UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

ENFOQUE ALGORÍTMICO PARA EL ALMACENAMIENTO DINÁMICO DE MATRICES DISPERSAS

INFORME DE SUFICIENCIA

PARA OPTAR EL TÍTULO PROFESIONAL DE:

INGENIERO ELECTRICISTA

PRESENTADO POR:

HEINER RICARDO LÓPEZ SANDOVAL

PROMOCIÓN
1997-I

LIMA – PERÚ
2007

ENFOQUE ALGORÍTMICO PARA EL ALMACENAMIENTO DINÁMICO DE MATRICES DISPERSAS

A mis padres:
Flor y Heiner
Por su dedicación, sacrificio y cariño

INDICE

INTRODUCCIÓN

El propósito del presente informe es el de mostrar e implementar técnicas de programación aplicadas a sistemas de potencia, específicamente en lo relacionado con la obtención de la matriz de admitancia en sistemas dispersos, así como la comparación de las estructuras de datos estática y dinámica mediante el diseño, implementación y comparación de programas.

Para lograr dicho fin, se toma como base y se hace énfasis en la comprensión de la técnica de diseño de algoritmos, como medio de representación del proceso de abstracción efectuado al tomar las características principales de un sistema eléctrico y enfocarlas como modelos de software. Adicionalmente a la proposición de algoritmos, se ha efectuado una implementación de los mismos en el lenguaje de programación C.

La metodología empleada para el desarrollo del presente trabajo está fundamentada en los siguientes puntos:

Paso N°1: Planteamiento del problema. Se proporcionan enunciados descriptivos relacionados con sistemas eléctricos.

Paso N°2: Conceptualización del problema.

Paso N°3: Abstracción del problema en un modelo de software que será simbolizado por un pseudocódigo, el cual es la representación del lenguaje algorítmico.

Paso N°4: En casos en los cuales se tenga un pseudocódigo extenso, se procederá a la modularización del mismo, mediante el diseño de procedimientos y funciones.

Paso N°5: Implementación del pseudocódigo en lenguaje C.

Paso N°6: Ejecución del programa implementado, mostrando los reportes pertinentes.

Dado que se hace incidencia en el diseño de programas, así como en la mayor simplicidad posible de los mismos para evitar contenidos extensos, el presente trabajo tiene las siguientes limitaciones:

1. Se trata directamente con parámetros eléctricos expresados en valores por unidad (p.u.).

2. No se ha efectuado el modelamiento generadores, compensadores o máquinas eléctricas. Como se verá más adelante, se trabaja en base a barras de envío, recepción, resistencia (p.u.), reactancia (p.u.) y admitancia shunt (p.u.).
3. Se toma como referencia algorítmica los fundamentos de la programación estructurada.

En el Capítulo I: *NOCIONES BÁSICAS DE ALGORITMOS*, se hace referencia a los conceptos fundamentales de la algoritmia, metodología de análisis de un problema, sentencias de control y decisión, procedimientos y funciones.

En el Capítulo II: *ESTRUCTURAS DE DATOS ESTÁTICAS*, se hace referencia a la forma de empleo de arrays unidimensionales, bidimensionales, caracteres, registros y archivos, así como el tratamiento de números complejos, que son ampliamente utilizados en sistemas de potencia.

En el Capítulo III: *ALMACENAMIENTO DINÁMICO DE MATRICES DISPERSAS*, se proporciona un breve comentario relacionado con las técnicas que permiten el almacenamiento de las matrices dispersas, la descripción del método de almacenamiento de la matriz de admitancias de nodo utilizando listas enlazadas, así como una comparación entre el uso de estructuras estáticas y dinámicas, mediante la generación de la matriz de admitancias de un sistema de 23 barras.

En el Capítulo IV: *INVERSION Y FACTORIZACIÓN DE MATRICES DISPERSAS*, se muestran dos aplicaciones relacionadas con la inversión de la matriz de admitancias mediante el método de Shipley-Coleman y la factorización de la misma mediante el método de Doolittle (LUDM) respectivamente. Se efectúa una descripción de ambos métodos, así como sus respectivos algoritmos.

Finalmente, quiero agradecer quienes hicieron posible la presentación de este trabajo. En primer término deseo mencionar el apoyo brindado por mis superiores en mi centro de trabajo, la Universidad de Lima, en las personas del Dr. Marco Aurelio Zevallos y Muñiz, Eco. Raúl Obregón Pérez y la Ing. Rosario Guzmán Jiménez por todas las facilidades proporcionadas a mi persona desde que se inició el curso de titulación hasta la fecha, así como a mis entrañables amigos, compañeros de trabajo y practicantes con los que departo momentos excepcionales en la Facultad de Ingeniería de Sistemas y el Laboratorio de Aprendizaje en Tecnologías de Información de la Escuela de Ingeniería. Mi agradecimiento al Ing. José Zorrilla Acosta, mi asesor, por sus observaciones y revisiones. Muchas

personas han trascendido en el lapso de tiempo transcurrido desde el curso de titulación, pero quiero hacer mención a mi compañero de estudios, de promoción y ahora ingeniero César Roldán Villasís por su aliento y motivación en momentos claves, a la Familia Vilela Zamora por su ayuda y apoyo en momentos no muy gratos, y en este último trayecto un agradecimiento especial para una excelente amiga, muchas gracias Anabelle por tus consejos.

CAPÍTULO I
NOCIONES BÁSICAS DE ALGORITMOS

1.1 Conceptos fundamentales

1.1.1 Algoritmo

Se entiende por *algoritmo* como la definición o especificación de un número finito de pasos a seguir para resolver un determinado problema. Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema, el algoritmo puede expresarse en un lenguaje de programación distinto y ejecutarse en una computadora distinta; sin embargo, el algoritmo siempre será el mismo. En la Figura 1.1 se muestra el proceso de abstracción implicado en la solución de problemas mediante algoritmos:

Figura 1.1: Proceso de abstracción en la solución de un problema

Ejemplo 1.1: El algoritmo que nos permitiría efectuar la suma de dos fasores, podría incluir los siguientes pasos:

Paso N°1: Obtener fasor **A**

Paso N°2: Obtener fasor **B**

Paso N°3: Sumar los fasores **A** y **B**

Paso N°4: Mostrar la suma de dichos fasores

1.1.2 Datos y tipos de datos

Se entiende por *dato* como la expresión general que describe los objetos con los cuales opera una computadora.

Los algoritmos, así como sus implementaciones utilizando un lenguaje de programación en particular, hacen uso extensivo de los datos para obtener una solución

satisfactoria de un problema dado. Los datos utilizados se disgregan en *tipos de datos*, los cuales a su vez presentan una clasificación genérica tal y como se muestra en la Figura 1.2:

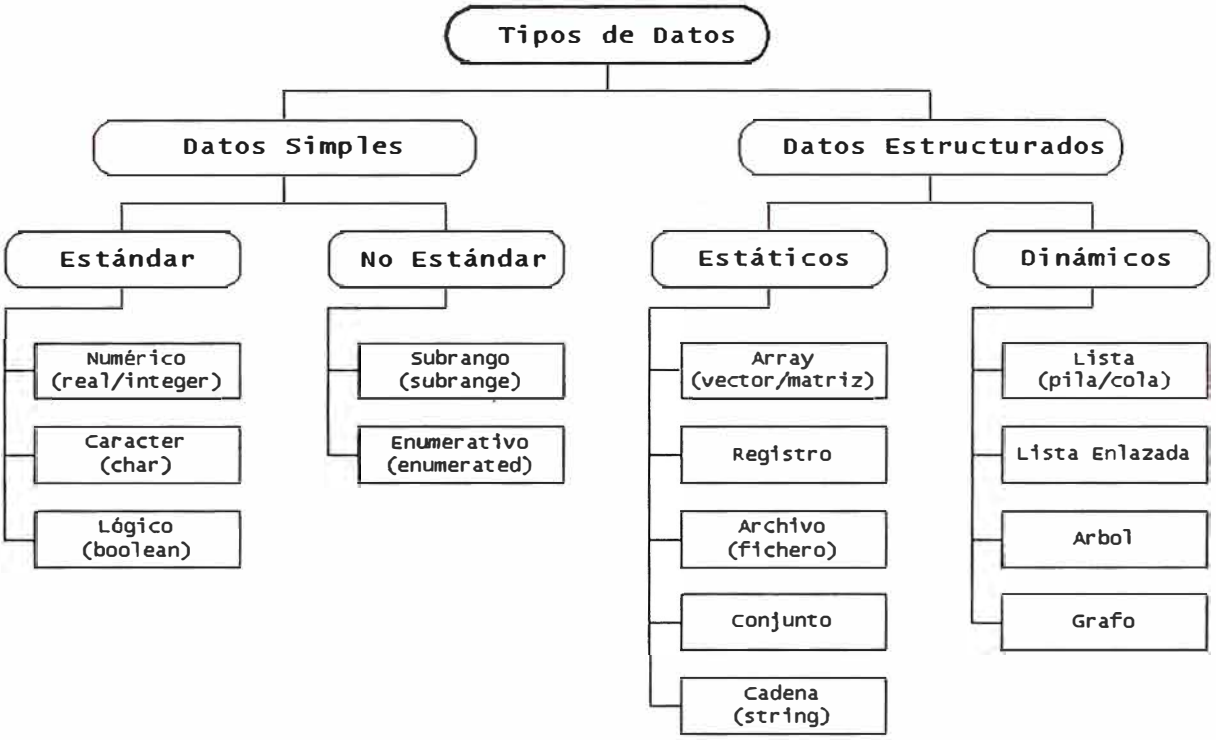


Figura 1.2: Clasificación de los tipos de datos

Los tipos de datos *numérico* y *caracter* son ampliamente utilizados por los lenguajes de programación existentes. Los tipos de datos *lógico*, *subrango* y *enumerativo* son propios de lenguajes estructurados (siendo los dos últimos los que proporcionan la posibilidad de declarar y definir nuevos tipos de datos). Los tipos de datos simples pueden ser organizados en estructuras de datos *estáticas* (aquellas cuyo tamaño se define antes de que el programa que implementa el algoritmo se ejecute) y *dinámicas* (aquellas que no tienen limitaciones en el tamaño de memoria ocupada).

A continuación se definirán tres tipos de datos básicos que serán utilizados en el desarrollo del presente trabajo:

Datos numéricos

El tipo de dato *numérico* es el conjunto de valores numéricos que puede ser representado de dos formas: *tipo numérico entero* y *tipo numérico real*.

Datos lógicos

El tipo de dato *lógico* también denominado *booleano* es aquel que puede tomar uno de dos valores: *verdadero* (true) o *falso* (false).

Datos tipo carácter y tipo cadena

El tipo de dato *caracter* se refiere a un elemento perteneciente a un conjunto de caracteres que la computadora reconoce. Este conjunto no es estándar, sin embargo en general se reconocen ciertos caracteres alfabéticos y numéricos:

- Caracteres alfabéticos: (A, B, …, Z) (a, b, …, z)
- Caracteres numéricos: (0, 1, 2, …, 9)
- Caracteres especiales: (+, -, *, /, ^, ., ;, <, >, $, …)

El tipo de dato *cadena* (string) es una sucesión de caracteres delimitados por comillas.

Ejemplo 1.2: A continuación se muestran algunos datos con sus respectivos tipos:

- -10 Dato de tipo numérico entero
- 'a' Dato de tipo caracter
- Verdadero Dato de tipo lógico
- 9.569 Dato de tipo numérico real
- '3' Dato de tipo caracter
- "pedido" Dato de tipo cadena

1.1.3 Constantes, variables e identificadores

Una *constante* se refiere a un valor fijo que no puede ser alterado por el algoritmo, mientras que una *variable* se refiere a un valor que será alterado durante la ejecución del algoritmo.

En un algoritmo, así como en su posterior implementación utilizando un lenguaje de programación determinado, las constantes y las variables deben tener nombres asociados a lo largo de toda la solución del problema. Los nombres que se utilizan para representar variables, constantes o tipos en un algoritmo son conocidos como *identificadores*. Se crean estos identificadores especificándolos en la declaración de una constante, variable, tipo, procedimiento o función (estos dos últimos términos se explicarán más adelante).

Ejemplo 1.3: Se desea calcular la velocidad angular eléctrica (ω) en función de la frecuencia **f**:

$$\omega = 2\pi f \qquad \textbf{(1.1)}$$

donde:

2π representa una constante, debido a que este factor no va a variar

f representa una variable, debido a que la frecuencia puede ser 50 Hz o 60 Hz.

Se podría esbozar un algoritmo que permita calcular la velocidad angular eléctrica:

Paso N°1: Solicitar el valor de la frecuencia especificada en Hz (f).

Paso N°2: Efectuar la multiplicación 2×π×f

Paso N°3: Mostrar la multiplicación efectuada

1.1.4 Tipos de operadores

Un *operador* es un símbolo que indica que se lleven a cabo ciertas manipulaciones matemáticas o lógicas. Existen tres clases generales de operadores: *aritméticos*, *relacionales* y *lógicos*, los cuales se muestran en la Tabla 1.1.

Tabla 1.1: Operadores Aritméticos, Relacionales y Lógicos

Tipos de Operadores	Operación	Símbolo
Operadores Aritméticos	Suma	+
	Resta	-
	Multiplicación	*
	División	/
	Potenciación	^
Operadores Relacionales	Menor	<
	Mayor	>
	Menor o igual	<=
	Mayor o igual	>=
	Diferente	<>
Operadores Lógicos	Negación	NO
	Intersección	Y
	Unión	O

Adicionalmente a los operadores aritméticos definidos en la Tabla 1.1, se incluyen los operadores *div* (permite obtener el cociente de una división entera) y *mod* (permite obtener el residuo de una división entera). Por otro lado, se tiene un número determinado

de operadores especiales denominados *funciones internas*, siendo las más usuales las que se muestran en la Tabla 1.2:

Tabla 1.2: Funciones internas

Función	Descripción	Tipo de Argumento	Resultado
abs(x)	valor absoluto de *x*	entero o real	Igual que argumento
arctan(x)	arco tangente de *x*	entero o real	Real
cos(x)	Coseno de *x*	entero o real	real
exp(x)	exponencial de *x*	entero o real	real
ln(x)	logaritmo neperiano de *x*	entero o real	real
log10(x)	logaritmo decimal de *x*	entero o real	real
round(x)	redondeo de *x*	real	entero
sen(x)	seno de *x*	entero o real	real
sqr(x)	cuadrado de *x*	entero o real	igual que argumento
sqrt(x)	raíz cuadrada de *x*	entero o real	real
trunc(x)	truncamiento de *x*	real	entero

1.1.5 Expresiones

Una *expresión* es una combinación de constantes, variables, símbolos de operaciones, paréntesis y nombres de funciones especiales. Esta misma idea es utilizada en la notación matemática tradicional, es decir, se tiende a seguir las reglas generales del álgebra que a menudo se dan por conocidas. En el caso que se deseen agrupar expresiones, se deben utilizar paréntesis.

Ejemplo 1.4: Se tiene la expresión algebraica para calcular el módulo de la impedancia de un circuito R-L-C:

$$\sqrt{R^2+\left(\omega L-\frac{1}{\omega C}\right)^2} \qquad \textbf{(1.2)}$$

La expresión algorítmica de **(1.2)** es la siguiente:

sqrt(R^2+(ω*L-1/(ω*C))^2) **(1.3)**

Las expresiones que tienen dos o más operandos requieren unas reglas matemáticas que permiten determinar el orden de las operaciones. Estas reglas se denominan *reglas de prioridad o precedencia* y son las siguientes:

- Se deben evaluar las operaciones que están encerradas entre paréntesis (en caso hubiere). Si existen diferentes paréntesis anidados (interiores unos a otros), las expresiones más internas son las que se evalúan primero.
- Las operaciones aritméticas dentro de una expresión suelen seguir el siguiente orden de prioridad: operador exponencial (^), operadores * y /, operadores + y –, y finalmente los operadores *div* y *mod*.

1.2 Solución de problemas

Para que un problema sea solucionado satisfactoriamente, se deben considerar tres etapas fundamentales: *análisis* del problema, *diseño* del algoritmo y la *implementación* del mismo en la computadora.

1.2.1 Análisis del problema

Se define lo que se desea hacer, además de las necesidades y los requerimientos del algoritmo. Surgen las siguientes preguntas:

¿Qué se debe hacer?

¿Qué se necesita?

¿Como se realiza?

¿Qué se desea obtener?

1.2.2 Diseño del algoritmo

Para diseñar un algoritmo, se siguen ciertas técnicas, tales como:

- **Recursos abstractos** – Se utiliza para diseñar un algoritmo general, que permita ser implementado posteriormente en cualquier lenguaje de programación. Al utilizar los recursos abstractos no se debe tomar en cuenta el lenguaje de programación que se utilizará, ni el equipo del cual se disponga.
- **Diseño modular** – Permite la división de un problema dado en problemas más simples, cada uno de los cuales será un *módulo*, el cual debe tener en promedio de 30 a 60 instrucciones. El diseño modular es útil porque el problema se puede solucionar en grupos de trabajo.
- **Diseño Descendente** – También llamado *Top-Down Design*, consiste en la descomposición del problema original en subproblemas más simples y a

continuación dividir estos subproblemas en otros más simples que pueden ser implementados para su solución en la computadora.

1.2.3 Implementación del algoritmo mediante la computadora

En general, la implementación de un algoritmo sigue tres etapas las cuales se enumeran a continuación:

- **Codificación** – Consiste en la traducción del algoritmo al lenguaje de programación escogido. Debe tenerse en cuenta que el algoritmo diseñado debe ser lo suficientemente flexible como para adaptarse a cualquier lenguaje de programación.
- **Ejecución** – Consiste en la ejecución de las órdenes del programa codificado en una computadora, de tal manera que pueda apreciarse el funcionamiento de las sentencias traducidas del algoritmo.
- **Comprobación** – Se proporcionan datos de prueba al programa y se procede a verificar si luego de ser procesados estos datos devuelven el resultado esperado.

1.2.4 Representación de los algoritmos

Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación elegido. Esto permitirá que un algoritmo pueda ser procesado indistintamente en cualquier lenguaje. Para conseguir este objetivo, se precisa que este algoritmo sea representado gráfica o numéricamente, de modo que las acciones sucesivas no dependan de la sintaxis de ningún lenguaje de programación, sino que la descripción pueda servir fácilmente para su transformación en un programa, es decir, su *codificación.* Los métodos usuales para representar un algoritmo son:

- **Diagrama de flujo** – Es aquel en que cada acción se representa mediante una figura y para poder visualizar la secuencia de las acciones se utilizan las flechas en la dirección que se especifique.
- **Diagrama N-S** – Las acciones se representan mediante rectángulos y no se utilizan las flechas que indican las secuencias, debido a que las secuencias de instrucciones se ejecutarán de arriba hacia abajo. También es conocido como *Diagrama de Chapin* o *Diagrama Nassi-Schneidermann.*
- **Pseudocódigo** – Es un lenguaje de especificación de algoritmos en el que se utilizarán palabras con ciertas estructuras predefinidas. El uso de tal lenguaje hace que el paso de codificación final (esto es, la traducción a un lenguaje de programación) sea relativamente fácil. El pseudocódigo nació como un lenguaje

similar al inglés y era un medio de representar básicamente las estructuras de control de programación estructurada. La ventaja del pseudocódigo es que al ser utilizado su uso en la planificación de un programa, el programador puede concentrarse en la lógica y en las estructuras de control y no preocuparse de las reglas de sintaxis de un lenguaje de programación específico. Inclusive es fácil modificar el pseudocódigo si se descubren errores o anomalías en la lógica de un programa.

Ejemplo 1.5: La Figura 1.3 muestra el diagrama de flujo, el diagrama N-S y el pseudocódigo utilizado para hallar la reactancia equivalente de dos reactancias conectadas en paralelo:

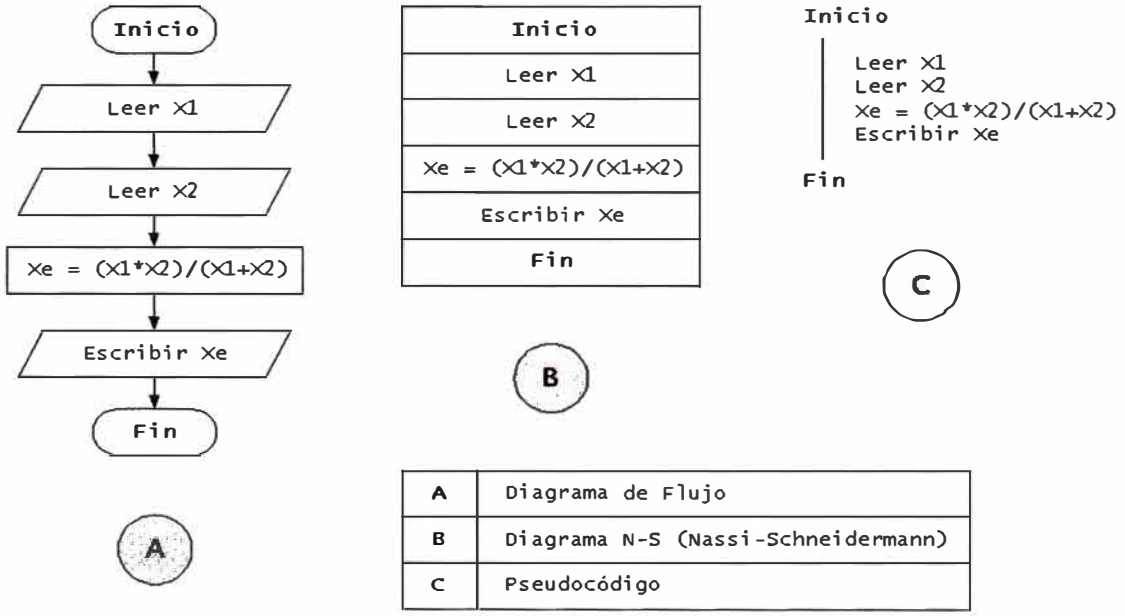


Figura 1.3: Representación del algoritmo para hallar la reactancia equivalente de dos reactancias en paralelo

1.3 El pseudocódigo

1.3.1 Definición

Se entiende por *pseudocódigo* como la forma de representar a los algoritmos, de tal manera que puede ser entendido por cualquier persona. El pseudocódigo se basa en ciertas palabras y estructuras mediante las cuales se deben de representar todas las acciones. Debe tomarse en consideración que un algoritmo realizado en pseudocódigo no funcionará directamente en la computadora. Para que éste pueda funcionar, debe ser traducido a un

lenguaje de programación. El pseudocódigo se utiliza para poder trabajar de manera estructurada con recursos abstractos.

1.3.2 Partes de un pseudocódigo

- **Definiciones** – En esta parte se colocará el nombre del pseudocódigo, así como la declaración de constantes, variables y subalgoritmos.
- **Cuerpo** – Viene a ser el algoritmo en sí y en donde se especifican todas las instrucciones necesarias para solucionar el problema planteado. El cuerpo está compuesto por tres grupos básicos de instrucciones: *solicitud de datos*, *realización de cálculos* y *reporte de resultados*.

1.3.3 Nomenclatura utilizada

Asignación

La operación de *asignación* consiste en otorgar un valor a una variable. Se representa con el símbolo u operador ←. Cuando se encuentra especificada en una sentencia de un lenguaje de programación en particular, se le denomina *instrucción o sentencia de asignación*. El formato general de una operación de asignación es el siguiente:

$$\textit{nombre variable} \leftarrow \textit{expresión} \qquad \textbf{(1.4)}$$

Entrada de información

La operación de entrada permite leer determinados valores y asignarlos a determinadas variables. Esta entrada se conoce como *operación de lectura*, la cual presenta el siguiente formato:

$$\textit{Leer variable(s)} \qquad \textbf{(1.5)}$$

Si se desea especificar más de una variable a ser leída, cada variable considerada en la lista debe estar separada por una coma.

Salida de información

La operación de salida permite mostrar los valores de las variables procesadas a lo largo del algoritmo. La salida puede aparecer en un dispositivo tal como pantalla, impresora, etc. Esta salida se conoce como *operación de escritura*, la cual presenta el siguiente formato:

Escribir cadena_caracteres / variable(s) **(1.6)**

Si se desea especificar más de una variable a ser mostrada, cada variable considerada en la lista debe estar separada por una coma. Si se desea mostrar un mensaje, la cadena de caracteres correspondiente debe estar escrita entre comillas.

Comentarios

Los comentarios sirven para orientar a la persona que está leyendo el pseudocódigo, no se constituye como una orden a ejecutar. El comentario va entre llaves "{}".

Ejemplo 1.6: Del sistema eléctrico mostrado en la Figura 1.4, elaborar un pseudocódigo que permita hallar el valor de la reactancia equivalente antes de que ocurra una falla trifásica en la línea CD.

