dat',status='old')
      open(unit=12,file='solsis.res',status='new')

      read(11,10)n
      read(11,11)np
 10   format(/,i2)
 11   format(/,i2)
      call matriz_b(n,np,b)
      call matriz_a(n,a)
      call kern(n,np,a,v,b)
      end
```

Como vemos, el programa principal es muy simple: declaraciones generales, apertura de ficheros, lectura de datos, en este caso a través de dos subrutinas, y llamada a una subrutina cuyo nombre lo dice todo:`kern`, de kernel(núcleo)

```
      subroutine matriz_b(n,np,b)
c...recuerda argumentos en el mismo orden y de la misma precision
      integer n,np,i,j
      real b
      dimension b(n,np)
```

```
      do i=1,np
         read(11,20)b(1,i)
 20      format(/,e10.3)
c...leemos el primer elemento aparte para saltar con
c...el el titulo 'pi'
         do j=2,n
            read(11,21)b(j,i)
 21         format(e10.3)
         enddo
      enddo
      end
```

Lectura de datos, lo de siempre...

```
      subroutine matriz_a(n,a)
      integer n,i
      real a
      dimension a(n,2)
      a(1,2)=.1e+01
      do i=2,n-1
         a(i,1)=.3e+01
         a(i,2)=.2e+01
      enddo
      a(n,1)=.4
      a(n,2)=.7
c...hemos almacenado la diagonal inferior en la primera
c...columna, y la principal en la segunda
      end
```

Montaje de la matriz del problema. En otros caso se podría organizar con una llamada desde la subrutina central (kern). Cuidado con los despistes: el elemento $a_{11}$ se almacena en `a(1,2)`

```
      subroutine kern(n,np,a,v,b)
      integer n,np,i
      real a,v,b
      dimension a(n,2),v(n),b(n,np)

      do i=1,np
```

```
          call solsis(n,a,v,b(1,i))
c...enviamos a solsis la direccion de memoria del primer
c...elemento de cada columna de b, asi solsis lo toma como un vector
c...y el algoritmo es mas sencillo
          call impr_res(n,v,i)
      enddo
      end
```

En esta subrutina es donde realmente uno tiene que demostrar que sabe lo que está haciendo, organizando adecuadamente los ciclos de operaciones y las llamadas a subrutinas con los valores adecuados. Modularizando adecuadamente, es fácil tener una visión global, *"divide y vencerás"*...

```
      subroutine solsis(n,l,v,b)
      integer n,i
      real l,v,b
      dimension l(n,2),v(n),b(n)
      v(1)=b(1)/l(1,2)
      do i=2,n
          v(i)=(b(i)-l(i,1)*v(i-1))/l(i,2)
      enddo
      end
```

Así es como se hacen las cosas: llamamos a una subrutina que resuelve un sistema. Le damos los datos adecuados, y sólo nos debemos de preocupar dentro de la propia subrutina de como funciona.

```
      subroutine impr_res(n,v,nprob)
      integer n,nprob,i
      real v
      dimension v(n)
      write(12,*)'Problema numero ',nprob
      do i=1,n
        write(12,300)i,v(i)
 300    format('v(',i2,')',e10.3)
      enddo
      end
```

Y finalmente una salida de datos sencilla...

El fichero de datos:

```
n
4
numero problemas
3
    p1
 0.121e+1
-0.165e+1
 0.436e+0
 0.203e+1
  p2
-0.743e+0
 0.157e+1
 0.000e+0
 0.653e+0
  p3
-0.245e+1
 0.941e+0
-0.702e+0
 0.000e+0
```

...y el de resultados:

```
 Problema numero  1
v( 1) 0.121E+01
v( 2)-0.264E+01
v( 3) 0.418E+01
v( 4) 0.513E+00
 Problema numero  2
v( 1)-0.743E+00
v( 2) 0.190E+01
v( 3)-0.285E+01
v( 4) 0.256E+01
 Problema numero  3
v( 1)-0.245E+01
v( 2) 0.415E+01
v( 3)-0.657E+01
```