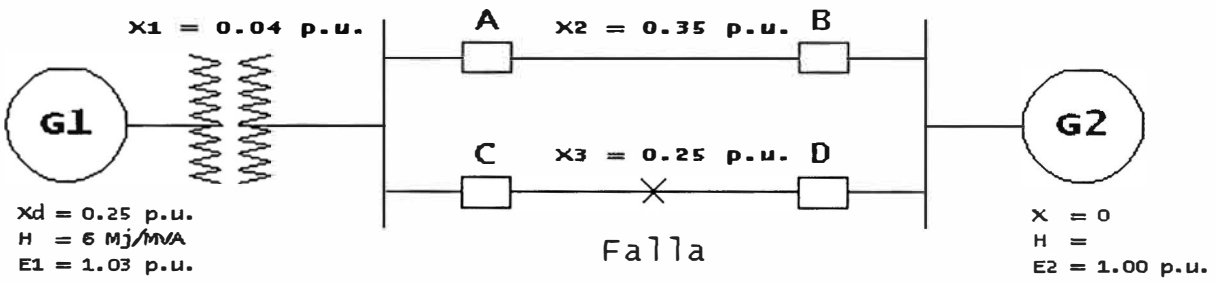


Figura 1.4: Sistema eléctrico a analizar

La expresión que permite calcular la reactancia antes de la falla es la siguiente:

$$X_{\text{antes de la falla}} = X_d + X_1 + \frac{X_2 X_3}{X_2 + X_3} \qquad \textbf{(1.7)}$$

El pseudocódigo que permite efectuar el cálculo respectivo se muestra a continuación:

```
01: Pseudocódigo  Reactancia_pre_falla
02:
03: Variables
04:
05:    Xd, X1, X2, X3 : real
06:    Xr             : real
07:
08: Inicio
09:
10:    { Lectura de los valores de las reactancias }
11:
12:    Paso N°1:  Leer  Xd, X1, X2, X3
13:
14:    { Cálculo de la reactancia pre-falla }
15:
16:    Paso N°2:  Xr ← Xd + X1 + (X2*X3)/(X2+X3)
17:
18:    { Imprimir resultado }
19:
20:    Paso N°3:  Escribir  "Xr = " , Xr
21:
22: Fin
```

Pseudocódigo 1.1: Reactancia_pre_falla

Las líneas N°1 a N°7 corresponden a las definiciones del pseudocódigo. En la línea N°1 se establece el nombre del pseudocódigo, que es de libre de elección del diseñador. En las líneas N°3 a N°6 se especifican las variables de entrada de tipo real, donde *Xd*, *X1*, *X2* y *X3* son las variables de entrada y la variable *Xr* es aquella donde se almacena el resultado. Las líneas N°8 a N°22 corresponden al cuerpo del pseudocódigo. En la línea N°12 se especifica la lectura de las variables de entrada. En la línea N°16 se efectúa el cálculo de la reactancia antes de la falla y dicho resultado se almacena en la variable *Xr*. Finalmente, en la línea N°20 se muestra el resultado previamente calculado. Nótese que en las líneas N°10, N°14 y N°18 se efectúan comentarios previos a las sentencias correspondientes a la entrada de datos, el procesamiento y la salida del resultado respectivamente.

Implementando el pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se tendrá lo siguiente:

```
01: #include <conio.h>
02: #include <stdio.h>
03:
04: /* Definicion de Variables */
05:
06:    float  Xd, X1, X2, X3 ;
07:    float  Xr ;
08:
09: void main()
10:  {
11:
12:    /* Lectura de valores de las reactancias */
13:
14:       clrscr() ;
15:       printf ("Ingreso de Reactancias: \n\n") ;
16:       printf ("- Xd = ") ;
17:       scanf ("%f", &Xd) ;
18:       printf ("- X1 = ") ;
19:       scanf ("%f", &X1) ;
20:       printf ("- X2 = ") ;
21:       scanf ("%f", &X2) ;
22:       printf ("- X3 = ") ;
23:       scanf ("%f", &X3) ;
24:
25:    /* Calculo de la reactancia pre-falla */
26:
27:       Xr = Xd + X1 + (X2*X3)/(X2+X3) ;
28:
29:   /* Imprimir resultado */
30:
31:      printf ("\nResultado:\n\n*** Xr = %7.4f p.u. ***\n",Xr) ;
32:      getch() ;
33: }
```

Programa 1.1: REAC_PF.CPP

En caso se desee efectuar una ejecución manual del pseudocódigo *Reactancia_pre_falla* mostrado líneas arriba (como paso previo a la codificación), se tendrá lo siguiente:

```
Paso N°1:    Xd = 0.25
             X1 = 0.04
             X2 = 0.35
             X3 = 0.25

Paso N°2:    Xr = 0.25 + 0.04 + (0.35*0.25)/(0.35+0.25)
             Xr = 0.25 + 0.04 + 0.0875/0.6
             Xr = 0.25 + 0.04 + 0.1458
             Xr = 0.4358

Paso N°3:    "Xr = 0.4358"
```

1.4 Estructuras de control

Las estructuras de control proporcionan a los algoritmos (y por ende a los programas) la capacidad de actuar de distinta manera bajo variadas situaciones. Las estructuras de control se dividen en 3 tipos básicos: *secuenciales*, *alternativas* y *repetitivas*.

1.4.1 Estructuras de control secuenciales

Una *estructura de control secuencial* es aquella en la cual las instrucciones se ejecutarán desde el inicio hasta el final, tal y como se muestra en la Figura 1.5:

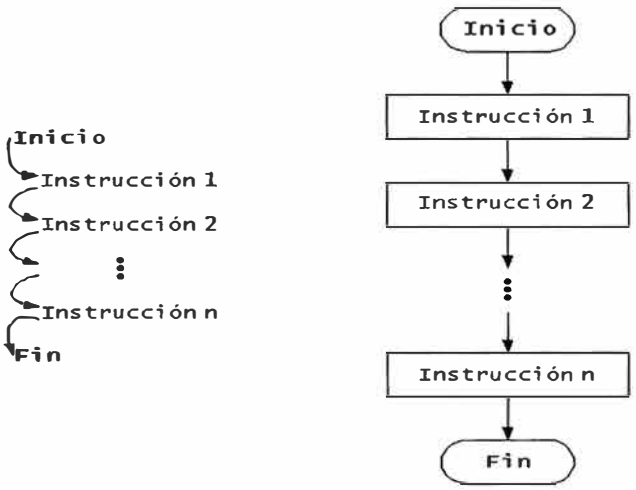


Figura 1.5: Pseudocódigo y diagrama de flujo de la estructura de control secuencial

1.4.2 Estructuras de control alternativas

Una *estructura de control alternativa* es aquella que permitirá elegir si se ejecutará o no un determinado grupo de instrucciones. Se considerarán las estructuras de control de alternativa *simple*, *doble* y *múltiple*.

Estructura de control de alternativa simple

Una *estructura de control de alternativa simple* es utilizada para indicar si se va a llevar a cabo una acción o no. Esta acción está supeditada a una determinada condición, tal y como se muestra en la Figura 1.6

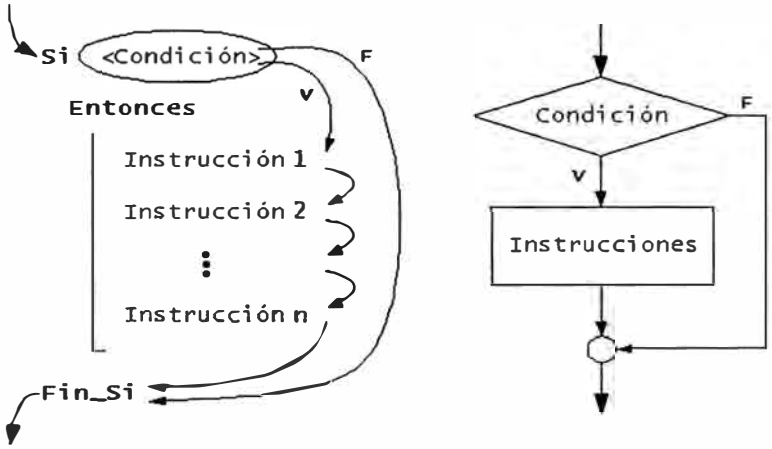


Figura 1.6: Pseudocódigo y diagrama de flujo de la estructura de control de alternativa simple

La estructura **Si-Entonces** evalúa la condición y si esta tiene un valor lógico *verdadero*, se ejecuta la secuencia de instrucciones *Instrucción 1, ..., Instrucción n* (o acción en caso de que solamente tenga que ejecutarse solamente una instrucción). Si la condición es falsa, se finaliza la ejecución de la estructura.

Estructura de control de alternativa doble

Una *estructura de control de alternativa doble* se utiliza para decidir entre dos grupos de instrucciones dada una condición, tal y como se muestra en la Figura 1.7:

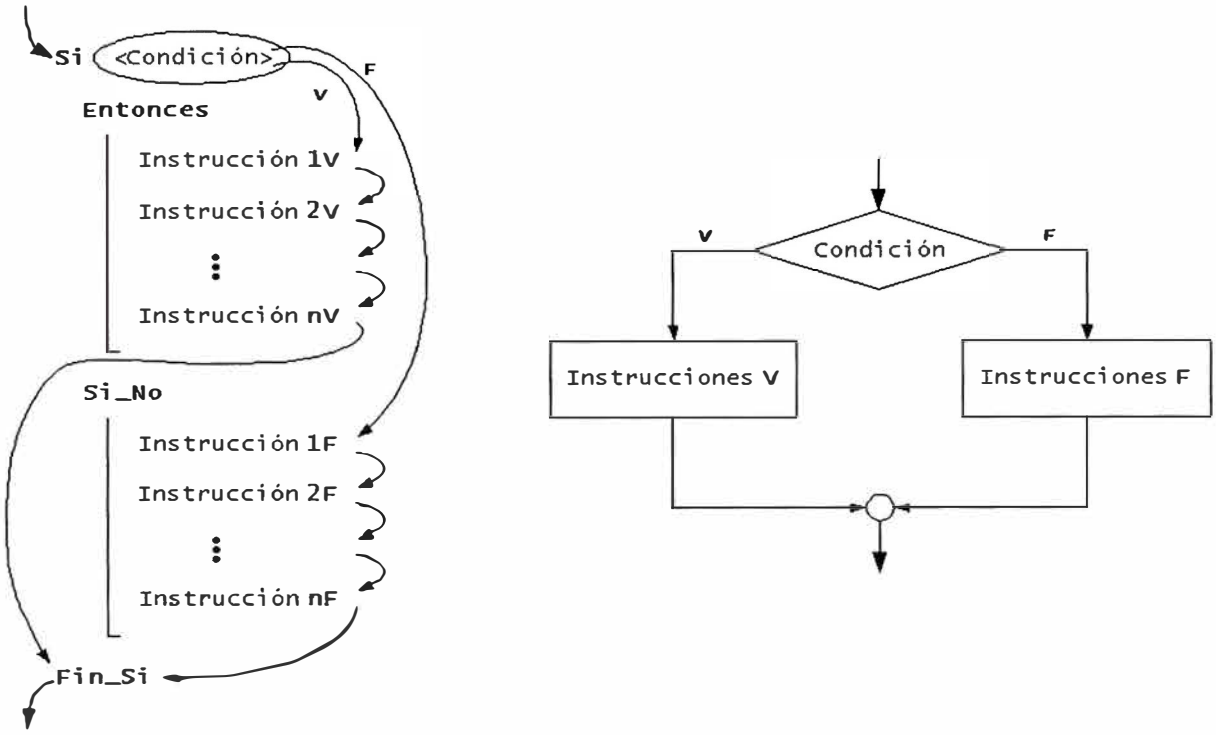


Figura 1.7: Pseudocódigo y diagrama de flujo de la estructura de control de alternativa doble

Esta estructura evalúa la condición dada y en caso tenga un valor lógico *verdadero*, se ejecuta la secuencia de instrucciones *Instrucción 1V, ..., Instrucción nV*. Si la condición tiene un valor lógico *falso*, se ejecuta la secuencia de instrucciones *Instrucción 1F, ..., Instrucción nF*.

Estructura de control alternativa múltiple

Una *estructura de control de alternativa múltiple* evaluará una expresión que podrá tomar *m* valores distintos (1, 2, 3, 4, ..., m). Según se elija uno de estos valores en la condición, se realizará una de las *m* acciones, o lo que es igual, el flujo del algoritmo seguirá un determinado camino entre los *m* posibles, tal y como se muestra en la Figura 1.8:

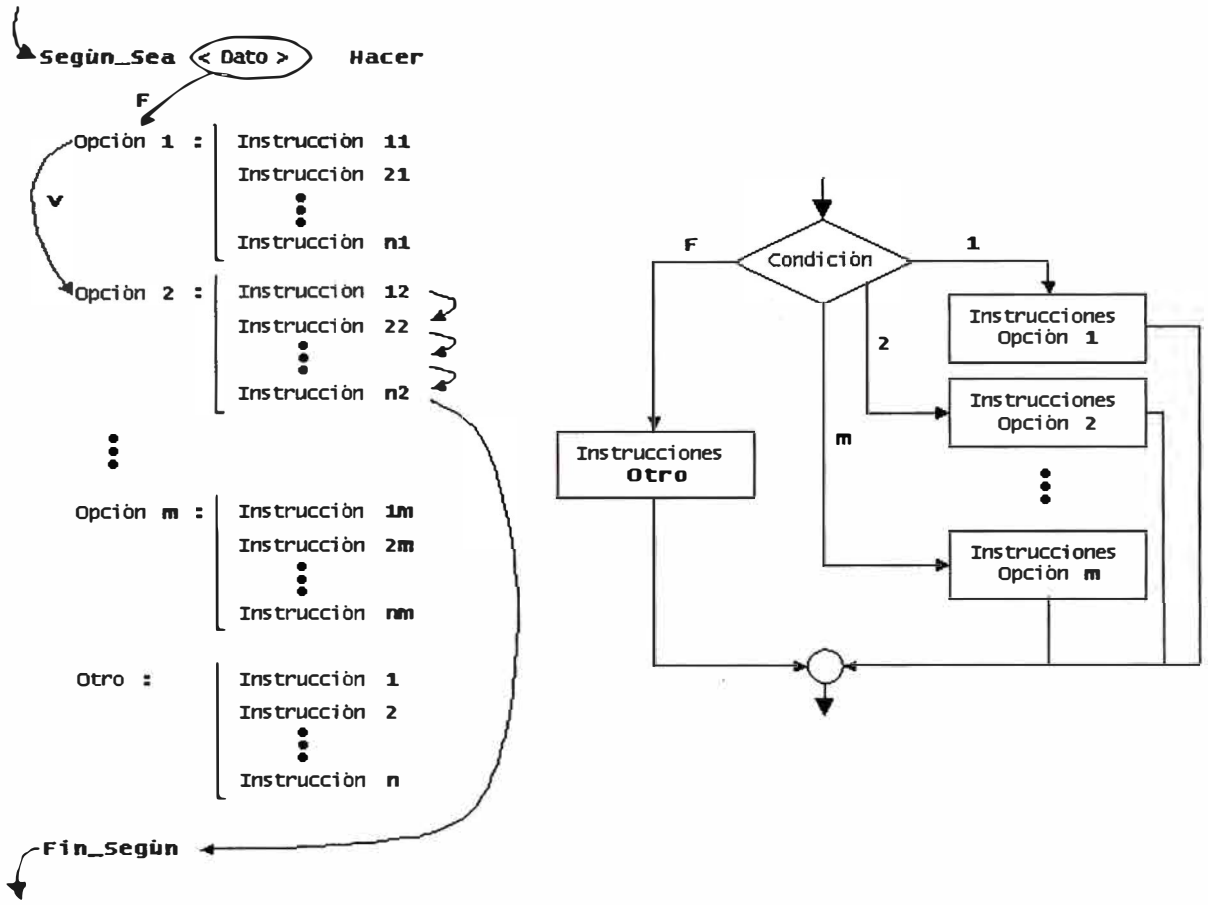


Figura 1.8: Pseudocódigo y diagrama de flujo de la estructura de control de alternativa múltiple

1.4.3 Estructuras de control repetitivas

Una *estrutura de control repetitiva* es utilizada para ejecutar una o más instrucciones varias veces. Para poder trabajar con las estructuras repetitivas se deben conocer los siguientes términos:

- **Lazo o bucle** – Se refiere al conjunto de instrucciones que se está repitiendo, es decir, una vez llegada a la última instrucción del lazo, se regresará de forma automática al inicio de la secuencia de instrucciones.
- **Contador** – Es una variable que se va incrementando de uno en uno y que permite conocer el número de lazos que están siendo ejecutados.
- **Acumulador** – Es una variable que va incrementándose una determinada cantidad por cada lazo ejecutado.

Estructura de control repetitiva *desde – hasta*

Se utiliza cuando el número de bucles a ejecutar es conocido. Se dispone de un valor inicial a partir del cual se empieza a incrementar el contador, así como un valor final al cual tiene que llegar el contador para que finalice la ejecución del conjunto de instrucciones al interior de esta estructura. Esto se encuentra representado en la Figura 1.9, la cual se muestra a continuación:

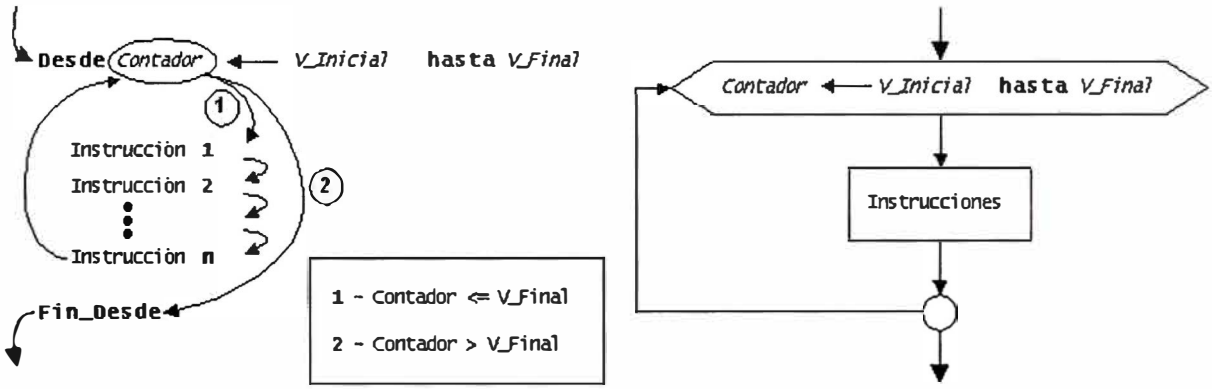


Figura 1.9: Pseudocódigo y diagrama de flujo de la estructura de control repetitiva *desde-hasta*

Ejemplo 1.7: Del sistema eléctrico mostrado en la Figura 1.10, G1 es una central que alimenta un sistema eléctrico representado por una barra infinita G2. Elaborar un psudocódigo que permita calcular las reactancias, constantes de potencia eléctrica y las curvas de potencia transmitida en un intervalo de 0 a π con un tamaño de paso de $\pi/20$ antes, durante y después de la falla trifásica que toma lugar en el punto medio de la línea de transmisión CD.

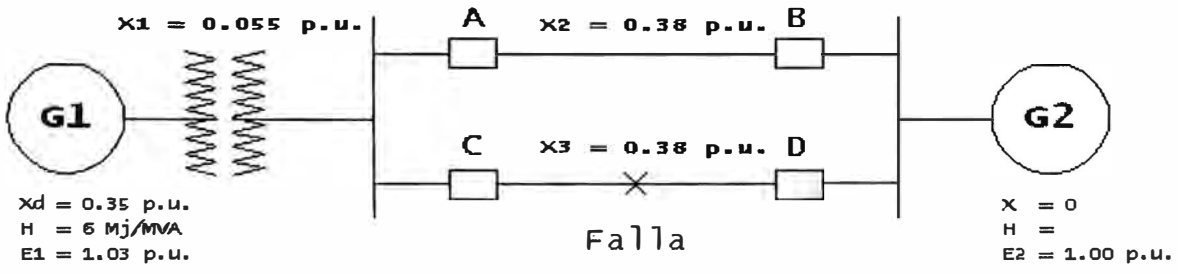


Figura 1.10: Sistema eléctrico a analizar

Para efectuar el cálculo de la reactancia antes de la falla, consideramos el esquema mostrado en la Figura 1.11:

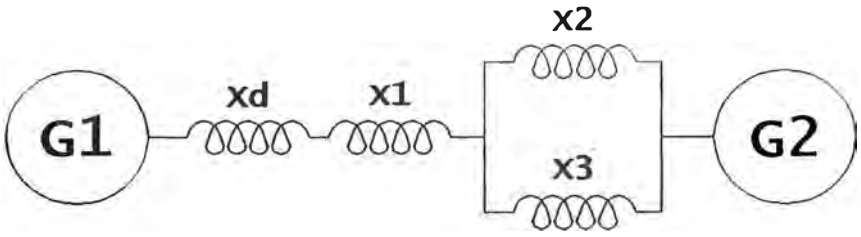


Figura 1.11: Esquema del sistema eléctrico antes de la falla

La expresión para calcular la reactancia correspondiente se muestra a continuación:

$$X_{\text{antes de la falla}} = X_{r1} = X_d + X_1 + \frac{X_2 X_3}{X_2 + X_3} \qquad \textbf{(1.8)}$$

Para efectuar el cálculo de la reactancia durante la falla, consideramos en esquema mostrado en la Figura 1.12:

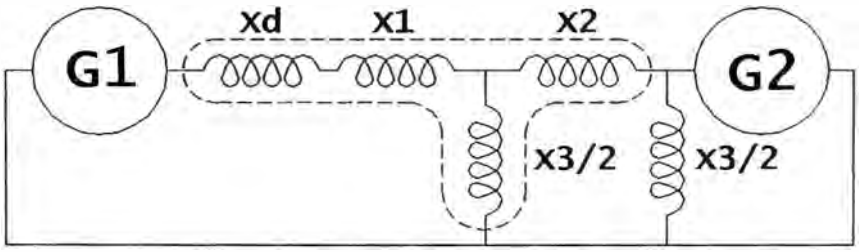


Figura 1.12: Esquema del sistema eléctrico durante la falla

Efectuando una transformación estrella-delta, la expresión para calcular la reactancia correspondiente se muestra a continuación:

$$X_{\text{durante la falla}} = X_{r2} = X_d + X_1 + X_2 + \frac{2X_2(X_d + X_1)}{X_3} \qquad \textbf{(1.9)}$$

Para efectuar el cálculo de la reactancia después de la falla, consideramos en esquema mostrado en la Figura 1.13:

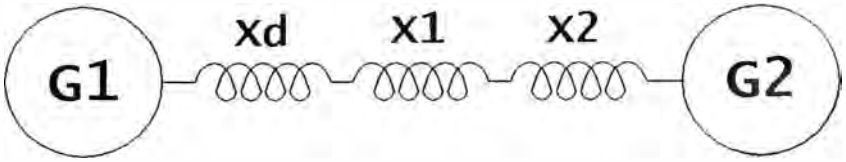


Figura 1.13: Esquema del sistema eléctrico después de la falla

La expresión para calcular la reactancia correspondiente se muestra a continuación:

$$X_{\text{después de la falla}} = X_{r3} = X_d + X_1 + X_2 \qquad \textbf{(1.10)}$$

Las constantes para la potencia eléctrica transmitida antes, durante y después de la falla son las siguientes:

$$C_1 = \frac{E_1 E_2}{X_{r1}} \;;\; C_2 = \frac{E_1 E_2}{X_{r2}} \;;\; C_3 = \frac{E_1 E_2}{X_{r3}} \qquad \textbf{(1.11)}$$

Las ecuaciones de potencia eléctrica transmitida antes, durante y después de la falla serán:

$$\begin{aligned} P_{e1} &= C_1 sen(\delta) \\ P_{e2} &= C_2 sen(\delta) \\ P_{e3} &= C_3 sen(\delta) \end{aligned} \qquad \textbf{(1.12)}$$

El pseudocódigo que permite efectuar el cálculo respectivo se muestra a continuación:

```
01: Pseudocódigo  Calculo_falla_trifasica
02:
03: Constantes
04:
05:   PI ← 3.141592654
06:   MAX ← 20
07:
```

```
08: Variables
09:
10:   { Tensiones de generador y barra infinita }
11:
12:     E1, E2 : real
13:
14:   { Reactancias del sistema eléctrico }
15:
16:     Xd, X1, X2, X3 : real
17:
18:   { Reactancias antes, durante y después de la falla }
19:
20:     Xr1, Xr2, Xr3 : real
21:
22:   { Constantes de potencia eléctrica transmitida }
23:
24:     C1, C2, C3 : real
25:
26:   { Variables de potencia eléctrica transmitida }
27:
28:     Pe1, Pe2, Pe3 : real
29:
30:   { Ángulo de potencia eléctrica transmitida }
31:
32:     delta : real
33:
34:   { Contador }
35:
36:     i : entero
37:
38: Inicio
39:
40:   { Lectura de datos del sistema eléctrico }
41:
42:     Paso N°01:  Escribir "E1 = "
43:                 Leer E1
44:                 Escribir "E2 = "
45:                 Leer E2
46:                 Escribir "Xd = "
47:                 Leer Xd
48:                 Escribir "X1 = "
49:                 Leer X1
50:                 Escribir "X2 = "
51:                 Leer X2
```

```
52:                Escribir "X3 = "
53:                Leer X3
54:
55:   { Cálculo de reactancias antes, durante y después de la falla }
56:
57:     Paso N°02:  Xr1 ← Xd + X1 + (X2*X3)/(X2+X3)
58:                 Xr2 ← Xd + X1 + X2 + 2*X2*(Xd+X1)/X3
59:                Xr3 ← Xd + X1 + X2
60:
61:   { Cálculo de las constantes de potencia eléctrica transmitida }
62:
63:     Paso N°03:  C1 ← E1*E2/Xr1
64:                C2 ← E1*E2/Xr2
65:                C3 ← E1*E2/Xr3
66:
67:   { Mostrar resultados }
68:
69:   { Reactancias antes, durante y después de la falla }
70:
71:     Paso N°04:  Escribir "Xr1 = " , Xr1
72:                Escribir "Xr2 = " , Xr2
73:                Escribir "Xr3 = " , Xr3
74:                Escribir "C1 = " , C1
75:                Escribir "C2 = " , C2
76:                Escribir "C3 = " , C3
77:
78:   { Curvas de potencia eléctrica transmitida }
79:
80:     Paso N°05:  Escribir "Curvas de Potencia"
81:
82:     Paso N°06:  delta ← 0
83:
84:     Paso N°07:  Desde  i ← 1  hasta (MAX+1) hacer
85:
86:                   Paso N°08:  Pe1 ← C1*sen(delta)
87:                               Pe2 ← C2*sen(delta)
88:                               Pe3 ← C3*sen(delta)
89:
90:                   Paso N°09:  Escribir  i, delta, Pe1, Pe2, Pe3
91:
92:                   Paso N°10:  delta ← delta + PI/MAX
93:
94:                Fin_Desde
```

```
95:
96: Fin
```

Pseudocódigo 1.2: Calculo_falla_trifasica

En la línea N°1 se establece el nombre del pseudocódigo. Entre las líneas N°3 y N°6 se definen dos constantes: el número π y el número de subintervalos requeridos por el enunciado. Entre las líneas N°8 y N°36 se definen las variables a ser utilizadas por el pseudocódigo y que se encuentran convenientemente comentadas. La entrada de los datos provenientes del sistema eléctrico se encuentra especificadas entre las líneas N°40 a la N°53. El cálculo de las reactancias se lleva a cabo de acuerdo a las expresiones **(1.8)**, **(1.9)** y **(1.10)**, mostrándose entre las líneas N°55 y N°59. El cálculo de las constantes de potencia eléctrica transmitida se lleva a cabo de acuerdo a la expresión **(1.11)** y se muestra entre las líneas N°61 y N°65. Finalmente se efectúa un reporte de las reactancias y constantes obtenidas, así como el reporte de los valores de ángulo y potencia eléctrica a intervalos de tamaño $\pi/20$, tal y como se muestra entre las líneas N°67 y N°94.