```
v( 4) 0.375E+01
```

Si nos hemos fijado un poco, la declaración de formaciones en el programa principal y en las subrutinas a las que llama es diferente. Pero lo importante es que sean iguales entre las que montan las matrices con los datos y las que tarabajan con ellas. ¿Por qué? Hemos ido dando pistas durante todo el documento. Si recordamos, una formación tenga la dimensión que tenga es almacenada trasponiendo las columnas y colocándolas una detrás de otra en línea. Pongamos como ejemplo la matriz b, que es (nmax x npmax) en el programa principal, y (n x m) en la subrutina de montaje. Pero el espacio de memoria es el mismo. Siguiendo el almacenamiento que realiza el programa, para el ejemplo particular resuelto:

**Dimensionamiento normal**

Punto vista programa principal: b(10x4)

| b11 | b21 | b31 | b41 | b51 | b61 | ... | b22 | b32 | b42 | b52 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| b11 | b21 | b31 | b41 | b12 | b22 | ... | b43 | 0 | 0 | 0 | ... |

Almacenado/interpretado por subrutinas:
b(nxnp)n=4,np=3

Si imprimimos desde la subrutina la matriz almacenada, utilizando el anidado típico `i=1,n, j=1,np` tendremos lo que esperamos, la matriz con *pinta de matriz*; pero si hacemos lo mismo en el programa principal, obtendremos:

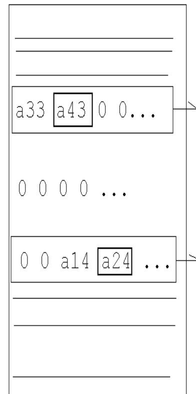
```
b11 b33 0
b21 b43 0
b31 b14 0
b41 b24 0
```

¿Por qué es esto? También dimos la pista: cuando enviamos de un módulo a otro una formación, la que recibe, toma una dirección de memoria perteneciente a esa formación, por defecto el primero, pero se puede indicar, como se ve en el ejemplo. A partir de ahí, organiza las variables según la propia declaración de la subrutina, ocupando, en este caso, las primeras n x np variables. Adapta a esta ordenación su propia estrategia de búsqueda, esto es, necesito un elemento de la formación, voy al primer elemento, de ahí al primero de la columna a la que pertenece, según la propia ordenación de la subrutina, y

de ahí al elemento requerido.

Ello nos lleva a la conclusión de que a pesar de que el espacio de memoria ocupado por el programa es constante desde el momento de la compilación[1], intuitivamente no parece la mejor forma de almacenamiento, matriz *esponjosa*, llena de grupos de ceros entre columna y columna. La causa es la siguiente: el procesador se va tomando líneas enteras de las páginas de memoria ocupadas por el programa, para su propia caché, en vez de tomar las componentes de una en una, porque al tratarse de formaciones se supone que a continuación se utilizarán las componentes cercanas. Claro, si incluímos ceros que no se van a usar, en general al menos una parte de ellos entrará en la última línea de una columna antes de buscar la referencia de la siguiente, esto es, su primer elemento, y de ahí el que se precise.

Vemos en esta posible disposición de memoria lo que pasa al coger la línea para leer el elemento a(4,3):tomamos los anteriores elementos de la columna y el siguiente, y los ceros que la completan:




En definitiva, cuanto más **compactos** estén los datos, mejor.

En clase de C.N. aprendimos una técnica para compactar los datos en la memoria. La conocemos con el equívoco nombre de *asignación dinámica de memoria*, cuando sería más propio llamarla *asignación compacta en memoria estática*, por ejemplo. Consiste básicamente en predecir cuanto va a ocupar realmente cada formación *conceptual* y ponerlas una detrás de otra sobre un vector *real*.

[1]al contrario que en todos los lenguajes modernos, que permiten acaparar y liberar memoria en tiempo de ejecución

La gracia del asunto es marcar sobre ese vector la variable exacta donde empieza cada formación *conceptual* y enviar su dirección a la subrutina que corresponda para que ésta interprete ese pedazo de memoria indicado con la pinta de la formación *conceptual*, para así no tener que modificar los algortimos escritos pensando en el almacenamiento *conceptual*. Veamos el ejemplo resuelto como cambiaría:

```
      program sol_sis
      integer nmax,n,np
      parameter(nmax=70)
c...frente a dos parametros que teniamos antes, que nos limitaban
c...el problema en dimension y numero de sistemas a resolver, ahora
c...solo dependemos del total de memoria disponible. Un problema
c...generico ocupara 2xn+n+np+n variables

      dimension work(nmax)
      open(unit=11,file='solsis.dat',status='old')
      open(unit=12,file='solsis.res',status='new')

      read(11,10)n
      read(11,11)np
 10   format(/,i2)
 11   format(/,i2)
      call matriz_b(n,np,work(2*n+1))
c...la matriz b se empezara a almacenar donde acaba a, y como
c...a(2xn)...

      call matriz_a(n,work)
c...a(conceptual) se almacena al principio de work(real)
c...podriamos indicar work(1) pero no es necesario

      call kern(n,np,work,work(2*n+n*np+1),work(2*n+1))
c...y v se almacenara segun se obtenga al final de b
      end
```

Y el resto del programa permanece exactamente igual. Observemos que además de ganar en velocidad, también ganamos en posibilidades: antes podíamos resolver hasta 4 sistemas de 10x10 (70 var.), ahora podemos resolver 20 sistemas de 3x3(69 var.), o 1 sistema 15x15(60 var.)... cualquier combinación

$(3{+}np){\cdot}n \leq nmax$. No debemos olvidar, tanto en programas con dimensionamiento normal y "dinámico", imponer una condición de parada de programa capaz de preveer antes de ralizar ninguna operacion que las exigencias de memoria no puedan cubrirse.

**Con "dimensionamiento dinamico"**

```
Punto de vista de programa principal: w(70)

... w7   w8   w9  w10  w11  w12  w13  ...  w19  w20  w21 ...
|  ...|a3,2| a4,2|b1,1|b2,1|b3,1|b4,1|b1,2|  ...  |b3,3|b4,3| v1 |  ...  |