Implementando el pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se tendrá lo siguiente:

```
001: #include <conio.h>
002: #include <stdio.h>
003: #include <math.h>
004:
005: /* Definicion de constantes */
006:
007:    const  int MAX = 20 ;
008:
009: /* Definicion de variables */
010:
011:    /* Tensiones del generador y de barra infinita      */
012:
013:    float  E1, E2 ;
014:
015:    /* Reactancias del sistema electrico              */
016:
017:    float  Xd, X1, X2, X3 ;
```

```
018:
019:    /* Reactancias antes, durante y despues de la falla */
020:
021:    float  Xr1, Xr2, Xr3 ;
022:
023:    /* Constantes de potencia electrica transmitida     */
024:
025:    float  C1, C2, C3 ;
026:
027:    /* Variables de potencia electrica transmitida      */
028:
029:    float  Pe1, Pe2, Pe3 ;
030:
031:    /* Angulos de potencia electrica transmitida        */
032:
033:    float  delta ;
034:
035:    /* Contador                                         */
036:
037:    int    i ;
038:
039: void main()
040: {
041:
042:    /* Lectura de datos del sistema electrico */
043:
044:    clrscr() ;
045:    printf ("*** Datos del Sistema Electrico ***\n\n") ;
046:    printf ("Tensiones del Sistema Electrico ...\n\n") ;
047:    printf ("E1 (p.u.) = ") ; scanf("%f", &E1) ;
048:    printf ("E2 (p.u.) = ") ; scanf("%f", &E2) ;
049:    printf ("\nReactancias del Sistema Electrico ...\n\n") ;
050:    printf ("Xd (p.u.) = ") ; scanf("%f", &Xd) ;
051:    printf ("X1 (p.u.) = ") ; scanf("%f", &X1) ;
052:    printf ("X2 (p.u.) = ") ; scanf("%f", &X2) ;
053:    printf ("X3 (p.u.) = ") ; scanf("%f", &X3) ;
054:    printf ("\nPresione cualquier tecla para continuar ...\n\n") ;
055:    getch() ;
056:
057:    /* Calcular reactancias antes, durante y despues de la falla */
058:
059:    Xr1 = Xd + X1 + (X2*X3)/(X2+X3) ;
060:    Xr2 = Xd + X1 + X2 + 2*X2*(Xd+X1)/X3 ;
061:    Xr3 = Xd + X1 + X2 ;
```

```
062:
063:    /* Calcular las constantes de potencia electrica transmitida */
064:
065:    C1 = E1*E2/Xr1 ;
066:    C2 = E1*E2/Xr2 ;
067:    C3 = E1*E2/Xr3 ;
068:
069:    /* Mostrar resultados */
070:
071:    clrscr() ;
072:    printf ("*** Resultados ***\n\n") ;
073:
074:    /* Reactancias antes, durante y despues de la falla */
075:
076:    printf ("Reactancias ... \n\n") ;
077:    printf ("- Antes de la falla   :  Xr1 = %10.4f\n",Xr1) ;
078:    printf ("- Durante la falla    :  Xr2 = %10.4f\n",Xr2) ;
079:    printf ("- Despues de la falla :  Xr3 = %10.4f\n\n",Xr3) ;
080:
081:    /* Constantes de potencia electrica transmitida */
082:
083:    printf ("Constantes de potencia electrica transmitida ...\n\n") ;
084:    printf ("- Antes de la falla   :  C1 = %10.4f\n",C1) ;
085:    printf ("- Durante la falla    :  C2 = %10.4f\n",C2) ;
086:    printf ("- Despues de la falla :  C3 = %10.4f\n\n",C3) ;
087:
088:    /* Curvas de potencia transmitida */
089:
090:    printf ("Curvas de potencia transmitida ...\n\n") ;
091:
092:    printf ("-----------------------------------------\n") ;
093:    printf ("| i  | Angulo |   P1   |   P2   |   P3   |\n") ;
094:    printf ("-----------------------------------------\n") ;
095:
096:    delta = 0 ;
097:
098:    for (i=1;i<=MAX+1;i++)
099:     {
100:
101:       Pe1 = C1*sin(delta) ;
102:       Pe2 = C2*sin(delta) ;
103:       Pe3 = C3*sin(delta) ;
104:       printf ("| %2i | %6.2f | %6.3f | %6.3f | %6.3f |\n",i, …
                  delta*180/M_PI,Pe1,Pe2,Pe3) ;
```

```
105:       delta = delta + M_PI/MAX ;
106:
107:     }
108:
109:    printf ("-----------------------------------------\n\n") ;
110:    printf ("Presione cualquier tecla para salir ...") ;
111:    getch() ;
112:
113: }
```

Programa 1.2: CALFT.CPP

Nótese que en el programa no se define la constante π, debido a que Borland C++ 3.1 tiene una constante similar que se encuentra definida en sus librerías matemáticas, bajo el nombre *M_PI*. Adicionalmente, cabe resaltar que se han efectuado refinamientos al código de tal manera que la entrada y la salida de datos sea lo más amigable posible, dado que el programa muestra los detalles principales de la implementación del algoritmo. La entrada y la salida del programa se muestran en las Figuras 1.14 (entrada de los datos del sistema eléctrico) y 1.15 (reporte de resultados) respectivamente.

Figura 1.14: Ingreso de datos del sistema eléctrico

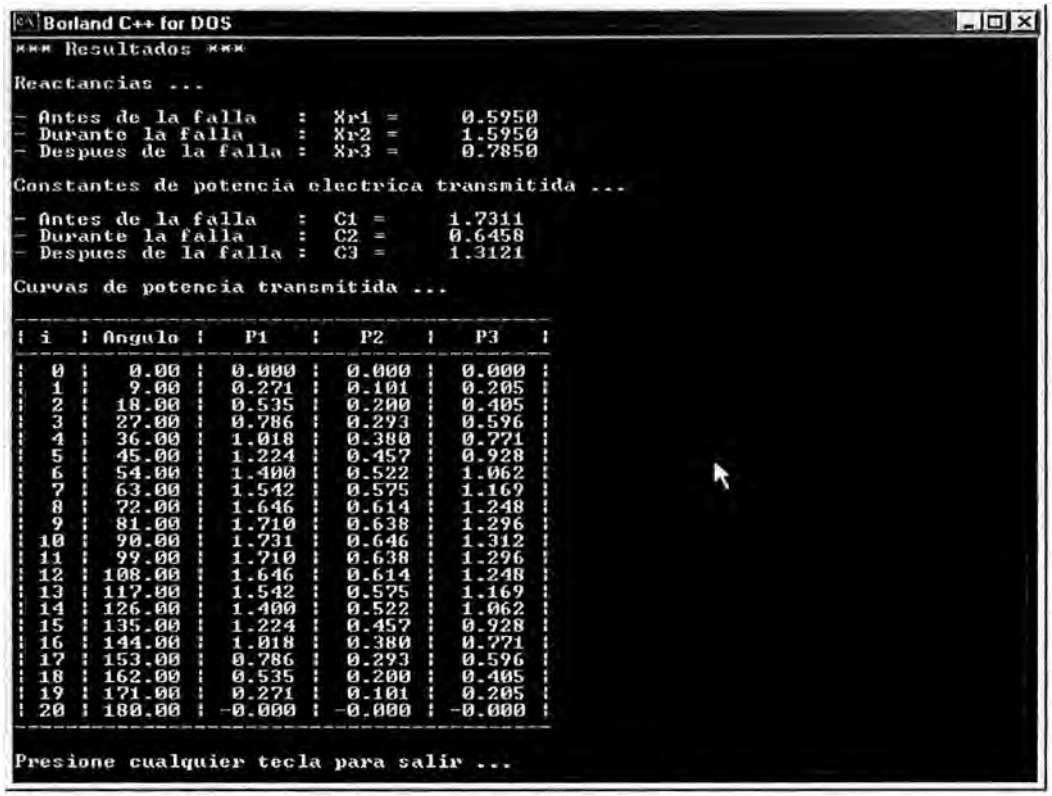


Figura 1.15: Reporte de resultados

Estructura de control repetitiva *mientras*

Esta estructura se va a utilizar cuando se desconoce el número de veces que se va a ejecutar un grupo de instrucciones. En este caso, la condición se analiza al inicio de la estructura y las instrucciones se ejecutarán mientras el valor lógico de la condición sea verdadera. Esta estructura se puede representar gráficamente tal y como se muestra en la Figura 1.16:

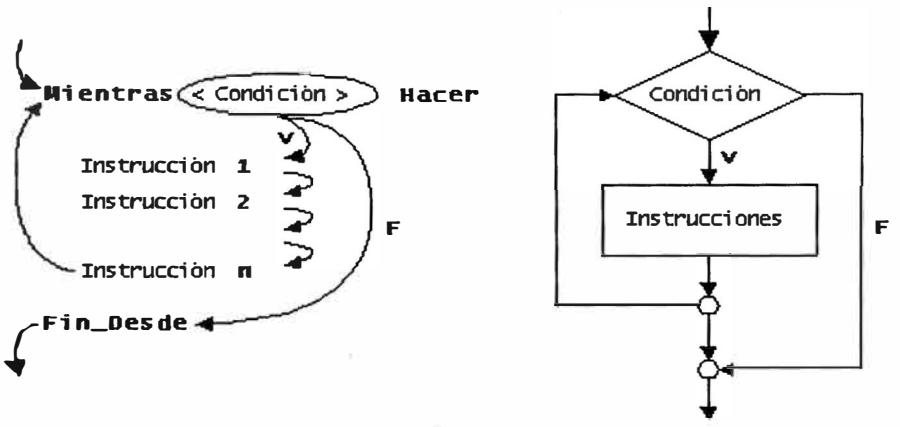


Figura 1.16: Pseudocódigo y diagrama de flujo de la estructura de control repetitiva *mientras*

Ejemplo 1.8: Modificar el segmento de pseudocódigo (así como el respectivo programa en C) del Ejemplo 1.7 que corresponde a las curvas de potencia transmitida, utilizando la estructura de control repetitiva *mientras*.

El segmento de pseudocódigo a considerar de acuerdo a las condiciones establecidas en el enunciado, es el siguiente:

```
 :

078:   { Curvas de potencia eléctrica transmitida }
079:
080:     Paso N°05:  Escribir "Curvas de Potencia"
081:
082:     Paso N°06:  delta ← 0
083:
084:     Paso N°07:  i ← 1
085:
086:     Paso N°08:  Mientras (i <= (MAX+1)) hacer
087:
088:                   Paso N°09:  Pe1 ← C1*sen(delta)
089:                               Pe2 ← C2*sen(delta)
090:                               Pe3 ← C3*sen(delta)
091:
092:                   Paso N°10:  Escribir  i, delta, Pe1, Pe2, Pe3
093:
094:                   Paso N°11:  delta ← delta + PI/MAX
095:
096:                   Paso N°12:  i ← i + 1
097:
098:                 Fin_Mientras
099:
100: Fin
```

La implementación del segmento de pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se muestra a continuación:

```
088:    /* Curvas de potencia transmitida */
089:
090:    printf ("Curvas de potencia transmitida ...\n\n") ;
091:
092:    printf ("-----------------------------------------\n") ;
093:    printf ("| i  | Angulo |   P1   |   P2   |   P3   |\n") ;
094:    printf ("-----------------------------------------\n") ;
095:
096:    delta = 0 ;
097:    i = 0 ;
098:
099:    while (i<=MAX)
100:     {
101:
102:       Pe1 = C1*sin(delta) ;
103:       Pe2 = C2*sin(delta) ;
104:       Pe3 = C3*sin(delta) ;
105:       printf ("| %2i | %6.2f | %6.3f | %6.3f | %6.3f |\n",i, …
                    delta*180/M_PI,Pe1,Pe2,Pe3) ;
106:       delta = delta + M_PI/MAX ;
107:       i = I + 1 ;
108:
109:     }
110:
111:    printf ("-----------------------------------------\n\n") ;
112:    printf ("Presione cualquier tecla para salir ...") ;
113:    getch() ;
114:
115: }
```

Estructura de control repetitiva *repetir – hasta*

Es semejante a la estructura anterior, con la diferencia de que la condición se analizará al final de la estructura, luego de haberse ejecutado todas las instrucciones. En este caso, si la condición es verdadera se continuará con la siguiente instrucción del algoritmo, de lo contrario se regresará al inicio de la estructura. Esta estructura se puede representar gráficamente tal y como se muestra en la Figura 1.17:

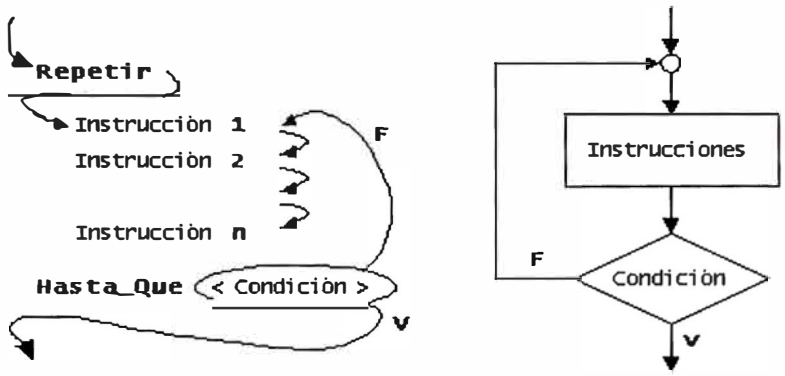


Figura 1.17: Pseudocódigo y diagrama de flujo de la estructura de control repetitiva *repetir-hasta*

Ejemplo 1.9: Modificar el segmento de pseudocódigo (así como el respectivo programa en C) del Ejemplo 1.7 que corresponde a las curvas de potencia transmitida, utilizando la estructura de control repetitiva *repetir-hasta.*

El segmento de pseudocódigo a considerar de acuerdo a las condiciones establecidas en el enunciado, es el siguiente:

```
:

078:    { Curvas de potencia eléctrica transmitida }
079:
080:      Paso N°05:  Escribir "Curvas de Potencia"
081:
082:      Paso N°06:  delta ← 0
083:
084:      Paso N°07:  i ← 1
085:
086:      Paso N°08:  Repetir
087:
088:                    Paso N°09:  Pe1 ← C1*sen(delta)
089:                                Pe2 ← C2*sen(delta)
090:                                Pe3 ← C3*sen(delta)
091:
092:                    Paso N°10:  Escribir  i, delta, Pe1, Pe2, Pe3
093:
094:                    Paso N°11:  delta ← delta + PI/MAX
```

```
095:
096:                    Paso N°12:  i ← i + 1
097:
098:                  Hasta_Que (i>(MAX+1))
099:
100: Fin
```

La implementación del segmento de pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se muestra a continuación:

```
:

088:   /* Curvas de potencia transmitida */
089:
090:   printf ("Curvas de potencia transmitida ...\n\n") ;
091:
092:   printf ("-----------------------------------------\n") ;
093:   printf ("| i  | Angulo |   P1   |   P2   |   P3   |\n") ;
094:   printf ("-----------------------------------------\n") ;
095:
096:   delta = 0 ;
097:   i = 0 ;
098:
099:   do
100:    {
101:
102:      Pe1 = C1*sin(delta) ;
103:      Pe2 = C2*sin(delta) ;
104:      Pe3 = C3*sin(delta) ;
105:      printf ("| %2i | %6.2f | %6.3f | %6.3f | %6.3f |\n",i, …
                  delta*180/M_PI,Pe1,Pe2,Pe3) ;
106:      delta = delta + M_PI/MAX ;
107:      i = i + 1 ;
108:
109:    }
110:   while  (i<=MAX)
111:
112:   printf ("-----------------------------------------\n\n") ;
113:   printf ("Presione cualquier tecla para salir ...") ;
114:   getch() ;
115:
116: }
```

1.5 Subalgoritmos

Un *subalgoritmo* se constituye como un conjunto de instrucciones que permite efectuar una determinada labor y que va a ser invocado y ejecutado desde un algoritmo principal. Es requisito indispensable que este conjunto de instrucciones se encuentre agrupado y tenga un nombre. Los subalgoritmos pueden ser *Funciones* o *Procedimientos*.

1.5.1 Funciones

Una *función* es un subalgoritmo que con ciertos datos proporcionados va a efectuar un cálculo o trabajo y devolverá un valor o resultado. Cada lenguaje de programación tiene sus propias funciones incorporadas, denominadas *funciones internas* o *intrínsecas* y las funciones definidas por el usuario, *funciones externas* (las cuales son definidas mediante una *declaración de función*). La declaración de una función será la siguiente:

```
<tipo de resultado> función <nombre_función> (argumento(s))
[declaraciones locales]
inicio
 ≡
 <acciones>                                                    (1.13)
 ≡
 devolver (<expresión>)
fin_función
```

Los argumentos son los datos proporcionados a la función, los cuales están separados mediante una coma. Se pueden tener variables locales declaradas dentro de la especificación de la función. En el cuerpo de la función se puede tener un conjunto de acciones, las cuales al final permiten a la función devolver un resultado.

La llamada a una función se efectúa de la siguiente manera:

```
variable ← Nombre_Función (argumento(s) actual(es))          (1.14)
```

Los argumentos actuales son las variables consideradas en el desarrollo de un algoritmo principal, cuyos valores son asignados a los argumentos formales de la

definición de la función. Finalmente, se devuelve el valor de la función al nombre de la función y se retorna al punto de llamada.

1.5.2 Procedimientos

Un *procedimiento* es un subalgoritmo que va a ser utilizado para ejecutar un conjunto de instrucciones que no necesariamente van a devolver un valor. Los procedimientos se utilizan cuando los algoritmos son demasiado grandes o cuando existe un conjunto de instrucciones que se repite en un mismo algoritmo en diferentes lugares. La forma de declarar un procedimiento es la siguiente:

procedimiento <Nombre_Procedimiento> (argumento(s) formal(es))

≡

<acciones> **(1.15)**

≡

fin_procedimiento

En este caso, dado que no se van a devolver valores, no se proporciona una instrucción especial dentro del cuerpo del procedimiento. Para indicar que se va a ejecutar un procedimiento desde el algoritmo principal, se utilizará la siguiente notación:

Llamar_A <Nombre_ Procedimiento> **(1.16)**

1.5.3 Variables globales y variables locales

Existen dos tipos de variables que según su condición se podrán utilizar en un algoritmo o subalgoritmo. Estos tipos de variables son los siguientes:

- **Variables globales** – Son aquellas que se definen al inicio del algoritmo y que van a poder ser utilizadas en cualquier parte de él o en su subalgoritmo, teniendo vigencia durante toda la ejecución del algoritmo.
- **Variables locales** – Son aquellas que se definen en un determinado subalgoritmo y que van a poder ser utilizadas en él o en algún subalgoritmo que sea invocado en él. Si una variable local es llamada desde otro lugar diferente, el subalgoritmo en donde se definió causará un error debido a que no existirá.

El ámbito de las variables globales y locales puede ser ilustrado en la Figura 1.18, la cual se muestra a continuación:

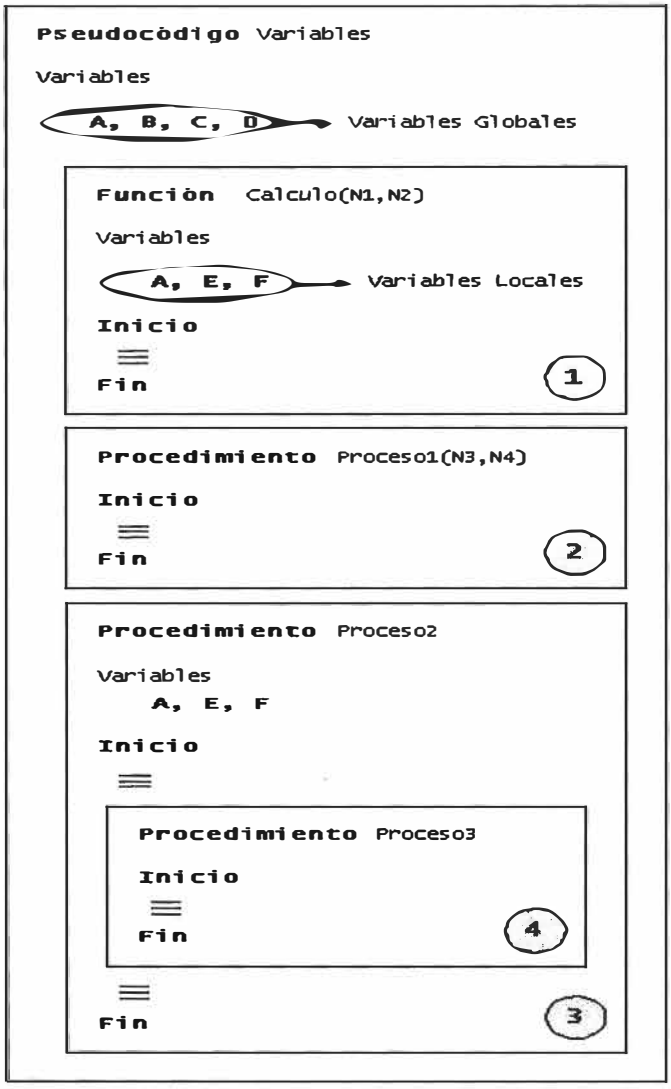


Figura 1.18: Ámbito de las variables globales y locales

En este esquema podemos apreciar que las variables globales son *A, B, C,* y *D*, las cuales pueden ser utilizadas en todo el desarrollo del pseudocódigo.

En la *Zona 1* se puede apreciar que se tienen como variables locales aquellas declaradas en la función *Calculo* (denominadas *A*, *E* y *F*) y las variables que pasan a esta función como parámetros (denominadas *N1* y *N2*). Las variables globales que pueden ser utilizadas en esta función serán *B*, *C* y *D*, dado que existe una variable *A* declarada en el interior de la función.

En la *Zona 2* se puede apreciar que las variables locales en el procedimiento *Proceso1* son las variables pasadas a este procedimiento como parámetros (denominadas *N3*, *N4*). Las variables globales que pueden ser utilizadas en este procedimiento serán *A*, *B*, *C* y *D*.

En la *Zona 3* se puede apreciar que se tienen como variables locales aquellas declaradas en el procedimiento *Proceso2* (denominadas *A*, *E* y *F*). Las variables globales que pueden ser utilizadas en este procedimiento serán *B*, *C* y *D*.

En la *Zona 4* se puede apreciar que no se tienen declaradas variables locales en el procedimiento *Proceso3*, sin embargo, tiene como variables globales a todas las variables válidas en el procedimiento *Proceso2*.

Ejemplo 1.10: Modificar el pseudocódigo (así como el respectivo programa en C) del Ejemplo 1.7 de tal manera que se tengan rutinas que permitan el ingreso de datos, el cálculo de las reactancias antes, durante y después de la falla, así como el reporte de los resultados obtenidos.

Los pseudocódigos que se requerirán para cumplir con las condiciones del enunciado anterior, se muestran en la Tabla 1.3:

Algoritmo	Descripción
Calculo_falla_trifasica_pf	Es el pseudocódigo principal que permitirá efectuar llamadas a las subrutinas definidas.
Ingresa_datos	Subrutina que permite el ingreso de los datos utilizados durante el desarrollo del pseudocódigo principal.
R_antes_f	Subrutina que permite el cálculo de la reactancia equivalente antes de la falla del sistema eléctrico considerado. Devuelve un valor numérico de tipo real.
R_durante_f	Subrutina que permite el cálculo de la reactancia equivalente durante la falla del sistema eléctrico considerado. Devuelve un valor numérico de tipo real.
R_despues_f	Subrutina que permite el cálculo de la reactancia equivalente después de la falla del sistema eléctrico considerado. Devuelve un valor numérico de tipo real.
Muestra_datos	Subrutina que permite efectuar un reporte de los resultados obtenidos.

Tabla 1.3: Descripción de pseudocódigos

El pseudocódigo principal ***Calculo_falla_trifasica_pf*** es el encargado de efectuar llamadas a las subrutinas consideradas en la Tabla 1.3. Así mismo, las subrutinas proporcionarán un retorno una vez finalizada la ejecución de la subrutina invocada. Este retorno se da en términos de ejecución de tareas, asignación de valores, cambios de variables, etc. Estas llamadas y retornos se representan gráficamente, tal y como se muestra en la Figura 1.19:

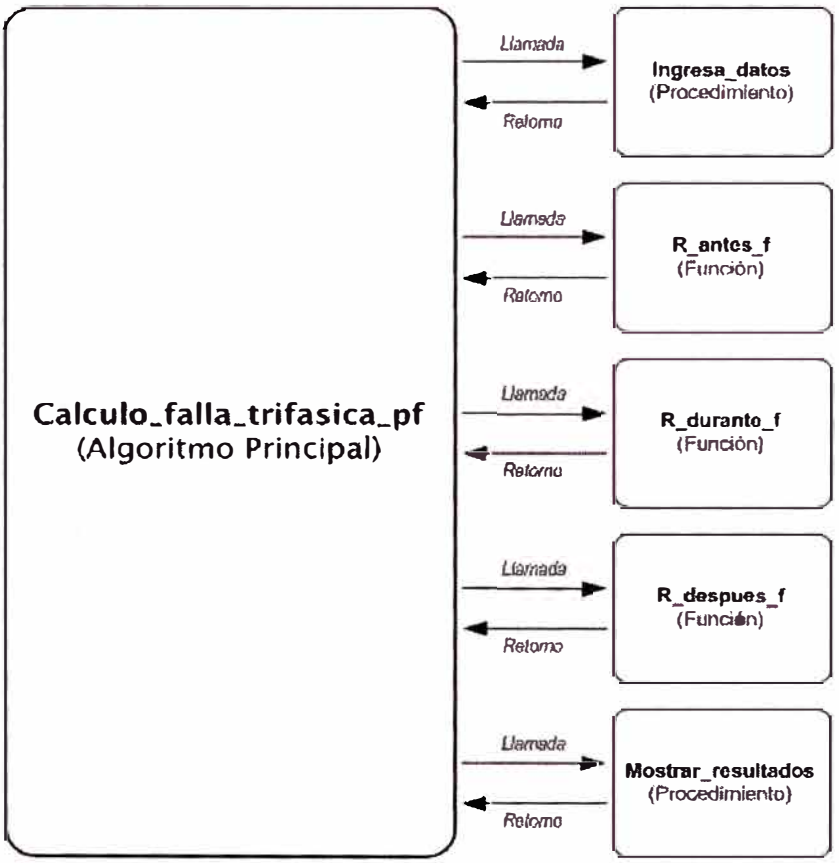


Figura 1.19: Llamadas y retornos entre el algoritmo principal y sus subrutinas.

El desarrollo del pseudocódigo ***Calculo_falla_trifasica_pf*** se muestra a continuación:

```
01: Pseudocódigo  Calculo_falla_trifasica_pf
02:
03: Constantes
04:
05:   PI ← 3.141592654
06:   MAX ← 20
```

```
07:
08: Variables
09:
10:   { Tensiones de generador y barra infinita }
11:
12:     E1, E2 : real
13:
14:   { Reactancias del sistema eléctrico }
15:
16:     Xd, X1, X2, X3 : real
17:
18:   { Reactancias antes, durante y después de la falla }
19:
20:     Xr1, Xr2, Xr3 : real
21:
22:   { Constantes de potencia eléctrica transmitida }
23:
24:     C1, C2, C3 : real
25:
26: Inicio
27:
28:   { Lectura de datos del sistema eléctrico }
29:
30:     Paso N°01:  Llamar_A Ingresa_datos
31:
32:   { Cálculo de reactancias antes, durante y después de la falla }
33:
34:     Paso N°02:  Xr1 ← R_antes_f(Xd,X1,X2,X3)
35:                 Xr2 ← R_durante_f(Xd,X1,X2,X3)
36:                 Xr3 ← R_despues_f(Xd,X1,X2)
37:
38:   { Cálculo de las constantes de potencia eléctrica transmitida }
39:
40:     Paso N°03:  C1 ← E1*E2/Xr1
41:                 C2 ← E1*E2/Xr2
42:                 C3 ← E1*E2/Xr3
43:
44:   { Mostrar resultados }
45:
46:     Paso N°04:  Llamar_A Mostrar_resultados
47:
48: Fin
```

Pseudocódigo 1.3: Calculo_falla_trifasica_pf

En la línea N°1 se establece el nombre del pseudocódigo. Entre las líneas N°3 y N°6 se definen dos constantes: el número π y el número de subintervalos requeridos por el enunciado del Ejemplo 1.7. Entre las líneas N°8 y N°24 se definen las variables a ser utilizadas por el pseudocódigo (que tienen el carácter de variables globales) y que se encuentran convenientemente comentadas. La entrada de datos provenientes del sistema eléctrico se encuentra especificada en la línea N°30, donde se efectúa una llamada al procedimiento ***Ingresa_datos***. El cálculo de las reactancias se lleva a cabo de acuerdo a las expresiones **(1.8)**, **(1.9)** y **(1.10)**, invocando a las funciones ***R_antes_f***, ***R_durante_f*** y ***R_despues_f***, y asignando los valores obtenidos a sus respectivas variables, lo cual se muestra entre las líneas N°34 y N°36. El cálculo de las constantes de potencia eléctrica transmitida se lleva a cabo de acuerdo a la expresión **(1.11)** y se muestra entre las líneas N°40 y N°42. Finalmente se efectúa un reporte de las reactancias y constantes obtenidas, así como el reporte de los valores de ángulo y potencia eléctrica a intervalos de tamaño $\pi/20$, tal y como se muestra en la línea N°46, invocando al procedimiento ***Mostrar_resultados***.

El desarrollo del pseudocódigo ***Ingresa_datos***, se muestra a continuación:

```
01: Procedimiento  Ingresa_datos
02:
03: Inicio
04:
05:   Paso N°01:  Escribir "E1 = "
06:               Leer E1
07:               Escribir "E2 = "
08:               Leer E2
09:               Escribir "Xd = "
10:               Leer Xd
11:               Escribir "X1 = "
12:               Leer X1
13:               Escribir "X2 = "
14:               Leer X2
15:               Escribir "X3 = "
16:               Leer X3
17:
18: Fin_Procedimiento
```

Pseudocódigo 1.4: Procedimiento *Ingresa_datos*

En este procedimiento solamente se efectúa el ingreso de datos, en el cual se asignan valores a las variables globales correspondientes a las tensiones de generador y barra infinita, así como los valores de las reactancias del sistema eléctrico considerado en la Figura 1.10. Estos valores ingresados y asignados a las respectivas variables conforman el retorno que efectúa el procedimiento al algoritmo principal.

En la línea N°34 del algoritmo principal, se desea calcular el valor de la reactancia antes de la falla, representado por la variable global *Xr1*. Para tal fin, se produce una llamada a la función ***R_antes_f***, a la que se le proporciona como datos los valores de las variables globales *Xd*, *X1*, *X2* y *X3*. El desarrollo del pseudocódigo ***R_antes_f***, se muestra a continuación:

```
01: real R_antes_f  (E a:real,E b:real, E c:real,E d:real)
02:
03: Variables
04:
05:   { Variable local de la función }
06:
07:     resultado : real
08:
09: Inicio
10:
11:   { Cálculo de la reactancia antes de la falla }
12:
13:     Paso N°01:  resultado ← (a + b + (c*d)/(c+d))
14:
15:     Paso N°02:  devolver resultado
16:
17: Fin_Función
```

Pseudocódigo 1.5: Función *R_antes_f*

En la línea N°1 se indica que el tipo de dato a devolver por la función es un valor numérico real, que tiene como parámetros de entrada (definidos por una **E**) a las variables *a*, *b*, *c* y *d*, las cuales se constituyen como variables locales de la función y cuyos valores han sido proporcionados por las variables globales *Xd*, *X1*, *X2*, y *X3* respectivamente.

Adicionalmente se tiene una variable denominada *resultado*, al cual se asignará el

cálculo efectuado en la línea N°13. Finalmente en la línea N°15 se devolverá el valor de la variable *resultado*, para que sea asignado a la variable global *Xr1*, tal y como se detalla en la línea N°42 del algoritmo principal.

En la línea N°35 del algoritmo principal, se desea calcular el valor de la reactancia durante la falla, representado por la variable global *Xr2*.. Para tal fin, se produce una llamada a la función ***R_durante_f***, a la que se le proporciona como datos los valores de las variables globales *Xd*, *X1*, *X2* y *X3*. El desarrollo del pseudocódigo ***R_durante_f***, se muestra a continuación:

```
01: real R_durante_f  (E e:real, E f:real, E g:real, E h:real)
02:
03: Variables
04:
05:   { Variable local de la función }
06:
07:     respuesta : real
08:
09: Inicio
10:
11:   { Cálculo de la reactancia durante la falla }
12:
13:     Paso N°01:  respuesta ← (e + f + g + 2*g+(e + f)/h)
14:
15:     Paso N°02:  devolver respuesta
16:
17: Fin_Función
```

Pseudocódigo 1.6: Función *R_durante_f*

En la línea N°1 se indica que el tipo de dato a devolver por la función es un valor numérico real, que tiene como parámetros de entrada (definidos por una **E**) a las variables *e*, *f*, *g* y *h*, las cuales se constituyen como variables locales de la función y cuyos valores han sido proporcionados por las variables globales *Xd*, *X1*, *X2*, y *X3* respectivamente. Adicionalmente se tiene una variable denominada *respuesta*, al cual se asignará el cálculo efectuado en la línea N°13. Finalmente en la línea N°15 se devolverá el valor de la variable

respuesta, para que este sea asignado a la variable global *Xr2*, tal y como se detalla en la línea N°43 del algoritmo principal.

En la línea N°36 del algoritmo principal, se desea calcular el valor de la reactancia después de la falla, representado por la variable global *Xr3*. Para tal fin, se produce una llamada a la función ***R_despues_f***, a la que se le proporciona como datos los valores de las variables globales *Xd*, *X1* y *X2*. El desarrollo del pseudocódigo ***R_después_f***, se muestra a continuación:

```
01: real R_despues_f (E Xd:real, E X1:real, E X2:real)
02:
03: Inicio
04:
05:   { Cálculo de la reactancia después de la falla }
06:
07:     Paso N°01:  devolver (Xd + X1 + X2)
08:
09: Fin_Función
```

Pseudocódigo 1.7: Función *R_despues_f*

En la línea N°1 se indica que el tipo de dato a devolver por la función es un valor numérico real, que tiene como parámetros de entrada (definidos por una **E**) a las variables *Xd*, *X1* y *X2*, las cuales se constituyen como variables locales de la función y cuyos valores han sido proporcionados por las variables globales *Xd*, *X1* y *X2* respectivamente (cabe aclarar que si bien es cierto que ambos conjuntos de variables presentan identificadores idénticos, tienen un ámbito diferenciado). En la línea N°7 se devolverá directamente el valor calculado, para que este sea asignado a la variable global *Xr3*, tal y como se detalla en la línea N°36 del algoritmo principal.

En la línea N°46 del algoritmo principal, se desea mostrar el resultado obtenido por los cálculos efectuados previamente, así como las curvas de potencia consideradas en el Ejemplo 1.7. Para tal fin, se produce una llamada al procedimiento ***Mostrar_resultados***, el cual toma los valores de las variables globales consideradas en el algoritmo principal

(líneas N°12, N°16, N°20 y N°24). El desarrollo del pseudocódigo ***Mostrar_resultados*** se muestra a continuación:

```
01: Procedimiento  Mostrar_resultados
02:
03: Variables
04:
05:   { Variables de potencia eléctrica transmitida }
06:
07:     Pe1, Pe2, Pe3 : real
08:
09:   { Ángulo de potencia eléctrica transmitida }
10:
11:     delta : real
12:
13:   { Contador }
14:
15:     i : entero
16:
17: Inicio
18:
19:   { Reactancias antes, durante y despues de la falla }
20:
21:     Paso N°01:  Escribir "Xr1 = " , Xr1
22:                 Escribir "Xr2 = " , Xr2
23:                 Escribir "Xr3 = " , Xr3
24:
25:   { Constantes de potencia electrica }
26:
27:     Paso N°02:  Escribir "C1 = " , C1
28:                 Escribir "C2 = " , C2
29:                 Escribir "C3 = " , C3
30:
31:   { Curvas de potencia eléctrica transmitida }
32:
33:     Paso N°03:  delta ← 0
34:
35:     Paso N°04:  Desde  i ← 1  hasta (MAX+1) hacer
36:
37:                   Paso N°05:  Pe1 ← C1*sen(delta)
38:                               Pe2 ← C2*sen(delta)
39:                               Pe3 ← C3*sen(delta)
40:
```

```
41:                    Paso N°06:  Escribir  i, delta, Pe1, Pe2, Pe3
42:
43:                    Paso N°07:  delta ← delta + PI/MAX
44:
45:                  Fin_Desde
46:
47: Fin_Procedimiento
```

Pseudocódigo 1.8: Mostrar resultados

Entre las líneas N°5 y N°15 se tienen definidas variables de tipo local, las cuales tendrán un ámbito y permanencia a lo largo de la ejecución de esta subrutina. Entre las líneas N°21 y N°23 se muestra el valor de las reactancias antes, durante y después de la falla. Entre las líneas N°27 al N°29 se muestra el valor de las constantes de potencia eléctrica. Finalmente, entre las líneas N°33 y N°45 se muestran los valores correspondientes a las curvas de potencia transmitida.

Implementando el pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se tendrá lo siguiente:

```
001: #include <conio.h>
002: #include <stdio.h>
003: #include <math.h>
004:
005: /* Definicion de constantes */
006:
007:    const  int MAX = 20 ;
008:
009: /* Definicion de variables */
010:
011:    /* Tensiones del generador y de barra infinita      */
012:
013:    float  E1, E2 ;
014:
015:    /* Reactancias del sistema electrico               */
016:
017:    float  Xd, X1, X2, X3 ;
018:
019:    /* Reactancias antes, durante y despues de la falla */
020:
021:    float  Xr1, Xr2, Xr3 ;
022:
```

```
023:    /* Constantes de potencia electrica transmitida     */
024:
025:    float  C1, C2, C3 ;
026:
027: /* Definicion de procedimientos y funciones */
028:
029:    void Ingresa_datos() ;
030:    float R_antes_f(float, float, float, float) ;
031:    float R_durante_f(float, float, float, float) ;
032:    float R_despues_f(float, float, float) ;
033:    void Mostrar_resultados() ;
034:
035:
036: void main()
037:  {
038:
039:    /* Lectura de datos del sistema electrico */
040:
041:    clrscr() ;
042:    Ingresa_datos() ;
043:
044:    /* Calculo de reactancias antes, durante y despues de la falla */
045:
046:    Xr1 = R_antes_f(Xd,X1,X2,X3) ;
047:    Xr2 = R_durante_f(Xd,X1,X2,X3) ;
048:    Xr3 = R_despues_f(Xd,X1,X2) ;
049:
050:    /* Calculo de las constantes de potencia electrica transmitida */
051:
052:    C1 = E1*E2/Xr1 ;
053:    C2 = E1*E2/Xr2 ;
054:    C3 = E1*E2/Xr3 ;
055:
056:    /* Mostrar resultados */
057:
058:    clrscr() ;
059:    printf ("*** Resultados ***\n\n") ;
060:    Mostrar_resultados() ;
061:    printf ("----------------------------------------\n\n") ;
062:    printf ("Presione cualquier tecla para salir ...") ;
063:    getch() ;
064:
065:  }
066:
```

```
067:
068: /**/void Ingresa_datos()
069:     {
070:       printf ("*** Datos del Sistema Electrico ***\n\n") ;
071:       printf ("Tensiones del Sistema Electrico ...\n\n") ;
072:       printf ("E1 (p.u.) = ") ; scanf("%f", &E1) ;
073:       printf ("E2 (p.u.) = ") ; scanf("%f", &E2) ;
074:       printf ("\nReactancias del Sistema Electrico ...\n\n") ;
075:       printf ("Xd (p.u.) = ") ; scanf("%f", &Xd) ;
076:       printf ("X1 (p.u.) = ") ; scanf("%f", &X1) ;
077:       printf ("X2 (p.u.) = ") ; scanf("%f", &X2) ;
078:       printf ("X3 (p.u.) = ") ; scanf("%f", &X3) ;
079:       printf ("\nPresione cualquier tecla para continuar ...\n\n") ;
080:       getch() ;
081:     }
082:
083: /**/float R_antes_f(float a, float b, float c, float d)
084:     {
085:       float  resultado ;    // Variable Local
086:       resultado = (a + b + (c*d)/(c+d)) ;
087:       return resultado ;
088:     }
089:
090: /**/float R_durante_f(float e, float f, float g, float h)
091:     {
092:       float  respuesta ;    // Variable Local
093:       respuesta = (e + f + g + 2*g*(e+f)/h) ;
094:       return respuesta ;
095:     }
096:
097: /**/float R_despues_f(float Xd, float X1, float X2)
098:     {
099:       return (Xd + X1 + X2) ;
100:     }
101:
102: /**/void Mostrar_resultados()
103:     {
104:
105:       /* Variables de potencia electrica transmitida */
106:
107:       float  Pe1, Pe2, Pe3 ;
108:
109:       /* Angulos de potencia electrica transmitida    */
110:
```

```
111:        float  delta ;
112:
113:        /* Contador */
114:
115:        int    i ;
116:
117:        /* Reactancias antes, durante y despues de la falla */
118:
119:        printf ("Reactancias ... \n\n") ;
120:        printf ("- Antes de la falla   : Xr1 = %10.4f\n",Xr1) ;
121:        printf ("- Durante la falla    : Xr2 = %10.4f\n",Xr2) ;
122:        printf ("- Despues de la falla : Xr3 = %10.4f\n\n",Xr3) ;
123:
124:        /* Constantes de potencia electrica transmitida */
125:
126:        printf ("Constantes de potencia electrica transmitida...\n\n");
127:        printf ("- Antes de la falla   : C1 = %10.4f\n",C1) ;
128:        printf ("- Durante la falla    : C2 = %10.4f\n",C2) ;
129:        printf ("- Despues de la falla : C3 = %10.4f\n\n",C3) ;
130:
131:        /* Curvas de potencia transmitida */
132:
133:        printf ("Curvas de potencia transmitida ...\n\n") ;
134:        printf ("-----------------------------------------\n") ;
135:        printf ("| i  | Angulo |   P1   |   P2   |   P3   |\n") ;
136:        printf ("-----------------------------------------\n") ;
137:        delta = 0 ;
138:        for (i=0;i<=MAX;i++)
139:         {
140:           Pe1 = C1*sin(delta) ;
141:           Pe2 = C2*sin(delta) ;
142:           Pe3 = C3*sin(delta) ;
143:           printf ("| %2i | %6.2f | %6.3f | %6.3f | %6.3f …
                        |\n",i,delta*180/M_PI,Pe1,Pe2,Pe3) ;
144:           delta = delta + M_PI/MAX ;
145:         }
146:     }
```

Programa 1.3: CALFT_PF.CPP

A diferencia de lo desarrollado a nivel algorítmico, cuando se lleva a cabo la implementación en un lenguaje de programación, los procedimientos y funciones se definen de acuerdo a la sintaxis y a la semántica de dicho lenguaje.

En el caso del lenguaje C, se parte del hecho de que siempre se va a trabajar con funciones. Para dicho fin se llevarán a cabo las definiciones de las estructuras de dichas funciones antes de la definición del cuerpo principal del programa, tal y como se muestra entre las líneas N°29 y N°33.

Dado que los procedimientos no devuelven valor alguno, su definición empieza con la palabra reservada **`void`** (líneas N°29, N°33, N°68 y N°102). Cabe resaltar que las funciones que requieren parámetros, debe indicarse en su declaración los tipos de datos que se van a tomar en cuenta (líneas N°30 al N°32). Para que una función implementada en lenguaje C devuelva un valor, se requiere el uso de la palabra reservada **`return`** (líneas N°87, N°94 y N°99).

El cuerpo principal del programa, desde donde se llevan a cabo las llamadas a las subrutinas, empieza con **`main()`**. Como la estructura del lenguaje C está basada en funciones, se requiere que inclusive el cuerpo principal devuelva un tipo de dato, por lo cual el cuerpo principal del programa empieza con **`void main()`**.

Al iniciarse la ejecución del cuerpo principal de las subrutinas definidas, se tiene que el conjunto de variables locales a cada función empieza su existencia con un valor inicial por defecto. En el caso del tipo numérico **`float`**, sus variables se inicializan en 0.0, mientras que las variables del tipo numérico **`int`**, se inicializan en un número entero positivo aleatorio diferente de cero.

La entrada y salida del programa son las mismas que las mostradas en las Figuras 1.14 (entrada de los datos del sistema eléctrico) y 1.15 (reporte de resultados).

CAPÍTULO II
ESTRUCTURAS DE DATOS ESTÁTICAS

2.1 Introducción

Una *estructura de datos* es una colección de datos que pueden ser caracterizados por su organización y las operaciones que se definen en ella.

Las estructuras de datos son muy importantes en el diseño e implementación de sistemas computacionales. Tal y como se mostró en la Figura 1.2 (Capítulo I), los tipos de datos más frecuentes utilizados en los diferentes lenguajes de programación, son los siguientes:

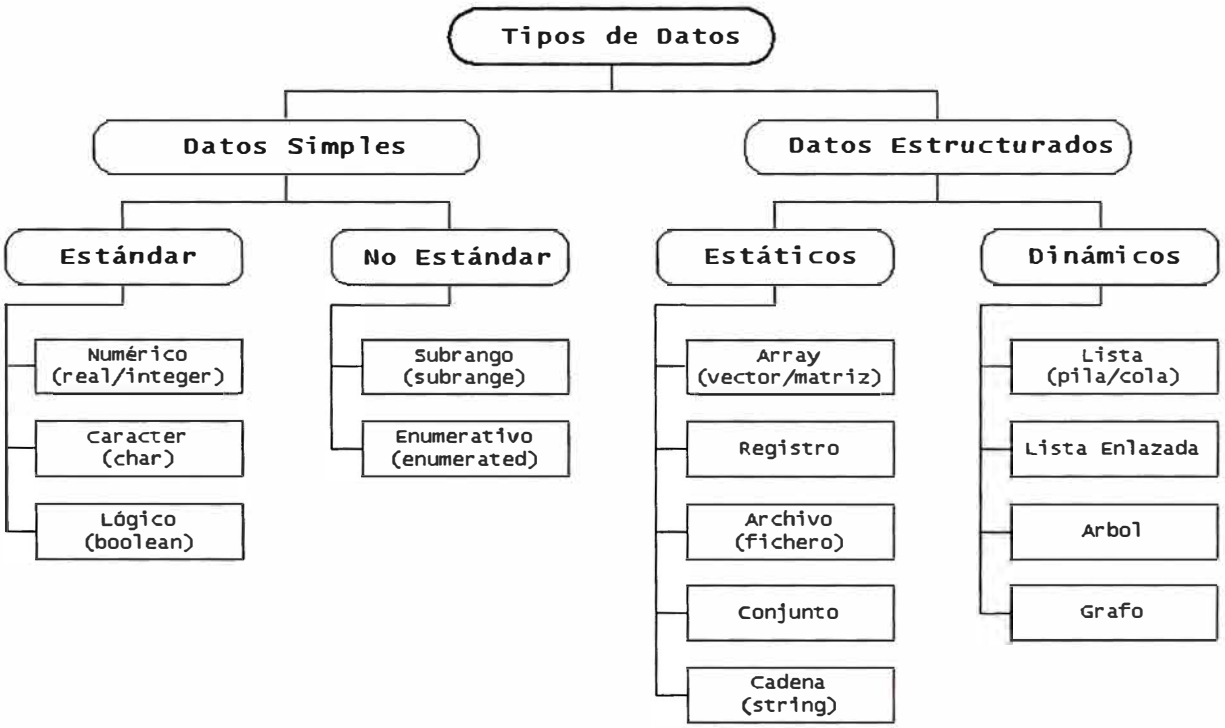


Figura 2.1: Clasificación de los tipos de datos

Los tipos de datos *simples* o *primitivos* significan que no están compuestos de otras estructuras de datos; los más frecuentes y utilizados por casi todos los lenguajes de programación son: *enteros*, *reales* y *carácter* (*char*), siendo los tipos *lógicos*, *subrango* y *enumerativos* propios de los lenguajes estructurados como Pascal. Los tipos de datos

compuestos están basados en tipos de datos primitivos; el ejemplo más representativo es la *cadena* (*string*) de caracteres.