Almacenado/interpretado por subrutinas: b(nxnp),n=4,np=3
```

*Ejercicio 5.1: Programar un producto de matrices de dimension variable, modularizando la entrada y salida de datos y el algoritmo de producto, aplicando dimensionamiento normal para almacenamiento, y reprogramarlo para almacenamiento con "dimensionamiento dinámico"* [2]

*Ejercicio 5.2: Programar la solución de varios sistemas de ecuaciones tridiagonales no simétricos mediante factorización LDU, de dimensión variable*[3]*. Los elementos de la matriz serán asignados en el propio programa, mientras que se leerá un solo término independiente en el archivo de datos y a partir de ahí se resolverán tantos sistemas como se precise en el el archivo de datos, tomando para cada uno como término independiente la solución del anterior. Utilizar "dimensionamiento dinámico" para el almacenamiento*

[2]este es el problema con el que me lo enseñron a mí, en concreto, el Dr. Ingeniero Fermín Navarrina Martínez... *"al César lo que es del César..."*

[3]este programa tiene una estructura útil para resolver problemas en diferencias finitas

# Capítulo 6

# Consejillos finales para depurar programas

Creo que ya hemos ido comentando la mayoría de los problemas que nos podemos ir encontrando. De todas formas, aún hay más:

· Por un lado tenemos los errores de compilación: aunque nos de un susto encontrarnos de repente con la pantalla llena de errores la primera vez que compilamos un programa, no van a ser los mayores problemas [1]. La forma de solucionarlos, evidentemente, conocer a fondo la sintaxis. Que no nos confundan los mensajes de error sobre la linea en que se encuentra: el compilador interpreta el programa hasta que *ya no puede más*, pero el error puede empezar antes.

· Llevar un control de versiones, aunque no sea con CVS, por lo menos indicar en el código fuente con comentarios número de versión, fecha y observaciones sobre contenidos y modificaciones respecto a anteriores.

· Insertar tabulaciones en las anidaciones de bucles o condiciones, que se vea claramente que instrucciones pertenecen a quien. A veces, errores en anidaciones no son detectados por el compilador si pueden ejecutarse, aunque no sea como pretendemos.

· Vigilar que el tipo de las variables sea el mismo por todos los puntos del programa donde se llamen. Esto no da error, pero sí avisa al *linkar*[2].

·Vigilar el orden de los argumentos en las llamadas a subrutinas. Son útiles para esto las herramientas cortar/pegar de los editores de texto. Si se

[1] por cierto, para usuarios del compilador de Open-VMS, recordar `for /lis=error.lis programa.for` para enviar la lista de errores al fichero `error.lis`

[2] siempre que el linker esté configuarado para ello

intercambian variables incluso aunque sean de tipos diferentes el linker no dará error, sino aviso (*warning*).

· Empezar a probar las subrutinas de bajo nivel primero, e insertar instrucciones de impresión de todos los datos para observar que sean recibidos correctamente cuando son llamadas. Si no es así, ascender en niveles de subrutinas hasta encontrar el error. Es recomendable, para programas muy grandes, véase programas de elementos finitos, establecer nomenclaturas de niveles y grupos para no perderse.

· Si el programa se detiene abruptamente durante la ejecución, habrá que empezar a investigar a que es debido por el mensaje de error: si es por fallos en operaciones, tipo división por cero, o usos de tipos de valores o variables inadecuados; si es por falta de asignación de memoria, tendremos los `segmentation fault` o `memory adress access violation`, y tendremos que empezar a buscar en algún punto donde nos salimos de dimensión; los errores de lectura de datos son fácilmente identificables, ya que indican `format error...` pero para mi son un auténtico suplicio para corregirlos. Si los errores persisten después de una inspección visual del código fuente, debemos ir insertando instrucciones de impresión de mensajes que nos indiquen hasta donde se ejecuta. Recomendable, además, ir imprimiendo variables y resultados de operaciones a cada paso. Habrá que hacerlo, de todas formas, si los resultados no son correctos.

· Como ya he dicho, lo primero es comprobar las subrutinas más básicas, si están bien, entonces es el ensamblaje con el resto del programa. No basta con observar que las operaciones de un bucle se realizan bien una sola vez, ya que éste puede estar mal diseñado y al hacer el paso siguiente dará resultados erróneos.

· Inicializar las variables a 0, especialmente si se repite la ejecución de un trozo de código, ya que puede acumular valores, y ser un desastre.

· Puede pasar que un método esté bien programado, y sin embargo nos de resultados muy extraños esto se puede deber a que los valores que se pretenden solucionar no están dentro del radio de convergencia del problema, así que si es un problema físico se deben usar valores reales. Puede deberse también a una discretización gruesa, en ese caso observaremos normalmente valores que se mantienen en general dentro de un rango pero que oscilan cuando debiera haber una curva suave. Si es debido a la malla, refinando el problema mejora.

· Finalmente, si después de todas las comprobaciones el programa sigue sin funcionar, habrá que pensar que hay un error en el método numérico que

estamos empleando. No sería la primera vez...

Existe una herramienta del proyecto GNU para depurar programas en varos lenguajes, el GDB y su capa para X-Window DDD, no me canso de recomendar que si uno tiene tiempo aprenda a manejarla porque a la larga, ahorra bastante trabajo.

Y hasta aquí hemos llegado. Si se me ocurre algo más, ¡en la siguiente versión!

D.t.y. Otoño 2001

## Bibliografía

Ellis TM.R. *"FORTRAN 77 Programming. With an Introduction to the FORTRAN 90 Standard"*, $2^a$ ed., Addison-Wesley, 1990

Press,W.H., Flannery B.P., Teukolsky,S.A., Vetterling,W.T. *"Numerical recipes in Fortran"* Cambridge University Press, Cambridge,1986

# Capítulo 7

# GNU Free Documentation License

Version 1.2, November 2002
Copyright ©2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free"in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of çopyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used

for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

# 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **ƶou"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **Çover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public,

that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **.ºpaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **.ᴬcknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ.ªccording to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy

will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

# 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History"in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History"section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled .Acknowledgements.or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements.or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the

Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

# 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History"in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled .Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

# 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

# 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an .aggregate"if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

# 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their

copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled .ᴬcknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License .ºr any later version.ªpplies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.