Los tipos de datos simples pueden ser organizados en diferentes estructuras de datos: *estáticas* y *dinámicas*. Las **estructuras de datos estáticas** son aquellas en las que el tamaño ocupado en memoria se define antes de que el programa se ejecute y no puede modificarse dicho tamaño durante la ejecución del mismo. Estas estructuras están implementadas en casi todos los lenguajes: *array* (vectores/tablas-matrices), *registros*, *archivos* (los *conjuntos* son específicos del lenguaje Pascal).

Las **estructuras de datos dinámicas** no tienen limitaciones o restricciones en el tamaño de memoria ocupada que son propias de las estructuras estáticas. Mediante el uso de un tipo de dato específico, denominado *puntero*, es posible construir estructuras de datos dinámicas que son soportadas por la mayoría de los lenguajes, y en aquellos que sí tienen estas características ofrecen soluciones eficaces y efectivas en la solución de problemas complejos. Las estructuras dinámicas por excelencia son las *listas* – enlazadas, pilas, colas -, *árboles* – binarios, árbol-b, de búsqueda binaria – y *grafos*.

La elección del tipo de estructura de datos idónea a cada aplicación dependerá esencialmente del tipo de aplicación y, en menor medida, del lenguaje, ya que en aquellos en que no está implementada una estructura deberá ser simulada con el algoritmo adecuado, dependiendo del propio algoritmo y las características del lenguaje su fácil o difícil solución.

Una característica importante que diferencia a los tipos de datos es la siguiente: los tipos de datos simples tienen como característica común que cada variable representa a un elemento; los tipos de datos estructurados tienen como característica común que un *identificador* (nombre) puede representar múltiples datos individuales, pudiendo cada uno de éstos ser referenciado independientemente.

2.2 Arreglos (arrays)

Un **array** (vector o matriz) es un conjunto finito y ordenado de elementos de un mismo tipo de dato. Bajo estos términos, un *vector* es un **array unidimensional** y una *matriz* es un **array bidimensional**, cuyos elementos son referenciados por índices que no son más que números enteros positivos encerrados entre corchetes.

2.2.1 Arrays unidimensionales

Para definir algorítmicamente un array unidimensional, se tendrá la siguiente expresión:

<nombre_de_array> : array [dimensión] de <tipo_dato> **(2.1)**

La forma de referirnos a un elemento de un vector unidimensional se efectúa de la siguiente manera:

<nombre_de_array>[i] **(2.2)**

donde ***i*** es el i-ésimo elemento de un array unidimensional de dimensión ***n***.

Las notaciones empleadas para efectuar operaciones de asignación, lectura y escritura, se muestran a continuación en la Tabla 2.1:

Tabla 2.1: Operaciones de asignación, lectura y escritura en arrays unidimensionales

Operación	Notación
Asignación	***<nombre_array>[i] ← valor, variable***
Lectura	***Leer <nombre_array>[i]***
Escritura	***Escribir <nombre_array>[i]***

Nota: ***i*** es el i-ésimo elemento de un array unidimensional ***n***.

Ejemplo 2.1: De acuerdo al enunciado del Ejemplo 1.7, elaborar un pseudocódigo que utilice arrays unidimensionales para almacenar los valores correspondientes a los ángulos de potencia y las curvas de potencia transmitida.

Considerando los parámetros del sistema eléctrico mostrado en la Figura 1.10 (Capítulo II), el pseudocódigo que concuerda con los requerimientos del enunciado es el que se muestra a continuación:

```
001: Pseudocódigo  Calculo_falla_trifasica_array
002:
003: Constantes
004:
005:   PI ← 3.141592654
006:   MAX ← 20
007:
008: Variables
009:
010:   { Tensiones de generador y barra infinita }
011:
012:     E1, E2 : real
013:
014:   { Reactancias del sistema eléctrico }
015:
016:     Xd, X1, X2, X3 : real
017:
018:   { Reactancias antes, durante y después de la falla }
019:
020:     Xr1, Xr2, Xr3 : real
021:
022:   { Constantes de potencia eléctrica transmitida }
023:
024:     C1, C2, C3 : real
025:
026:   { Curvas de potencia eléctrica transmitida }
027:
028:     Pe1, Pe2, Pe3 : array [1..MAX+1] de real
029:
030:   { Ángulos de potencia }
031:
032:     delta : array [1..MAX+1] de real
033:
034:   { Contador }
035:
036:     i : entero
037:
038: Inicio
```

```
039:
040:   { Lectura de datos del sistema eléctrico }
041:
042:     Paso N°01:  Escribir "E1 = "
043:                 Leer E1
044:                 Escribir "E2 = "
045:                 Leer E2
046:                 Escribir "Xd = "
047:                 Leer Xd
048:                 Escribir "X1 = "
049:                 Leer X1
050:                 Escribir "X2 = "
051:                 Leer X2
052:                 Escribir "X3 = "
053:                 Leer X3
054:
055:   { Cálculo de reactancias antes, durante y después de la falla }
056:
057:     Paso N°02:  Xr1 ← Xd + X1 + (X2*X3)/(X2+X3)
058:                 Xr2 ← Xd + X1 + X2 + 2*X2*(Xd+X1)/X3
059:                 Xr3 ← Xd + X1 + X2
060:
061:   { Cálculo de las constantes de potencia eléctrica transmitida }
062:
063:     Paso N°03:  C1 ← E1*E2/Xr1
064:                 C2 ← E1*E2/Xr2
065:                 C3 ← E1*E2/Xr3
066:
067:   { Cálculo de las curvas de potencia eléctrica transmitida }
068:
069:     Paso N°04:  delta[1] ← 0
070:
071:     Paso N°05:  Desde  i ← 1  hasta (MAX+1) hacer
072:
073:                   Paso N°06:  Pe1[i] ← C1*sen(delta[i])
074:                               Pe2[i] ← C2*sen(delta[i])
075:                               Pe3[i] ← C3*sen(delta[i])
076:
077:                   Paso N°07:  Si (i<(MAX+1))
078:                                 Entonces
079:                                   delta[i+1] ← delta[i] + PI/MAX
080:                               Fin_Si
081:
```

```
082:                Fin_Desde
083:
084:   { Mostrar resultados }
085:
086:   { Reactancias antes, durante y después de la falla }
087:
088:     Paso N°08:  Escribir "Xr1 = " , Xr1
089:                 Escribir "Xr2 = " , Xr2
090:                 Escribir "Xr3 = " , Xr3
091:                 Escribir "C1 = " , C1
092:                 Escribir "C2 = " , C2
093:                 Escribir "C3 = " , C3
094:
095:   { Curvas de potencia eléctrica transmitida }
096:
097:     Paso N°09:  Escribir "Curvas de Potencia"
098:
099:     Paso N°10:  Desde  i ← 1  hasta (MAX+1) hacer
100:
101:                   Paso N°11:  Escribir  i,delta[i],Pe1[i],Pe2[i],…
                                             Pe3[i]
102:
103:                 Fin_Desde
104:
105: Fin
```

Pseudocódigo 2.1: Calcula_falla_trifasica_array

En la línea N°1 se establece el nombre del pseudocódigo. Entre las líneas N°3 y N°6 se definen dos constantes: el número π y el número de subintervalos requeridos. Entre las líneas N°8 y N°36 se definen las variables a ser utilizadas por el pseudocódigo y que se encuentran convenientemente comentadas. Cabe resaltar que en las líneas N°28 y N°32 se declaran variables como arrays unidimensionales.

La entrada de los datos provenientes del sistema eléctrico se encuentra especificada entre las líneas N°40 y N°53. El cálculo de las reactancias se lleva a cabo de acuerdo a las expresiones **(1.8)**, **(1.9)** y **(1.10)** y se muestra entre las líneas N°55 y N°59. El cálculo de las constantes de potencia eléctrica transmitida se lleva a cabo de acuerdo a la expresión **(1.11)** y se muestra entre las líneas N°61 y N°65.

Entre las líneas N°67 y N°82 se muestran las sentencias algorítmicas que permiten calcular los valores correspondientes a los ángulos de potencia y las curvas de potencia transmitida. Los subíndices de los arrays unidimensionales definidos empiezan en 1. Mediante la estructura de control repetitiva *desde-hasta* se generan los valores solicitados, donde el valor del contador que hace referencia a los subíndices (*i*), no puede exceder el subíndice de máximo valor (MAX+1).

Finalmente, entre las líneas N°84 y N°103 se efectúa un reporte de resultados, mostrando las reactancias y constantes obtenidas, así como los valores de ángulo y potencia eléctrica a intervalos de tamaño $\pi/20$ mediante un bucle *desde-hasta*, tal y como se muestra entre las líneas N°99 y N°103.

Implementando el pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se tendrá lo siguiente:

```
001: #include <conio.h>
002: #include <stdio.h>
003: #include <math.h>
004:
005: /* Definicion de constantes */
006:
007:    const  int MAX = 20 ;
008:
009: /* Definicion de variables */
010:
011:    /* Tensiones del generador y de barra infinita      */
012:
013:    float  E1, E2 ;
014:
015:    /* Reactancias del sistema electrico               */
016:
017:    float  Xd, X1, X2, X3 ;
018:
019:    /* Reactancias antes, durante y despues de la falla */
020:
021:    float  Xr1, Xr2, Xr3 ;
022:
023:    /* Constantes de potencia electrica transmitida     */
024:
025:    float  C1, C2, C3 ;
```

```
026:
027:    /* Potencia electrica transmitida                */
028:
029:    float  P1[MAX+1] ,     // Antes de la falla
030:           P2[MAX+1] ,     // Durante la falla
031:           P3[MAX+1]   ;   // Despues de la falla
032:
033:    /* Angulos de potencia                           */
034:
035:    float  delta[MAX+1] ;
036:
037:    /* Contador                                      */
038:
039:    int i ;
040:
041: void main()
042:  {
043:
044:    /* Lectura de datos del sistema electrico */
045:
046:    clrscr() ;
047:    printf ("*** Datos del Sistema Electrico ***\n\n") ;
048:    printf ("Tensiones del Sistema Electrico ...\n\n") ;
049:    printf ("E1 (p.u.) = ") ; scanf("%f", &E1) ;
050:    printf ("E2 (p.u.) = ") ; scanf("%f", &E2) ;
051:    printf ("\nReactancias del Sistema Electrico ...\n\n") ;
052:    printf ("Xd (p.u.) = ") ; scanf("%f", &Xd) ;
053:    printf ("X1 (p.u.) = ") ; scanf("%f", &X1) ;
054:    printf ("X2 (p.u.) = ") ; scanf("%f", &X2) ;
055:    printf ("X3 (p.u.) = ") ; scanf("%f", &X3) ;
056:    printf ("\nPresione cualquier tecla para continuar ...\n\n") ;
057:    getch() ;
058:
059:    /* Calculo de reactancias */
060:
061:    Xr1 = Xd + X1 + (X2*X3)/(X2+X3) ;
062:    Xr2 = Xd + X1 + X2 + 2*X2*(Xd+X1)/X3 ;
063:    Xr3 = Xd + X1 + X2 ;
064:
065:    /* Calculo de las constantes de potencia electrica transmitida */
066:
067:    C1 = E1*E2/Xr1 ;
068:    C2 = E1*E2/Xr2 ;
069:    C3 = E1*E2/Xr3 ;
```

```
070:
071:    /* Calculo de las curvas de potencia transmitida */
072:
073:    delta[0] = 0 ;
074:
075:    for (i=0;i<=MAX;i++)
076:     {
077:       P1[i] = C1*sin(delta[i]) ;
078:       P2[i] = C2*sin(delta[i]) ;
079:       P3[i] = C3*sin(delta[i]) ;
080:
081:       if (i<MAX)  delta[i+1] = delta[i] + M_PI/MAX ;
082:     }
083:
084:    /* Mostrar resultados */
085:
086:    clrscr() ;
087:    printf ("*** Resultados ***\n\n") ;
088:
089:    /* Reactancias antes, durante y despues de la falla */
090:
091:    printf ("Reactancias ... \n\n") ;
092:    printf ("- Antes de la falla   :  Xr1 = %10.4f\n",Xr1) ;
093:    printf ("- Durante la falla    :  Xr2 = %10.4f\n",Xr2) ;
094:    printf ("- Despues de la falla :  Xr3 = %10.4f\n\n",Xr3) ;
095:
096:    /* Constantes de potencia electrica transmitida */
097:
098:    printf ("Constantes de potencia electrica transmitida ...\n\n");
099:    printf ("- Antes de la falla   :  C1 = %10.4f\n",C1) ;
100:    printf ("- Durante la falla    :  C2 = %10.4f\n",C2) ;
101:    printf ("- Despues de la falla :  C3 = %10.4f\n\n",C3) ;
102:
103:    /* Curvas de potencia transmitida */
104:
105:    printf ("Curvas de potencia transmitida ...\n\n") ;
106:
107:    printf ("-----------------------------------------\n") ;
108:    printf ("| i  | Angulo |   P1   |   P2   |   P3   |\n") ;
109:    printf ("-----------------------------------------\n") ;
110:
111:    for (i=0;i<=MAX;i++)
112:     {
113:       printf ("| %2i | %6.2f | %6.3f | %6.3f | %6.3f …
```

```
                    |\n",i,delta[i]*180/M_PI,P1[i],P2[i],P3[i]) ;
114:     }
115:    printf ("-----------------------------------------\n\n") ;
116:    printf ("Presione cualquier tecla para salir ...") ;
117:    getch() ;
118:
119:  }
```

Programa 2.1: CALFT_A.CPP

A diferencia de lo desarrollado a nivel algorítmico, cuando se lleva a cabo la declaración y utilización de los arrays unidimensionales en el lenguaje de programación C, se debe tomar en cuenta lo siguiente:

- El formato de declaración de un array unidimensional es el siguiente:

  tipo nombre_de_variable[tamaño] ; **(2.3)**

- Todos los arrays unidimensionales tienen el 0 como índice de su primer elemento. Por tanto, cuando se escribe **`int vector[10]`** se está declarando un array de números enteros que tiene diez elementos, desde **`vector[0]`** hasta **`vector[9]`**.

Para calcular los valores correspondientes, se hace referencia a cada uno de los elementos del array unidimensional, tal y como se lleva a cabo entre las líneas N°73 y N°82, mediante la sentencia de control ***for***.

Finalmente, entre las líneas N°84 y N°114, se tiene un reporte de las reactancias y constantes de potencia obtenidas, así como las curvas de potencia transmitida, mostrándose en pantalla los valores correspondientes de cada conjunto de elementos, mediante la sentencia de control ***for***.

2.2.2 Arrays bidimensionales

El *array bidimensional* se puede considerar como un vector de vectores. Para definir algorítmicamente un array bidimensional, se tendrá la siguiente expresión:

<nombre_de_array> : array [dimensión,dimensión] de <tipo_dato> **(2.4)**

La forma de referirnos a un elemento de un vector bidimensional se efectúa de la siguiente manera:

<nombre_de_array>[i][j] **(2.5)**

donde ***i*** es la i-ésima fila, y ***j*** es la j-ésima columna del array bidimensional de dimensión ***m×n***. Las notaciones empleadas para efectuar operaciones de asignación, lectura y escritura, se muestran a continuación en la Tabla 2.2:

Tabla 2.2: Operaciones de asignación, lectura y escritura en arrays bidimensionales.

Operación	Notación
Asignación	*<nombre_array>[i][j] ← valor, variable*
Lectura	*Leer <nombre_array>[i][j]*
Escritura	*Escribir <nombre_array>[i][j]*

Nota: ***i*** es el i-ésima fila y j es la j-ésima columna de un elemento de un array bidimensional ***m×n***.

Ejemplo 2.2: En la Figura 2.2 se muestra el diagrama unifilar de un pequeño sistema de potencia. El diagrama de impedancias correspondiente (especificado en p.u.) se muestra en la Figura 2.3. Desarrollar la matriz de admitancias de nodo.

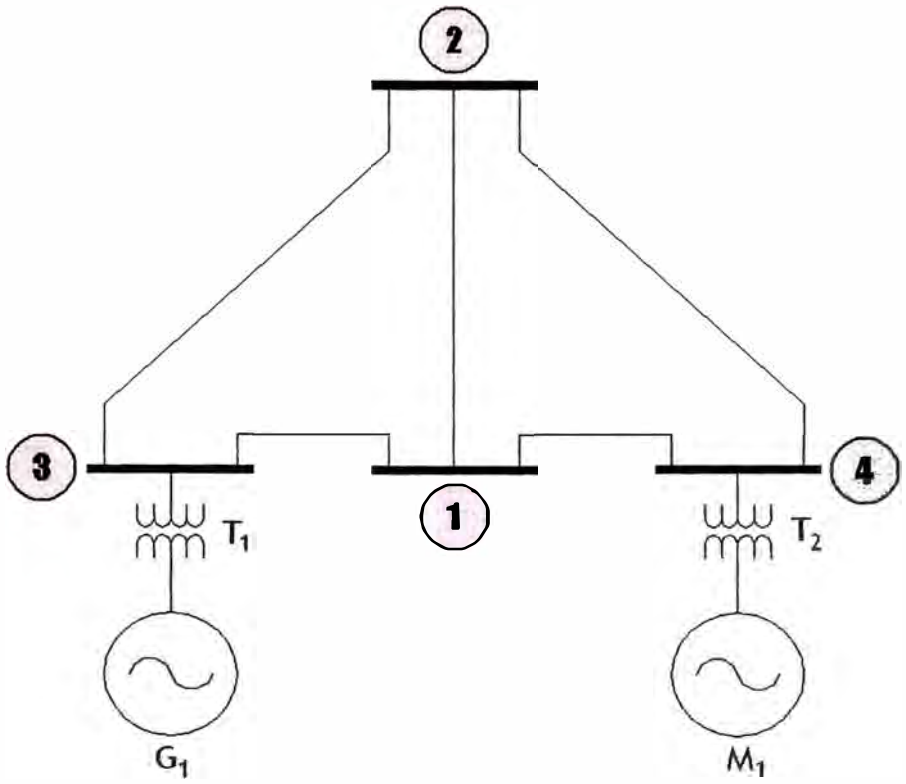


Figura 2.2: Diagrama unifilar del sistema de cuatro barras.

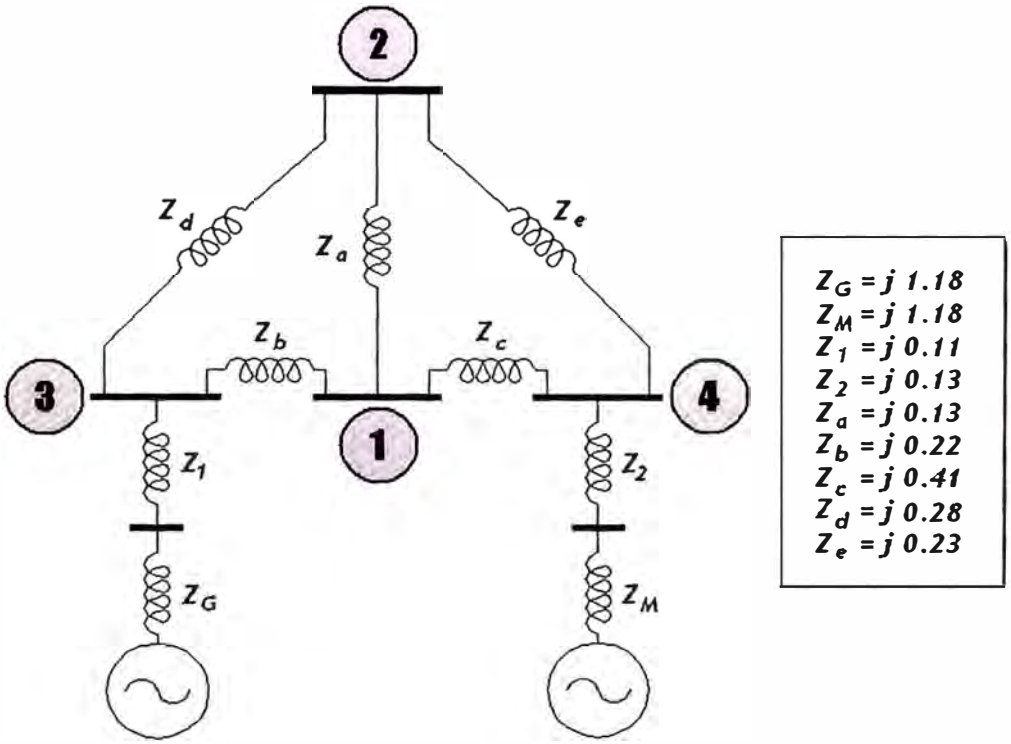


Figura 2.3: Diagrama de reactancias para la Figura 2.2.

Las reactancias del motor y del generador se pueden combinar con las respectivas reactancias de los transformadores de elevación de tensión. Para hallar la matriz de admitancias respectiva, utilizaremos las siguientes expresiones:

$$
\begin{aligned}
Y_{11} &= \frac{1}{Z_a} + \frac{1}{Z_b} + \frac{1}{Z_c} \\
Y_{12} &= -\frac{1}{Z_a} \\
Y_{13} &= -\frac{1}{Z_b} \\
Y_{14} &= -\frac{1}{Z_c} \\
Y_{21} &= -\frac{1}{Z_a} \\
Y_{22} &= \frac{1}{Z_a} + \frac{1}{Z_d} + \frac{1}{Z_e} \\
Y_{23} &= -\frac{1}{Z_d} \\
Y_{24} &= -\frac{1}{Z_e} \\
Y_{31} &= -\frac{1}{Z_b} \\
Y_{32} &= -\frac{1}{Z_d} \\
Y_{33} &= \frac{1}{Z_b} + \frac{1}{Z_d} + \frac{1}{Z_1 + Z_G} \\
Y_{34} &= 0 \\
Y_{41} &= -\frac{1}{Z_c} \\
Y_{42} &= -\frac{1}{Z_e} \\
Y_{43} &= 0 \\
Y_{44} &= \frac{1}{Z_c} + \frac{1}{Z_e} + \frac{1}{Z_2 + Z_M}
\end{aligned}
\qquad \textbf{(2.6)}
$$

De acuerdo al conjunto de expresiones anterior, la matriz de admitancia será:

$$\begin{bmatrix} \frac{1}{Z_a}+\frac{1}{Z_b}+\frac{1}{Z_c} & -\frac{1}{Z_a} & -\frac{1}{Z_b} & -\frac{1}{Z_c} \\ -\frac{1}{Z_a} & \frac{1}{Z_a}+\frac{1}{Z_d}+\frac{1}{Z_e} & -\frac{1}{Z_d} & -\frac{1}{Z_e} \\ -\frac{1}{Z_b} & -\frac{1}{Z_d} & \frac{1}{Z_b}+\frac{1}{Z_d}+\frac{1}{Z_1+Z_G} & 0 \\ -\frac{1}{Z_c} & -\frac{1}{Z_e} & 0 & \frac{1}{Z_c}+\frac{1}{Z_e}+\frac{1}{Z_2+Z_M} \end{bmatrix} \quad \textbf{(2.7)}$$

El pseudocódigo que concuerda con los requerimientos del enunciado es el que se muestra a continuación:

```
001: Pseudocódigo  Calculo_matriz_admitancia
002:
003: Variables
004:
005:   { Tensiones de generador y barra infinita }
006:
007:     Za, Zb, Zc, Zd, Ze, Z1, Z2, ZG, ZM : real
008:
009:   { Matriz de admitancias }
010:
011:     Y : array [1..4,1..4] de real
012:
013:   { Contador }
014:
015:     i, j : entero
016:
017: Inicio
018:
019:   { Lectura de datos del sistema eléctrico }
020:
021:     Paso N°01:  Escribir "Za = "
022:                 Leer Za
023:                 Escribir "Zb = "
024:                 Leer Zb
```

```
025:                Escribir "Zc = "
026:                Leer Zc
027:                Escribir "Zd = "
028:                Leer Zd
029:                Escribir "Ze = "
030:                Leer Ze
031:                Escribir "Z1 = "
032:                Leer Z1
033:                Escribir "Z2 = "
034:                Leer Z2
035:                Escribir "ZG = "
036:                Leer ZG
037:                Escribir "ZM = "
038:                Leer ZM
039:
040:  { Cálculo de la matriz de admitancia }
041:
042:    Paso N°02:  Y[1][1] ← 1/Za + 1/Zb + 1/Zc
043:                Y[1][2] ← 1/Za
044:                Y[1][3] ← 1/Zb
045:                Y[1][4] ← 1/Zc
046:                Y[2][1] ← 1/Za
047:                Y[2][2] ← 1/Za + 1/Zd + 1/Ze
048:                Y[2][3] ← 1/Zd
049:                Y[2][4] ← 1/Ze
050:                Y[3][1] ← 1/Zb
051:                Y[3][2] ← 1/Zd
052:                Y[3][3] ← 1/Zb + 1/Zd + 1/(Z1+ZG)
053:                Y[3][4] ← 0
054:                Y[4][1] ← 1/Zc
055:                Y[4][2] ← 1/Ze
056:                Y[4][3] ← 0
057:                Y[4][4] ← 1/Zc + 1/Ze + 1/(Z2+ZM)
058:
059:  { Mostrar Resultados }
060:
061:    Paso N°03:  Desde  i ← 1  hasta 4 hacer
062:
063:                  Paso N°04:  Desde j ← 1  hasta 4 hacer
064:
065:                                Paso N°05:  Si (i=j)
066:                                              Entonces
```

```
067:                                                  Escribir " -j "
068:                                               Si_no
069:                                                 Escribir " j "
070:                                              Fin_Si
071:
072:                                  Paso N°06:  Escribir Y[i][j]
073:
074:                               Fin_Desde
075:
076:                     Fin_Desde
077:
078: Fin
```

Pseudocódigo 2.2: Calculo_matriz_admitancia

En la línea N°1 se establece el nombre del pseudocódigo. Entre las líneas N°5 y N°15 se definen las variables a ser utilizadas por el pseudocódigo y que se encuentran convenientemente comentadas. Cabe resaltar que en la línea N°11 se declara la variable ***Y*** como array bidimensional (o matriz).

La entrada de los datos provenientes del sistema eléctrico se encuentra especificadas entre las líneas N°19 y N°38. El cálculo de los componentes de la matriz de admitancias se lleva a cabo de acuerdo a la expresión **(2.7)** y se muestra entre las líneas N°42 y N°57. Finalmente, entre las líneas N°59 y N°76 se efectúa un reporte de resultados, mostrando la matriz de admitancia obtenida, mediante un bucle *desde-hasta* de tipo *anidado* (son bucles que se encuentran uno dentro de otro), dado que se efectúa un "recorrido" de filas y columnas para mostrar cada uno de los valores correspondientes de la matriz de orden 4. En este algoritmo hemos trabajado con números reales, por lo cual el discernimiento del signo, así como la visualización de la unidad imaginaria, se efectúa entre las líneas N°65 y N°70.

Implementando el pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se tendrá lo siguiente:

```
001: #include <conio.h>
002: #include <stdio.h>
003: #include <math.h>
```

```
004:
005: /* Definicion de variables */
006:
007: /* Reactancias del Sistema electrico (en p.u.) */
008:
009:    float  Za ,      // Nodos 1 y 2
010:           Zb ,      // Nodos 1 y 3
011:           Zc ,      // Nodos 1 y 4
012:           Zd ,      // Nodos 2 y 3
013:           Ze ,      // Nodos 3 y 4
014:           Z1 ,      // Transformador 1
015:           Z2 ,      // Transformador 2
016:           ZG ,      // Generador
017:           ZM   ;    // Motor
018:
019:    /* Matriz de admitancias                          */
020:
021:    float  Y[4][4] ;
022:
023:    /* Contadores                                     */
024:
025:    int    i, j ;
026:
027: void main()
028:  {
029:
030:    /* Lectura de datos del sistema electrico */
031:
032:    clrscr() ;
033:    printf ("** Reactancias del Sistema Electrico (p.u.) **\n") ;
034:    printf ("\n\n") ;
035:    printf ("- Za = ") ; scanf("%f", &Za) ;
036:    printf ("- Zb = ") ; scanf("%f", &Zb) ;
037:    printf ("- Zc = ") ; scanf("%f", &Zc) ;
038:    printf ("- Zd = ") ; scanf("%f", &Zd) ;
039:    printf ("- Ze = ") ; scanf("%f", &Ze) ;
040:    printf ("- Z1 = ") ; scanf("%f", &Z1) ;
041:    printf ("- Z2 = ") ; scanf("%f", &Z2) ;
042:    printf ("- ZG = ") ; scanf("%f", &ZG) ;
043:    printf ("- ZM = ") ; scanf("%f", &ZM) ;
044:
045:    /* Calculo de la matriz de admitancia */
046:
047:    Y[0][0] = 1/Za + 1/Zb + 1/Zc ;
```

```
048:    Y[0][1] = 1/Za ;
049:    Y[0][2] = 1/Zb ;
050:    Y[0][3] = 1/Zc ;
051:    Y[1][0] = 1/Za ;
052:    Y[1][1] = 1/Za + 1/Zd + 1/Ze ;
053:    Y[1][2] = 1/Zd ;
054:    Y[1][3] = 1/Ze ;
055:    Y[2][0] = 1/Zb ;
056:    Y[2][1] = 1/Zd ;
057:    Y[2][2] = 1/Zb + 1/Zd + 1/(Z1+ZG) ;
058:    Y[2][3] = 0 ;
059:    Y[3][0] = 1/Zc ;
060:    Y[3][1] = 1/Ze ;
061:    Y[3][2] = 0 ;
062:    Y[3][3] = 1/Zc + 1/Ze + 1/(Z2+ZM) ;
063:
064:    /* Mostrar resultados */
065:
066:    printf ("\n\n") ;
067:    printf ("**  Matriz de Admitancias  **\n") ;
068:    printf ("\n\n") ;
069:
070:    for (i=0;i<4;i++)
071:     {
072:       for (j=0;j<4;j++)
073:        {
074:          if (i==j)  printf ("  -j ") ;
075:           else  printf ("   j ") ;
076:          printf ("%7.4f  ",Y[i][j]) ;
077:        }
078:       printf ("\n\n") ;
079:     }
080:
081:    printf ("\n\n") ;
082:    printf ("Presione cualquier tecla para salir ...") ;
083:    getch() ;
084:
085:  }
```

Programa 2.2: CALMA.CPP

A diferencia de lo desarrollado a nivel algorítmico, cuando se lleva a cabo la declaración y utilización de los arrays bidimensionales en el lenguaje de programación C se debe tomar en cuenta lo siguiente:

- La forma de declaración de un array bidimensional es la siguiente:

tipo nombre_de_variable[tamaño_fila][tamaño_columna] ; **(2.8)**

- Todos los arrays bidimensionales tienen el [0][0] como su primer elemento. Por tanto, cuando se escribe **`float Y[4][4]`** se está declarando una matriz de números reales que tiene dieciseis elementos, desde **`Y[0][0]`** hasta **`Y[3][3]`**.

Para calcular los valores correspondientes, se hace referencia a cada uno de los elementos del array bidimensional, tal y como se lleva a cabo entre las líneas N°47 y N°62, mediante la sentencia de asignación "=".

Finalmente, se tiene un reporte de resultados, mostrándose en pantalla los valores correspondientes a las reactancias de cada uno de los componentes de la matriz de admitancias, ya que se consideró de antemano que la componente resistiva de las impedancias consideradas en el problema es cero. Para tal finalidad se utiliza la sentencia de control ***for*** de tipo anidado.

El ingreso de datos, así como la salida de resultados, mediante la ejecución del programa CALMA.CPP, se muestra a continuación en la Figura 2.4:

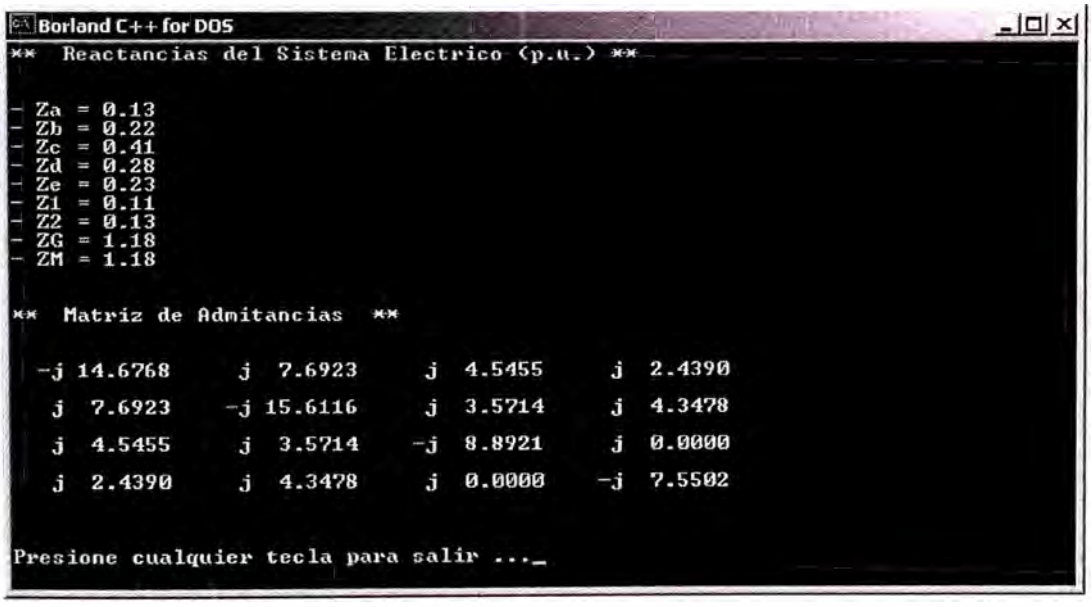


Figura 2.4: Ingreso de datos y reporte de resultados del sistema eléctrico.

Si bien es cierto que el pseudocódigo ***Calculo_matriz_admitancia***, y su correspondiente programa satisfacen los requerimientos del problema, se está utilizando un nivel de abstracción forzado para simular la existencia de números complejos. Para modelar el comportamiento de un número complejo, recurriremos a una estructura de datos denominada *registro*.

2.3 Registros

Un *registro* es una colección de información, normalmente relativa a una entidad particular. Un registro es una colección de campos lógicamente relacionados, que pueden ser tratados como una unidad por algún programa, proporcionando un medio eficaz de mantener conglomerada la información relacionada. Un *campo* es un item o elemento de datos elementales.

La forma mediante la cual se declara un registro es la siguiente:

```
Tipos
  registro : <tipo_registro>
    <nombre_campo> : <tipo_dato>
    ≡
  fin_registro
Variables
  ≡
  <nombre_registro> : <tipo_registro>
  ≡
```

(2.9)

En la proposición de un pseudocódigo se tiene una sección denominada ***Tipos***, en la cual se pueden crear nuevos tipos de datos, conocido como *tipos de datos definidos por el usuario*.

En el caso del registro se efectúa una definición del mismo proporcionando un nombre para el nuevo tipo creado, como en *<tipo_registro>*. Dentro del registro se definen los campos, que no son más que una simple declaración de variables. Se pone término a la definición del registro con la cláusula *fin_registro*, y se le declara como variable dentro de la sección *Variables*.

Las notaciones empleadas para efectuar operaciones de asignación, lectura y escritura, se muestran a continuación en la Tabla 2.2:

Tabla 2.3: Operaciones de asignación, lectura y escritura en registros.

Operación	Notación
Asignación	***nombre_registro.nombre_campo ← valor, variable***
Lectura	***Leer nombre_registro.nombre_campo***
Escritura	***Escribir nombre_registro.nombre_campo***

Para tener acceso a los elementos de un registro, se debe colocar el nombre del registro seguido por un punto y a continuación el nombre del campo.

Ejemplo 2.3: En la Figura 2.5 se muestra el diagrama de un sistema de potencia de tres barras. Los valores (en p.u.) de las impedancias de línea consideradas se muestran en la Tabla 2.4. Almacenar en registros los valores de las impedancias.

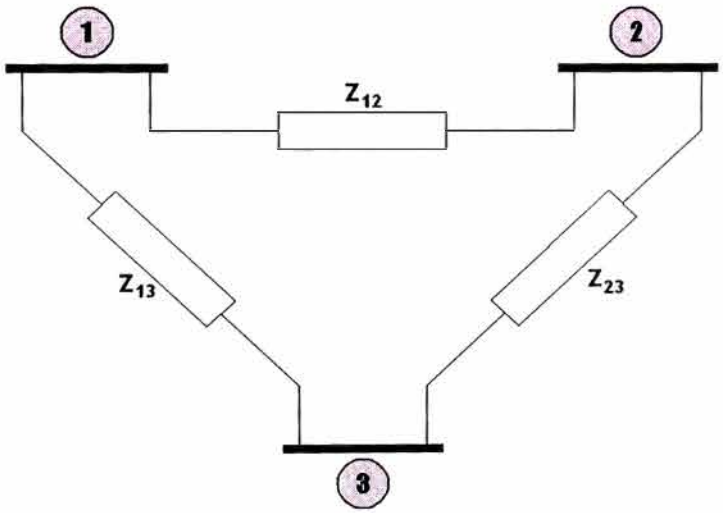


Figura 2.5: Sistema de potencia de tres barras.

Tabla 2.4: Valores de las impedancias de línea.

Barras	Impedancia de Línea (p.u.)
1 – 2	0.02 + j 0.10
2 – 3	0.04 + j 0.15
1 – 3	0.03 + j 0.18

Para almacenar las impedancias de línea en registros, primero debemos modelar la estructura del registro para que contenga dos campos de números reales (digamos x e y). A continuación podemos definir tres variables de tipo registro para almacenar estas tres impedancias, o de lo contrario podemos definir un array unidimensional de registros de tamaño 3, que permita almacenar las impedancias consideradas.

Para efectos del ejemplo, vamos a considerar la definición de un array de registros, tal y como se muestra a continuación:

```
001: Pseudocódigo  Almacena_admitancia_registro
002:
003: Tipos
004:
005:   registro : admitancia
006:     x : real
007:     y : real
008:   fin_registro
009:
010: Variables
011:
012:   { Vector de registros }
013:
014:     z : array [1..3] de admitancia
015:
016:   { Contador }
017:
018:     i : entero
019:
020: Inicio
021:
022:   { Asignación de datos al array unidimensional }
023:
024:     Paso N°01:  z[0].x ← 0.02  // Resistencia de línea 1-2
025:                 z[0].y ← 0.01  // Reactancia de línea 1-2
```

```
026:                z[1].x ← 0.04  // Resistencia de línea 2-3
027:                z[1].y ← 0.15  // Reactancia de línea 2-3
028:                z[2].x ← 0.03  // Resistencia de línea 1-3
029:                z[2].y ← 0.04  // Reactancia de línea 1-3
030:
031:   { Muestra valores del array de registros }
032:
033:     Paso N°02:  Desde i ← 1 hasta 3 hacer
034:
035:                   Escribir z[i].x, " + j ", z[i].y
036:
037:                 Fin_Desde
038:
039: Fin
```

Pseudocódigo 2.3: Almacena_admitancia_registro

En la línea N°1 se establece el nombre del pseudocódigo. Entre las líneas N°3 y N°8 se define el registro *admitancia* que tiene dos campos reales, el componente ***x*** corresponde a la parte real y el componente ***y*** corresponde a la parte imaginaria de la admitancia respectivamente. Entre las líneas N°10 y N°18 se definen las variables a ser utilizadas por el pseudocódigo y que se encuentran convenientemente comentadas. Cabe resaltar que en la línea N°11 se declara la variable ***z*** como array unidimensional de registros de tipo *admitancia* de tamaño (o dimensión) 3.

La asignación de las impedancias de línea a cada uno de los elementos del array de registros se muestra entre las líneas N°24 y N°29. Nótese que la asignación de las componentes real e imaginaria se efectúa de manera individual, de acuerdo a la descripción presentada en la Tabla 2.3.

Finalmente, entre las líneas N°31 y N°37 se efectúa un reporte de las asignaciones efectuadas, mostrando el contenido de los registros del array unidimensional, utilizando para dicho fin un bucle *desde-hasta*.

Implementando el pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se tendrá lo siguiente:

```
001: #include <conio.h>
002: #include <stdio.h>
003:
004: /* Definicion de tipos */
005:
006:    struct admitancia {
007:
008:      double  x ;
009:      double  y ;
010:
011:    } ;
012:
013: /* Definicion de variables */
014:
015:    /* Vector de Registros */
016:
017:       struct admitancia z[3] ;
018:
019:    /* Contadores                                  */
020:
021:       int    i ;
022:
023: void main()
024:  {
025:
026:       clrscr() ;
027:
028:    /* Asignacion de datos al array unidimensional */
029:
030:       z[0].x = 0.02 ;
031:       z[0].y = 0.01 ;
032:       z[1].x = 0.04 ;
033:       z[1].y = 0.15 ;
034:       z[2].x = 0.03 ;
035:       z[2].y = 0.04 ;
036:
037:    /* Muestra valores del array de registros */
038:
039:       printf ("***  Muestra impedancias asignadas  **\n\n") ;
040:
041:       for (i=0;i<3;i++)
042:         printf ("z[%d] = %5.2lf + j %5.2lf\n",i,z[i].x,z[i].y) ;
```

```
043:
044:        printf ("\n\n") ;
045:        printf ("Presione cualquier tecla para salir ...") ;
046:        getch() ;
047:
048:  }
```

Programa 2.3: REGADM.CPP

Entre las líneas N°6 y N°11 se define el registro admitancia mediante la palabra reservada *struct*. Entre las líneas N°13 y N°21 se definen las variables a ser utilizadas por el programa. En la línea N°17 se tiene declarado el array unidimensional de registros tipo admitancia *z* de tamaño 3.

Entre las líneas N°28 y N°35 se tiene la asignación de los valores correspondientes a las impedancias de línea en sus componentes resistiva y reactiva respectivamente. Finalmente, entre las líneas N°37 y N°42, se muestra el reporte de las asignaciones llevadas a cabo mediante un bucle ***for*** mostrando por separado las componentes resistiva y reactiva de cada una de las impedancias de línea.

El reporte de resultados, mediante la ejecución del programa REGADM.CPP, se muestra a continuación en la Figura 2.6:

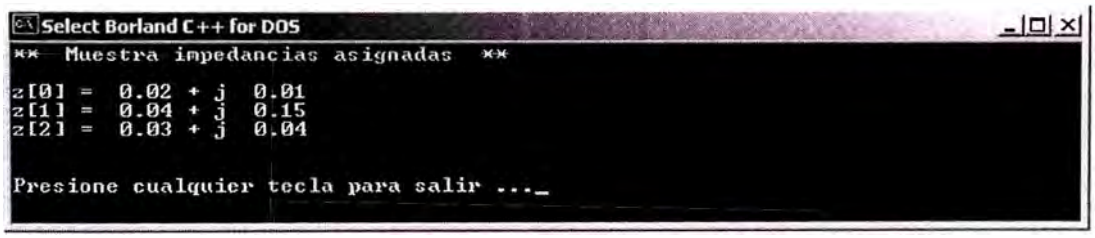


Figura 2.6: Reporte de resultados del array unidimensional *z*.

2.4 Registros y la representación de números complejos

Dentro de la definición de los tipos de datos básicos de un algoritmo, no se contempla la declaración de las variables de tipo complejas. Una solución a ello es definir un registro que almacena los componentes real e imaginario de un número complejo. Sin embargo, en el lenguaje de programación C/C++ - específicamente hablando del Borland C++ - considera dentro de sus librerías la declaración de variables complejas.

Una consecuencia directa del uso específico de las librerías de una versión de un lenguaje de programación en particular es la pérdida de portabilidad en la codificación de un algoritmo. Software de aplicación como el MATLAB, permite la utilización de números complejos de manera directa, lo cual incide en una manipulación más flexible de dichos números.

Extenderemos los tipos de datos básicos de tal manera que ante la declaración de una variable compleja, las operaciones matemáticas efectuadas con números complejos tendrán el mismo conjunto de operadores. Se produce implícitamente una extensión del lenguaje algorítmico, que es la *sobrecarga de operadores*.

Ejemplo 2.4: De acuerdo al enunciado del Ejemplo 2.2 y a lo expresado líneas arriba, desarrollar un pseudocódigo que permita obtener la matriz de admitancias de nodo del sistema de potencia mostrado en la Figura 2.3.

El pseudocódigo que concuerda con los requerimientos del enunciado es el que se muestra a continuación:

```
001: Pseudocódigo  Calculo_matriz_admitancia_modificado
002:
003: Variables
004:
005:   { Tensiones de generador y barra infinita }
006:
007:     Za, Zb, Zc, Zd, Ze, Z1, Z2, ZG, ZM : complejo
008:
009:   { Matriz de admitancias }
010:
011:     Y : array [1..4,1..4] de complejo
012:
013:   { Contador }
014:
015:     i, j : entero
016:
017: Inicio
018:
019:   { Lectura de datos del sistema eléctrico }
020:
021:     Paso N°01:  Escribir "Za = "
022:                 Leer Za
023:                 Escribir "Zb = "
024:                 Leer Zb
```

```
025:                Escribir "Zc = "
026:                Leer Zc
027:                Escribir "Zd = "
028:                Leer Zd
029:                Escribir "Ze = "
030:                Leer Ze
031:                Escribir "Z1 = "
032:                Leer Z1
033:                Escribir "Z2 = "
034:                Leer Z2
035:                Escribir "ZG = "
036:                Leer ZG
037:                Escribir "ZM = "
038:                Leer ZM
039:
040:  { Cálculo de la matriz de admitancia }
041:
042:    Paso N°02:  Y[1][1] ← 1/Za + 1/Zb + 1/Zc
043:                Y[1][2] ← 1/Za
044:                Y[1][3] ← 1/Zb
045:                Y[1][4] ← 1/Zc
046:                Y[2][1] ← 1/Za
047:                Y[2][2] ← 1/Za + 1/Zd + 1/Ze
048:                Y[2][3] ← 1/Zd
049:                Y[2][4] ← 1/Ze
050:                Y[3][1] ← 1/Zb
051:                Y[3][2] ← 1/Zd
052:                Y[3][3] ← 1/Zb + 1/Zd + 1/(Z1+ZG)
053:                Y[3][4] ← 0
054:                Y[4][1] ← 1/Zc
055:                Y[4][2] ← 1/Ze
056:                Y[4][3] ← 0
057:                Y[4][4] ← 1/Zc + 1/Ze + 1/(Z2+ZM)
058:
059:  { Mostrar Resultados }
060:
061:    Paso N°03:  Desde  i ← 1  hasta 4 hacer
062:
063:                  Paso N°04:  Desde j ← 1  hasta 4 hacer
064:
065:                                Paso N°05:  Escribir Y[i][j]
066:
```

```
067:                                    Fin_Desde
068:
069:                        Fin_Desde
070:
071: Fin
```

Pseudocódigo 2.4: Calculo_matriz_modificado

En la línea N°1 se establece el nombre del pseudocódigo. Entre las líneas N°5 y N°15 se definen las variables a ser utilizadas por el pseudocódigo y que se encuentran convenientemente comentadas. Cabe resaltar que se están declarando variables con el tipo de dato *complejo*. La entrada de los datos provenientes del sistema eléctrico se encuentra especificada entre las líneas N°19 y N°38. El cálculo de los componentes de la matriz de admitancias se lleva a cabo de acuerdo a la expresión **(2.7)** y se muestra entre las líneas N°42 y N°57. Finalmente, entre las líneas N°59 y N°69 se efectúa un reporte de resultados, mostrando la matriz de admitancia obtenida, mediante un bucle *desde-hasta* de tipo anidado, dado que se efectúa un "recorrido" de filas y columnas para mostrar cada uno de los valores correspondientes de la matriz de orden 4.

Implementando el pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se tendrá lo siguiente:

```
001: #include <iostream.h>
002: #include <conio.h>
003: #include <stdio.h>
004: #include <math.h>
005: #include <complex.h>
006:
007: /* Definicion de variables */
008:
009:    /* Reactancias del Sistema electrico (en p.u.) */
010:
011:       complex  Za ,     // Nodos 1 y 2
012:                Zb ,     // Nodos 1 y 3
013:                Zc ,     // Nodos 1 y 4
014:                Zd ,     // Nodos 2 y 3
015:                Ze ,     // Nodos 3 y 4
```

```
016:              Z1 ,     // Transformador T1
017:              Z2 ,     // Transformador T2
018:              ZG ,     // Generador
019:              ZM  ;    // Motor
020:
021:    /* Matriz de admitancias                    */
022:
023:       complex  Y[4][4] ;
024:
025:    /* Contadores                               */
026:
027:       int      i, j ;
028:
029: void main()
030:  {
031:
032:    /* Lectura de datos del sistema electrico */
033:
034:       clrscr() ;
035:       printf ("**  Reactancias del Sistema Electrico (p.u.) **\n") ;
036:       printf ("\n\n") ;
037:       printf ("- Za = ") ; cin >> Za ;
038:       printf ("- Zb = ") ; cin >> Zb ;
039:       printf ("- Zc = ") ; cin >> Zc ;
040:       printf ("- Zd = ") ; cin >> Zd ;
041:       printf ("- Ze = ") ; cin >> Ze ;
042:       printf ("- Z1 = ") ; cin >> Z1 ;
043:       printf ("- Z2 = ") ; cin >> Z2 ;
044:       printf ("- ZG = ") ; cin >> ZG ;
045:       printf ("- ZM = ") ; cin >> ZM ;
046:
047:    /* Calculo de la matriz de admitancia */
048:
049:       Y[0][0] = 1/Za + 1/Zb + 1/Zc ;
050:       Y[0][1] = 1/Za ;
051:       Y[0][2] = 1/Zb ;
052:       Y[0][3] = 1/Zc ;
053:       Y[1][0] = 1/Za ;
054:       Y[1][1] = 1/Za + 1/Zd + 1/Ze ;
055:       Y[1][2] = 1/Zd ;
056:       Y[1][3] = 1/Ze ;
057:       Y[2][0] = 1/Zb ;
058:       Y[2][1] = 1/Zd ;
059:       Y[2][2] = 1/Zb + 1/Zd + 1/(Z1+ZG) ;
```

```
060:      Y[2][3] = 0 ;
061:      Y[3][0] = 1/Zc ;
062:      Y[3][1] = 1/Ze ;
063:      Y[3][2] = 0 ;
064:      Y[3][3] = 1/Zc + 1/Ze + 1/(Z2+ZM) ;
065:
066:   /* Mostrar resultados */
067:
068:      printf ("\n\n") ;
069:      printf ("** Matriz de Admitancias **\n") ;
070:      printf ("\n\n") ;
071:
072:      for (i=0;i<4;i++)
073:       {
074:         for (j=0;j<4;j++)
075:          printf ("%7.3lf + j %7.3lf ",real(Y[i][j]),imag(Y[i][j]));
076:         printf ("\n\n") ;
077:       }
078:
079:      printf ("\n\n") ;
080:      printf ("Presione cualquier tecla para salir ...") ;
081:      getch() ;
082:
083:  }
```

Programa 2.4: CALMAC.CPP

En este código se tiene la inclusión de dos librerías: *complex.h* y *iostream.h*. La primera se utiliza en la declaración de variables complejas así como la sobrecarga de operadores soportados por la librería *math.h*. La segunda es utilizada para invocar a la sentencia *cin*, que permite el ingreso por consola de números complejos.

Entre las líneas N°7 y N°27 se tienen las declaraciones de variables a ser utilizadas en el programa. Cabe resaltar que tanto las variables que representan a las impedancias como a la matriz de admitancias del sistema eléctrico han sido declaradas de tipo *complex*.

Entre las líneas N°32 y N°45 se tiene el ingreso de datos complejos. El formato para ingresar números complejos mediante la sentencia *cin* tiene la forma ***(parte_real,parte_imaginaria)***, de tal manera que la impedancia **`j0.125`** se ingresa al programa como **`(0,0.125)`**. Este conjunto de sentencias es más completo que el llevado

a cabo entre las líneas N°47 y N°62 del Programa 2.2, ya que en este último solamente se podían ingresar números reales.

Entre las líneas N°49 y N°64 se lleva a cabo el cálculo de las componentes de la matriz de admitancia, que es similar al de las líneas N°47 al N°62 del Programa 2.2. Aquí podemos apreciar a nivel de código que existe una *sobrecarga de operadores*.

Finalmente, se tiene un reporte de resultados, mostrándose en pantalla los valores de los elementos de la matriz de admitancias (tanto el componente resisitivo como el reactivo). Para tal finalidad se utiliza la sentencia de control ***for*** de tipo anidado.

El ingreso de datos, así como el reporte de resultados mediante la ejecución del programa CALMAC.CPP, se muestra a continuación en la Figura 2.7:

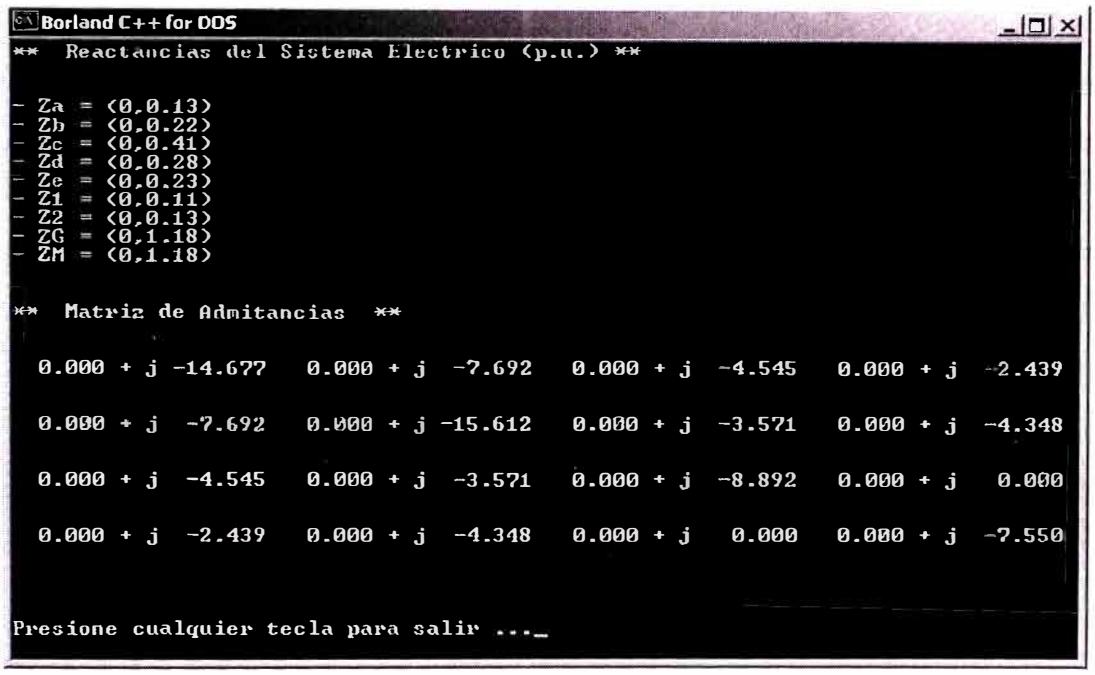


Figura 2.7: Ingreso de datos y reporte de resultados para el programa CALMAC.CPP.

2.5 Cadenas de caracteres

Tal y como se citó en la sección 1.2.3 del Capítulo II: ***Nociones Básicas de Algoritmos***, una *cadena* (*string*) de caracteres es un conjunto de caracteres – incluido el espacio en blanco – que se almacena en un área contigua de memoria. La cadena que no contiene caracteres se denomina *cadena vacía* o *nula*, no se debe confundir con una cadena solamente compuesta por espacios en blanco.

Una *variable de cadena* o *tipo carácter* es una variable cuyo valor es una cadena de caracteres, las cuales se deben declarar dentro del algoritmo, y que presentan el siguiente formato:

Variables

≡

<nombre_cadena> : cadena **(2.10)**

≡

Esencialmente, un *string* es un array unidimensional de caracteres, por lo cual se puede hacer referencia a cada uno de los caracteres de la cadena por su posición en la misma. Por ejemplo si se tiene una variable tipo cadena – por ejemplo **`texto`** - con un valor *"Este es un ejemplo"*, el valor de **`texto[1]`** es *'E'*, el valor de **`texto[7]`** es *'s'* y así sucesivamente.

Al asignar un valor a una variable cadena, se encierra entre comillas (más de un carácter) mientras que para la declaración de caracteres se utilizan apóstrofes.

De acuerdo a la declaración de la longitud, las variables de cadena se dividen de acuerdo a lo especificado en la Tabla 2.5:

Tabla 2.5: Tipos de variable cadena.

Variable cadena	Descripción
Estática	Su longitud se define antes de ejecutar el programa y no puede cambiarse a lo largo de éste.
Semiestática	Su longitud puede variar durante la ejecución del programa, sin sobrepasar un límite máximo especificado al principio.
Dinámica	Su longitud puede variar sin limitación dentro del programa.

Las notaciones empleadas para efectuar operaciones de asignación, lectura y escritura de cadenas, se muestran a continuación en la Tabla 2.6:

Tabla 2.6: Operaciones de asignación, lectura y escritura en registros.

Operación	Notación
Asignación	***nombre_variable_cadena ← valor, variable***
Lectura	***Leer(nombre_variable_cadena)***
Escritura	***Escribir(nombre_variable_cadena)***

Ejemplo 2.5: Considerando las condiciones del Ejemplo 2.2, modificar el pseudocódigo de tal manera que se permita almacenar en forma de texto la referencia de cada una de las impedancias de línea, adicionalmente a los valores de las impedancias propiamente dichas.

Para el presente ejemplo, definiremos un registro que contenga como primer campo una variable tipo cadena que tenga la referencia de la impedancia de línea (BARRA 1-2, BARRA 2-3, BARRA 1-3) y cuyo segundo campo sea una variable de tipo compleja. Definiremos en la sección de variables un array unidimensional de tipo registro de tamaño 3. A continuación asignaremos los valores correspondientes y finalmente procederemos a mostrar el reporte de resultados.

El pseudocódigo que concuerda con los requerimientos del enunciado es el que se muestra a continuación:

```
001: Pseudocódigo  Almacena_admitancia_registro
002:
003: Tipos
004:
005:   registro : admitancia
006:     texto : cadena
007:     z     : complejo
008:   fin_registro
009:
```

```
010: Variables
011:
012:   { Vector de registros }
013:
014:     Red : array [1..3] de admitancia
015:
016:   { Contador }
017:
018:     i : entero
019:
020: Inicio
021:
022:   { Asignación de datos al array unidimensional }
023:
024:     Paso N°01:  Red[1].texto ← "BARRA 1-2"
025:                 Red[1].z ← 0.02 + j 0.10
026:                 Red[2].texto ← "BARRA 2-3"
027:                 Red[2].z ← 0.04 + j 0.15
028:                 Red[3].texto ← "BARRA 2-3"
029:                 Red[3].z ← 0.03 + j 0.18
030:
031:   { Muestra valores del array de registros }
032:
033:     Paso N°02:  Desde i ← 1 hasta 3 hacer
034:
035:                  Escribir Red[i].texto, " : ", Red[i].z
036:
037:                Fin_Desde
038:
039: Fin
```

Pseudocódigo 2.5: Almacena_admitancia_registro

En la línea N°1 se establece el nombre del pseudocódigo. Entre las líneas N°3 y N°8 se define un tipo de dato registro denominado *admitancia*, que tiene como primer campo una variable tipo cadena que contendrá la información relacionada con la línea considerada, y el segundo campo es un número complejo que contedrá la impedancia de línea.

Entre las líneas N°10 y N°18 se definen las variables a ser utilizadas a lo largo del pseudocódigo. Nótese que en la línea N°18 se define el array unidimensional de tipo *admitancia*, denominado *Red* (de tamaño 3).

Entre las líneas N°22 y N°29 se efectúa una asignación de valores a cada uno de los elementos del array unidimensional declarado anteriormente. Finalmente, se efectúa un reporte de los datos asignados anteriormente, utilizando para ello la sentencia de control *desde-hasta*.

Implementando el pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se tendrá lo siguiente:

```
001: #include <conio.h>
002: #include <stdio.h>
003: #include <string.h>
004: #include <complex.h>
005:
006: /* Definicion de tipos */
007:
008:    struct admitancia {
009:      char     texto[15] ;
010:      complex  z ;
011:    } ;
012:
013: /* Definicion de variables */
014:
015:    /* Vector de Registros */
016:
017:       struct admitancia Red[3] ;
018:
019:    /* Contadores                                  */
020:
021:       int    i ;
022:
023: void main()
024:  {
025:
026:       clrscr() ;
027:
028:    /* Asignacion de datos al array unidimensional */
029:
```

```
030:       strcpy(Red[0].texto,"BARRA 1-2") ;
031:       Red[0].z = complex(0.02,0.10) ;
032:       strcpy(Red[1].texto,"BARRA 2-3") ;
033:       Red[1].z = complex(0.04,0.15) ;
034:       strcpy(Red[2].texto,"BARRA 1-3") ;
035:       Red[2].z = complex(0.03,0.18) ;
036:
037:    /* Muestra valores del array de registros */
038:
039:       printf ("** Muestra impedancias asignadas **\n\n") ;
040:
041:       for (i=0;i<3;i++)
042:         printf ("%s : %5.2lf + j %5.2lf\n",Red[i].texto, …
                    real(Red[i].z),imag(Red[i].z)) ;
043:
044:       printf ("\n\n") ;
045:       printf ("Presione cualquier tecla para salir ...") ;
046:       getch() ;
047:
048:  }
```

Programa 2.5: REGADM_C.CPP

En este código se tiene la inclusión de la librería *string.h*, la cual permite la utilización de funciones a ser utilizadas para el tratamiento de cadenas de caracteres. Entre las líneas N°6 y N°11 se define el tipo de dato *admitancia*, el cual es un registro compuesto por la variable *texto* la cual ha sido limitada a una longitud máxima de 15 caracteres, y a continuación se define la variable compleja *z*.

Entre las líneas N°13 y N°21 se definen tanto el array *Red* de tamaño 3, como el contador que permitirá hacer referencia a cada uno de los elementos del array en cuestión. Entre las líneas N°28 y N°35 se tienen las asignaciones de valores a las respectivas variables.

Nótese que la asignación de una cadena de caracteres al campo de texto de cada uno de los elementos del array se lleva a cabo mediante la función *strcpy* perteneciente a la librería *string.h*, mientras que la asignación de números complejos se efectúa declarando en primer lugar el tipo de dato *complex* seguido del valor que se desea asignar.

Finalmente se obtiene el reporte de resultados, mostrando para ello cada uno de los campos componentes del array de registros, mediante el bucle *for*.

El reporte de resultados mediante la ejecución del programa REGADM_C.CPP, se muestra a continuación en la Figura 2.8:

Figura 2.8: Reporte de resultados del array unidimensional *Red*.

2.6 Archivos

Un *archivo* es un conjunto de datos estructurados en una colección de entidades elementales básicas denominadas *registros*, que son de igual tipo y constan a su vez de diferentes entidades de nivel más bajo denominadas *campos*.

Los tipos de *soporte* (medio físico donde se almacenan los datos) utilizados en la gestión de archivos se muestran en la Tabla 2.7:

Tabla 2.7: Soportes secuenciales y direccionables.

Tipo de soporte	Descripción
Soporte Secuencial	Los registros están escritos uno a continuación de otro. Para accesar a un registro *n* se requiere pasar por los primeros *n-1*.
Soporte direccionable	Las informaciones registradas se pueden localizar por su dirección y no se requiere pasar por los registros precedentes. Los registros deben poseer un campo clave que los diferencie del resto de registros del archivo.

Los tipos de acceso a los registros de un archivo, se muestran en la Tabla 2.8:

Tabla 2.8: Tipos de acceso en archivos.

Tipos de acceso	Descripción
Acceso Secuencial	Implica el acceso a un archivo según el orden de almacenamiento de sus registros, uno tras otro.
Acceso directo	Implica el acceso a un registro determinado, sin que ello implique la consulta de los registros precedentes. Este tipo de acceso sólo es posible con soportes direccionables.

La *organización* de un archivo define la forma en la que los registros se disponen sobre el soporte de almacenamiento. En la Tabla 2.9 se definen los tipos de organización de un archivo:

Tabla 2.9: Tipos de organización en archivos.

Tipos de acceso	Descripción
Organización secuencial	Es una sucesión de registros almacenados consecutivamente sobre el soporte externo. Para acceder a un registro *n* dado es obligatorio pasar por todos los *n-1* elementos que le preceden.
Organización directa o aleatoria	El orden físico no se corresponde con el orden lógico. Los datos se sitúan en el archivo y se accesa a ellos mediante su posición (lugar relativo que ocupan). Se pueden leer y escribir registros en cualquier orden y posición.
Organización secuencial indexada	El tipo de sus registros tiene un campo clave identificador, los registros están situados en un soporte direccionable y existe un índice por cada una de las posiciones direccionables.

En el presente trabajo, utilizaremos los *archivos de texto*, los cuales son un caso particular de archivos de organización secuencial. El archivo de texto es una serie continua de caracteres que se pueden leer uno tras otro, donde cada registro del archivo de texto es una cadena de caracteres.

A continuación se muestra la definición de una variable tipo archivo:

Variables

<nombre variable> : archivo **(2.10)**

Las operaciones básicas a considerar en el tratamiento de archivos tipo texto son las que se muestran a continuación:

Tabla 2.10: Operaciones básicas a efectuar con archivos.

Operación	Notación
Creación	***Crear(<variable_archivo>,<nombre_fisico>)***
Apertura	***Abrir(<variable_archivo>,<nombre_fisico>)***
Cierre	***Cerrar(<variable_archivo>)***
Lectura	***Leer(<variable_archivo>,<lista de variables>)***
Escritura	***Escribir(<variable_archivo>,<lista de variables>)***

Ejemplo 2.5: En la Figura 2.9 se muestra el diagrama unifilar de un sistema de potencia de 5 barras. Los datos de línea para el sistema mostrado se muestran en la Tabla 2.11. Desarrollar un pseudocódigo que permita obtener la matriz de admitancias del sistema considerado.

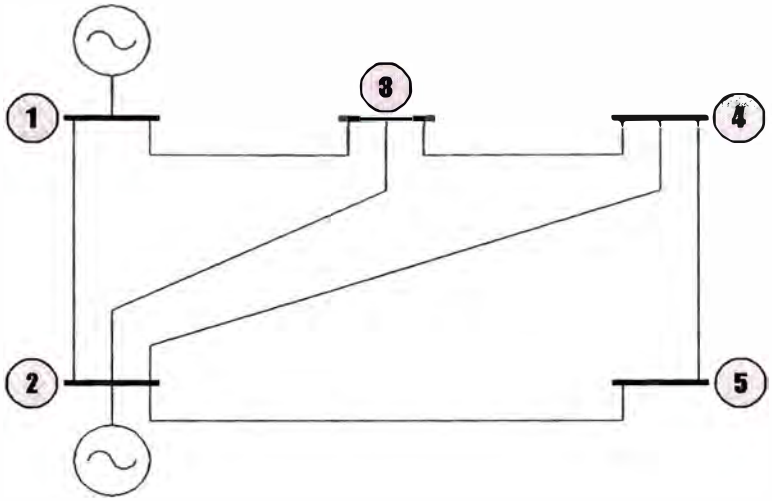


Figura 2.9: Sistema de potencia de cinco barras.

Tabla 2.11: Datos de línea para el sistema de cinco barras.

Barra de Envío	Barra de Recepción	Resistencia (p.u.)	Reactancia (p.u.)	$Y_s/2$
1	2	0.02	0.06	0.030
1	3	0.08	0.24	0.025
2	3	0.06	0.18	0.020
2	4	0.06	0.18	0.020
2	5	0.04	0.12	0.015
3	4	0.01	0.03	0.010
4	5	0.08	0.24	0.025

Para obtener la matriz de admitancia se debe considerar la siguiente expresión:

$$Y_{ij} = \begin{cases} \textbf{\textit{Elementos diagonales:}} & \textit{Suma de todas las admitancias incidentes (conectadas) al nodo cuando } i=j \\ \textbf{\textit{Elementos no diagonales:}} & \textit{Admitancia equivalente conectada entre los nodos } i \textit{ y } j\textit{, con signo cambiado } \forall i \neq j \end{cases} \quad \textbf{(2.11)}$$

Dado que tenemos un sistema de 5 barras, se hace tedioso el ingreso de datos, por lo cual necesitaremos que los datos mostrados en la Tabla 2.11 se encuentren en un archivo

de texto, de tal manera que éste pueda ser leído y procesado por el programa. El archivo de datos para el sistema de 5 barras, denominado ***BARRA5.DAT*** presenta el formato que se presenta en la Figura 2.10:

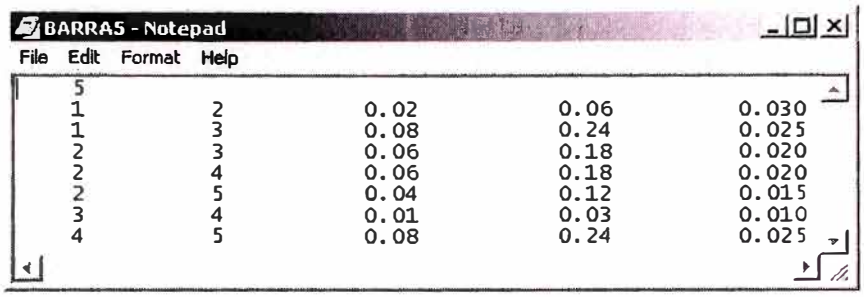


Figura 2.10: Archivo BARRA5.DAT

En la primera línea se tiene el número de barras del sistema eléctrico considerado. Las siguientes líneas especifican la barra de envío, barra de recepción, resistencia, reactancia y admitancia shunt, tal y como se mostró en la Tabla 2.11 líneas arriba.

El pseudocódigo que se derrollará consta de tres partes, las cuales se definen a continuación en la Tabla 2.12:

Tabla 2.12: Descripción de pseudocódigos

Algoritmo	Descripción
Calcula_matriz_admitancia_archivo	Es el pseudocódigo principal que permitirá procesar el archivo de entrada para obtener la matriz de admitancia y efectuará llamadas a las subrutinas *Inicializa_matriz* y *Genera_reporte*.
Inicializa_matriz	Subrutina que permite inicializar a cero los elementos de la matriz de admitancia, utiliza para ello la variable global ***Y***.
Genera_reporte	Subrutina que permite generar el archivo de salida, cuyo contenido es la matriz de admitancias generada por el algoritmo principal. Utiliza como datos de entrada el orden de la matriz y el nombre del archivo de salida.

El pseudocódigo principal ***Calcula_matriz_admitancia_archivo*** es el encargado de efectuar llamadas a las subrutinas consideradas en la Tabla 2.12. La interacción de las subrutinas con el algoritmo principal se representa en la Figura 2.11

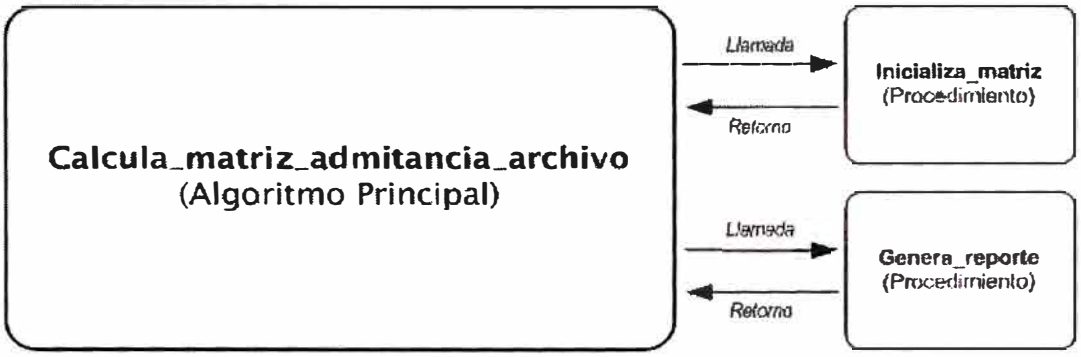


Figura 2.11: Llamadas y retornos entre el algoritmo principal y sus subrutinas.

El desarrollo del pseudocódigo ***Calcula_matriz_admitancia_archivo*** se muestra a continuación:

```
001: Pseudocódigo Calcula_matriz_admitancia_archivo
002:
003: Constantes
004:
005:   MAX = 10
006:
007: Variables
008:
009:   { Archivo de texto }
010:
011:     arch    : archivo    //Manipula archivos de entrada y salida
012:     a_ent ,              //Nombre de archivo de entrada
013:     a_sal   : cadena     //Nombre de archivo de salida
014:
015:   { Datos de línea }
016:
017:     i  ,                 //Barra de envío
018:     j       : entero     //Barra de recepción
019:     r  ,                 //Resistencia de línea
020:     x  ,                 //Reactancia de línea
021:     ys      : real       //Admitancia shunt de línea (Yc/2)
```

```
022:
023:   { Matriz de admitancias }
024:
025:      Y : Array [1..MAX,1..MAX] de complejo   //Matriz de admitancias
026:      n : entero            //Matriz de orden n (Orden máximo = MAX)
027:
028: Inicio
029:
030:   { Identificación de archivos }
031:
032:   Paso N°01: Escribir "Archivo de entrada : "
033:              Leer a_ent
034:              Escribir "Archivo de salida : "
035:              Leer a_sal
036:
037:   { Procesamiento del archivo de entrada }
038:
039:   Paso N°02: Si arch_e no existe
040:               Entonces
041:                 Escribir "Archivo no encontrado …"
042:               Si_No
043:
044:                 Paso N°03: Escribir "Procesando archivo …"
045:
046:                 Paso N°04: Llamar_A  Inicializa_matriz
047:
048:                 Paso N°05: Leer(arch,n)  //Orden de la matriz
049:
050:                 Paso N°06: Mientras ~(EOF(arch)) hacer
051:
052:                                 Paso N°07: Leer (arch,i,j,r,x,ys)
053:
054:                                 Paso N°09: Si (i=j)
055:                                              Entonces
056:                                                Y[i][j] ← Y[i][j]+1/(r+jx)+jys
057:                                              Si_No
058:                                                Y[i][i] ← Y[i][i]+1/(r+jx)+jys
059:                                                Y[j][j] ← Y[j][j]+1/(r+jx)+jys
060:                                                Y[i][i] ← - 1/(r+jx)
061:                                                Y[i][i] ← - 1/(r+jx)
062:                                            Fin_Si
063:
064:                               Fin_Mientras
065:
```

```
066:                  Paso N°10: Llamar_A Genera_reporte(n,a_sal)
067:
068:              Fin_Si
069:
070:    Paso N°11: Cerrar(arch)
071:
072: Fin
```

Pseudocódigo 2.6: Calcula_matriz_admitancia_archivo

En la línea N°1 se define el nombre del pseudocódigo. A continuación en la línea N°5 se define la constante *MAX*, la cual define el orden máximo de la matriz de admitancia que se podrá dimensionar. En caso se requiera tener un orden mayor en la matriz de admitancia (que depende del número de barras del sistema) se podrá modificar el valor asignado a esta constante.

Entre las líneas N°7 y N°26 se definen las variables a ser utilizadas, dentro de las cuales se tienen las variables de manipulación de archivos (líneas N°11 al N°13), los datos de línea (líneas N°17 al N°21) que van a ser proporcionados por el archivo de entrada. En la línea N°24 se define la matriz de admitancia, de orden 10 (definido por la constante *MAX*), en la línea N°25 se define la variable *n* a la que se asignará el número de barras del sistema eléctrico considerado (y que en consecuencia definirá el orden de la matriz de admitancias).

Entre las líneas N°32 y N°35 se definen los nombres de los archivos de entrada (que contiene los datos a procesar) y de salida (que contiene la matriz de admitancias). Entre las líneas N°39 y N°67 se define el desarrollo principal del algoritmo.

En primer término se verifica que el nombre del archivo de entrada exista (línea N°39). Si el archivo de entrada existe, entonces se procede a inicializar la matriz de admitancia (línea N°46) mediante la llamada al procedimiento ***Inicializa_matriz***. A continuación se asigna a la variable *n* el número de barras contemplado en el archivo ***BARRA5.DAT*** (línea N°48) y mientras no se llegue al fin de archivo, se leen los datos correspondientes a la barra de envío, la barra de recepción, resistencia, reactancia y admitancia shunt de línea (línea N°52) así como la aplicación de la expresión **(2.11)** para la formación de la matriz de admitancia (líneas N°54 al N°62).

Una vez terminado el recorrido del archivo (así como el cálculo simultáneo de la matriz de admitancia) se procede a crear el archivo de salida mediante la llamada al procedimiento ***Genera_reporte*** (línea N°66) que tiene como parámetros de entrada el orden de la matriz, así como el nombre del archivo de salida. Finalmente, en la línea N°69 se procede a cerrar el archivo de entrada.

En la línea N°46 del algoritmo principal se efectúa una llamada al pseudocódigo ***Inicializa_matriz***, el cual se muestra a continuación:

```
01: Procedimiento  Ingresa_datos
02:
03: Variables
04:
05:   k, m : entero
06:
07: Inicio
08:
09:   Paso N°01:  Desde k ← 1 hasta MAX hacer
10:
11:                Paso N°02:  Desde m ← 1 hasta MAX hacer
12:
13:                              Y[k][m] ← 0 + j0
14:
15:                            Fin_Desde
16:
17:               Fin_Desde
18:
19: Fin_Procedimiento
```

Pseudocódigo 2.7: Procedimiento *Inicializa_matriz*

En este procedimiento se asigna cero a los elementos de la matriz de admitancia, dado que la declaración de la variable *Y* (que dentro del contexto del procedimiento es una variable global) no garantiza un valor inicial de sus elementos en cero. Utiliza para dicha finalidad una sentencia de control anidada *desde-hasta*. Para efectuar el recorrido de la matriz se utilizan las variables locales *k* y *m*.

En la línea N°65 del algoritmo principal se efectúa una llamada al pseudocódigo ***Genera_reporte***, el cual se muestra a continuación:

```
01: Procedimiento Genera_reporte (E n:entero,E nombre:cadena)
02:
03: Variables
04:
05:   arch   : archivo
06:   k, m   : entero
07:
08: Inicio
09:
10:   Paso N°01:  Crear(arch,nombre)
11:
12:   Paso N°02:  Escribir(arch,"Matriz de Admitancias")
13:
14:   Paso N°03:  Desde k ← 1 hasta n hacer
15:
16:                Paso N°04:  Desde m ← 1 hasta n hacer
17:
18:                              Escribir (arch,Y[k][m])
19:
20:                            Fin_Desde
21:
22:               Fin_Desde
23:
24:   Paso N°05:  Cerrar(arch,nombre)
25:
26: Fin_Procedimiento
```

Pseudocódigo 2.8: Procedimiento *Genera_reporte*

En la línea N°1 se indica que este procedimiento tiene como parámetros de entrada (definidos por una **E**) a las variables *n* y *nombre*, las cuales se constituyen como variables locales de la función y cuyos valores han sido proporcionados por las variables globales *n* y *n_sal* respectivamente. En la línea N°10 se crea el archivo de salida y entre las líneas N°12 y N°22 se registran los datos correspondientes a la matriz de admitancia, para finalizar con la grabación del archivo mediante una operación de cierre (línea N°24).

Implementando el pseudocódigo utilizando el lenguaje de programación C (utilizando un compilador Borland C++ 3.1), se tendrá lo siguiente:

```
001: #include <conio.h>
002: #include <stdio.h>
003: #include <string.h>
004: #include <complex.h>
005:
006: /* Definicion de constantes */
007:
008:    const int MAX = 10 ;
009:
010: /* Definicion de variables  */
011:
012:   /* Archivo de texto */
013:
014:       FILE     *archivo  ;    //Manipula archivos de entrada y salida
015:       char     a_ent[30] ;    //Nombre de archivo de entrada
016:       char     a_sal[30] ;    //Nombre de archivo de salida
017:
018:   /* Datos de linea */
019:
020:       int      i ,            //Barra de envío
021:                j    ;         //Barra de recepcion
022:       double   r ,            //Resistencia de linea
023:                x ,            //Reactancia de linea
024:                ys   ;         //Admitancia shunt de linea (Yc/2)
025:
026:   /* Matriz de admitancias */
027:
028:       complex  Y[MAX][MAX] ; //Matriz de admitancias
029:       int      n ;            //Dimension de Y (Tamaño maximo = MAX)
030:
031: /* Definicion de procedimientos y funciones */
032:
033:    void Inicializa_matriz() ;
034:    void Genera_reporte(int,char *) ;
035:
036: void main()
037:  {
038:
039:   /* Identificacion del archivos                    */
040:
041:       clrscr() ;
042:       printf ("*** Calculo de la Matriz de Admitancias ***\n\n") ;
043:       printf ("* Archivo de entrada : ") ;
```

```
044:      scanf ("%s", &a_ent) ;
045:      printf ("* Archivo de salida : ") ;
046:      scanf ("%s", &a_sal) ;
047:      printf ("\n\n") ;
048:
049:   /* Procesamiento del archivo de entrada           */
050:
051:      if ((archivo=fopen(a_ent,"rt"))==NULL)
052:       printf ("Archivo no encontrado ... \n\n") ;
053:      else
054:       {
055:        printf ("Procesando archivo %s ...\n\n",a_ent) ;
056:
057:        Inicializa_matriz() ;
058:
059:        fscanf(archivo,"%d",&n) ;
060:        while(!feof(archivo))
061:         {
062:           fscanf(archivo,"%d%d%lf%lf%lf",&i,&j,&r,&x,&ys) ;
063:           if (i==j)
064:            Y[i-1][i-1] = Y[i-1][i-1]+1/complex(r,x)+complex(0,ys);
065:           else
066:            {
067:             Y[i-1][i-1] = Y[i-1][i-1]+1/complex(r,x)+complex(0,ys);
068:             Y[j-1][j-1] = Y[j-1][j-1]+1/complex(r,x)+complex(0,ys);
069:             Y[i-1][j-1] = Y[j-1][i-1] = - 1/complex(r,x) ;
070:            }
071:         }
072:
073:        Genera_reporte(n,a_sal) ;
074:       }
075:
076:      fclose(archivo) ;
077:
078:   /* Salida del programa                           */
079:
080:      printf ("\n\n") ;
081:      printf ("Presione cualquier tecla para salir ...") ;
082:      getch() ;
083:
084:  }
085:
086: /**/void Inicializa_matriz()
087:      {
```

```
088:        int  k, m ;
089:
090:        for (k=0;k<MAX;k++)
091:          for (m=0;m<MAX;m++)  Y[k][m] = complex(0,0) ;
092:      }
093:
094: /**/void Genera_reporte(int n, char *nombre)
095:      {
096:        FILE  *arch ;
097:        int   k, m  ;
098:
099:        arch = fopen(nombre,"wt") ;
100:        fprintf(arch,"*** Matriz de Admitancias ***\n\n") ;
101:        for (k=0;k<n;k++)
102:        {
103:          for (m=0;m<n;m++)
104:          fprintf (arch,"%7.3lf+j%7.3lf  ",real(Y[k][m]), …
                        imag(Y[k][m])) ;
105:          fprintf (arch,"\n") ;
106:        }
107:        fclose(arch) ;
108:      }
```

Programa 2.6: CALMA_A.CPP

Para definir variables tipo archivo se utiliza la palabra reservada ***FILE*** (ver líneas N°14 y N°97). Cuando se desea abrir un archivo, se utiliza la sentencia ***fopen***, utilizando para ello el siguiente formato:

$$\textit{<variable\_archivo> = fopen(<nombre\_archivo>,tipo\_operación)} \qquad \textbf{(2.12)}$$

donde ***tipo_operacion*** contiene el modo de apertura deseado. Los diferentes modos de apertura se definen a continuación:

Tabla 2.13: Modos de apertura de un archivo.

Modo de apertura	Descripción
"r"	Abre un archivo de texto para lectura
"w"	Crea un archivo de texto para escritura
"a"	Abre un archivo de texto para añadir
"rb"	Abre un archivo binario para lectura
"wb"	Abre un archivo binario para escritura
"ab"	Abre un archivo binario para añadir
"r+"	Abre un archivo de texto para lectura/escritura
"w+"	Crea un archivo de texto para lectura/escritura
"a+"	Abre o Crea un archivo de texto para lectura/escritura
"r+b"	Abre un archivo binario para lectura/escritura
"w+b"	Crea un archivo binario para lectura/escritura
"a+b"	Abre o crea un archivo binario para lectura/escritura
"rt"	Abre un archivo de texto para lectura
"wt"	Crea un archivo de texto para escritura
"at"	Abre un archivo de texto para añadir
"r+t"	Abre un archivo de texto para lectura/escritura
"w+t"	Crea un archivo de texto para lectura/escritura
"a+t"	Abre o crea un archivo de texto para lectura/escritura

En la línea N°60 se tiene la función ***foef*** que indica si se ha llegado al final del archivo secuencial. Para leer los valores correspondientes a los datos de línea se utiliza la función ***fscanf***, similar a la función ***scanf*** con la diferencia que se invoca en primer término a la variable archivo.

Para generar el reporte de la matriz de admitancias, se utiliza la funcion ***fprintf***, similar a la función ***printf*** con la diferencia que se invoca en primer término a la variable archivo.

La ejecución del programa CALMA_A.CPP, se muestra a continuación en la Figura 2.8:

Figura 2.12: Ejecución del programa CALMA_A.CPP.

La matriz de admitancia obtenida se almacena en el archivo BARRA5.ADM, tal y como se muestra en la Figura 2.13:

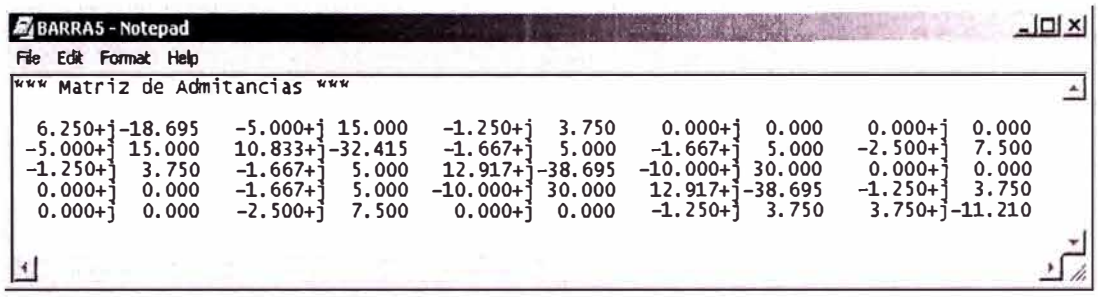


Figura 2.13: Contenido del archivo BARRA5.ADM.

Ejemplo 2.6: En la Figura 2.14 se muestra el diagrama unifilar de un sistema de potencia de 6 barras. Los datos de línea para el sistema mostrado se muestran en la Tabla 2.14. Utilizando el programa ***CALMA_A.CPP***, obtener la matriz de admitancia.

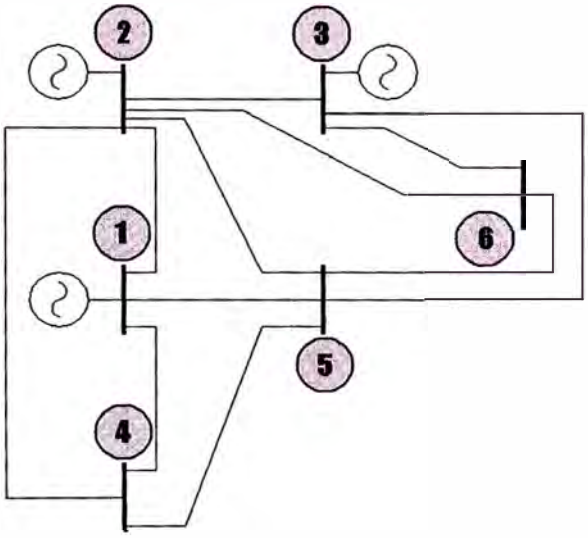


Figura 2.14: Sistema de potencia de seis barras.

Tabla 2.14: Datos de línea para el sistema de seis barras.

Barra de Envío	Barra de Recepción	Resistencia (p.u.)	Reactancia (p.u.)	$Y_s/2$
1	2	0.10	0.20	0.020
1	4	0.05	0.20	0.020
1	5	0.08	0.30	0.030
2	3	0.05	0.25	0.030
2	4	0.05	0.10	0.010
2	5	0.10	0.30	0.020
2	6	0.07	0.20	0.025
3	5	0.12	0.26	0.025
3	6	0.02	0.10	0.010
4	5	0.20	0.40	0.040
5	6	0.10	0.30	0.030

El archivo de datos para el sistema de 6 barras, denominado ***BARRA6.DAT*** presenta el formato que se presenta en la Figura 2.15:

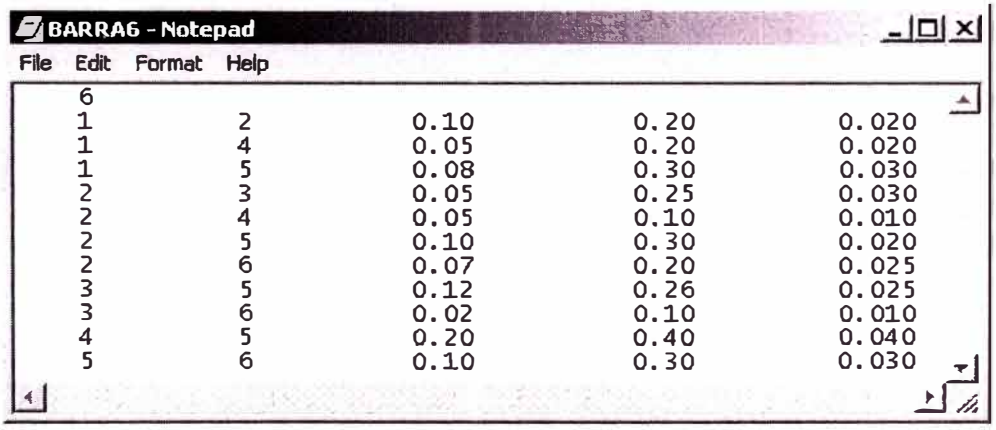


Figura 2.15 Archivo BARRA6.DAT

La ejecución del programa CALMA_A.CPP, se muestra a continuación en la Figura 2.16:

Figura 2.16: Ejecución del programa CALMA_A.CPP.

La matriz de admitancia obtenida se almacena en el archivo BARRA6.ADM, tal y como se muestra en la Figura 2.17:

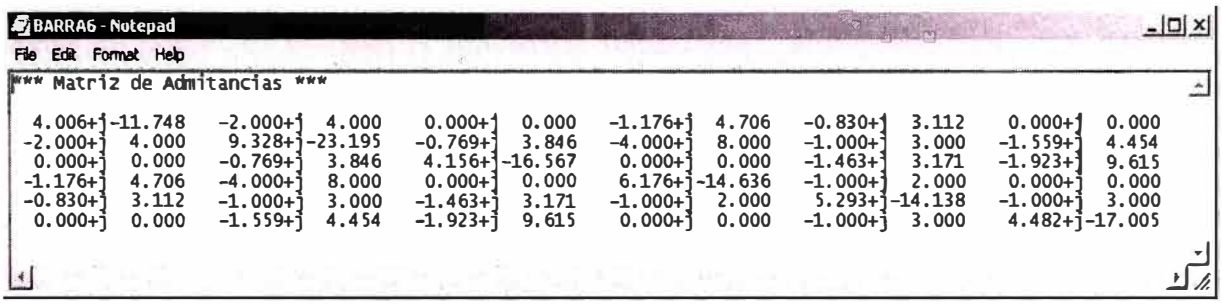


Figura 2.17: Contenido del archivo BARRA6.ADM.

Para efectos del presente trabajo, el programa soporta una matriz de admitancia hasta de orden 10. En caso se desee utulizar el programa para un sistema de potencia con una cantidad de barras mayor de 10, modificar la línea N°8 para cambiar la dimensión máxima, y respetar el formato de los archivos, tal y como se utiliza en los ejemplos citados anteriormente.

CAPÍTULO III
ALMACENAMIENTO DINÁMICO DE MATRICES DISPERSAS

3.1 Introducción

En el Capítulo II se desarrolló en detalle la obtención de la matriz de admitancias de un sistema de potencia utilizando estructuras de datos estáticas (arrays bidimensionales). Si bien es cierto que se tiene una velocidad de ejecución aceptable, también es cierto que para sistemas de potencias grandes la limitación fundamental al absorber una gran cantidad de datos será la memoria.

A fin de optimizar el uso de la memoria en los programas desarrollados en ingeniería eléctrica, se propone el uso de una estrucutura dinámica de datos (listas enlazadas) para almacenar aquellos componentes no nulos de una matriz de admitancias.

3.2 Matrices dispersas

Los sistemas de potencia de gran escala tienen un pequeño número de líneas de transmisión conectadas a cada subestación. Solamente unos pocos elementos de la matriz de admitancias son diferentes de cero, aquellos correspondientes a las líneas de transmisión o conexiones a transformadores. Bajo este punto de vista se dice que estamos tratando con matrices dispersas.

Desde el punto de vista de la programación, por ejemplo, velocidad computacional, precisión y almacenamiento de datos es deseable llevar a cabo cálculos solamente sobre aquellas entradas diferentes de cero sobre la matriz de admitancia.

Existen diversos métodos relacionados con las técnicas de dispersión, entre las que se encuentran:

A. Tabla de entrada simple

B. Tabla de entrada doble

C. Listas LIFO (Last In First Out – Primero en entrar, último en salir)

Este primer conjunto de técnicas están relacionadas con estructuras de datos estáticas (arrays unidimensionales y bidimensionales), los cuales requieren la separación de un

espacio fijo de memoria antes de producirse la ejecución de un programa que administre este tipo de estructuras.

Aparte de esta clasificación se tiene un método de almacenamiento estático que está ligado a la solución de sistemas dispersos por el método de sobre relajación sucesiva (SOR – Successive Over Relaxation) y que utiliza tres arrays unidimensionales (de tamaño fijo) para el almacenamiento lineal de una matriz bidimensional.

Existen esquemas de almacenamiento dinámico de matrices dispersas, dentro de los cuales podemos citar los siguientes:

A. Método de listas enlazadas.

B. Método de árboles binarios.

C. Método del array de punteros.

3.2.1 Método de listas enlazadas

Cuando se implementa una matriz dispersa con una lista enlazada, se utiliza una estructura para almacenar la información de cada elemento de la matriz, incluyendo su posición lógica y enlaces al elemento previo y al siguiente elemento. Cada estructura se coloca en la lista con el elemenento en un orden basado en los índices de la matriz. Su acceso se produce siguiendo los enlaces.

3.2.2 Método de árboles binarios

En esencia, un árbol binario es simplemente una lista doblemente enlazada modificada. Su gran ventaja sobre una lista es que se puede buscar rápidamente, lo que significa que las inserciones y las consultas pueden ser muy rápidas. En aplicaciones en las que se quiere una estructura de lista enlazada pero se necesitan tiempos de búsqueda cortos, el árbol binario es lo ideal.

3.2.3 Método del array de punteros

Este método hace que el acceso a los elementos de la matriz dispersa sea casi tan rápido como acceder a los elementos de un array bidimensional normal. A comparación de los dos métodos anteriores, este método requiere solamente un puntero, mientras que los anteriores requieren dos punteros para efectuar el acceso correspondiente.

3.3 Obtención de la matriz de admitancias mediante estructuras de datos dinámicas.

Teniendo como base el esquema de abstracción exigido para efectuar modelamiento algorítmico (y su posterior implementación en un lenguaje de programación) mediante la

explicación extensiva llevada a cabo en los Capítulos I y II, se procederá a desarrollar una aplicación que permita la obtención de la matriz de admitancia mediante un esquema de almacenamiento dinámico.

Para cumplir con dicho objetivo, se hace necesario conocer los fundamentos teóricos relacionados con punteros, así como el tratamiento de listas enlazadas. Estos dos temas están considerados en el Apéndice D.

Para dicho fin se ha implementado un programa denominado ***CALMA_LM.CPP*** que tiene como base la inclusión de una librería de gestión de listas enlazadas, denominada ***LIBLISTA.H*** que es incluído dentro dentro del código principal.

El programa ***CALMA_LM.CPP*** funciona de la siguiente manera:

Paso N°1: Se proporciona el nombre del archivo de entrada (con extensión DAT).

Paso N°2: Se proporciona el nombre del archivo de salida (con extensión ADM), el cual contendrá la matriz de admitancias (los elementos no nulos de la matriz dispersa).

Paso N°3: Se efectuará una comprobación de la existencia del archivo de entrada. En caso de no existir, el programa se dará por terminado.

Paso N°4: Una vez efectuada la comprobación, se procede a leer el archivo línea por línea, identificando la barra de envío, la barra de recepción, la resistencia de línea (en p.u.), la reactancia de línea (en p.u.), así como la impedancia shunt (Ys/2).

Paso N°5: Se efectúa una comprobación de la existencia del par (barra de envío,barra de recepción) en la lista enlazada. Si no existe entonces se produce la inserción del elemento en la lista para el caso de una componente que se encuentre fuera de la diagonal. En caso de existir, se efectúa una acumulación o suma de admitancias sucesivas para para el caso de las componentes diagonales.

Paso N°6: Se prosigue con los Pasos N°4 y N°5 hasta que se llegue al término del archivo de datos.

Paso N°7: Una vez culminada la asignación de elementos, se procede a la creación del archivo de salida, cuyo nombre se proporcionó en el Paso N°2.

Como ejemplo, en la siguiente sección veremos la aplicación a un sistema de 23 barras.

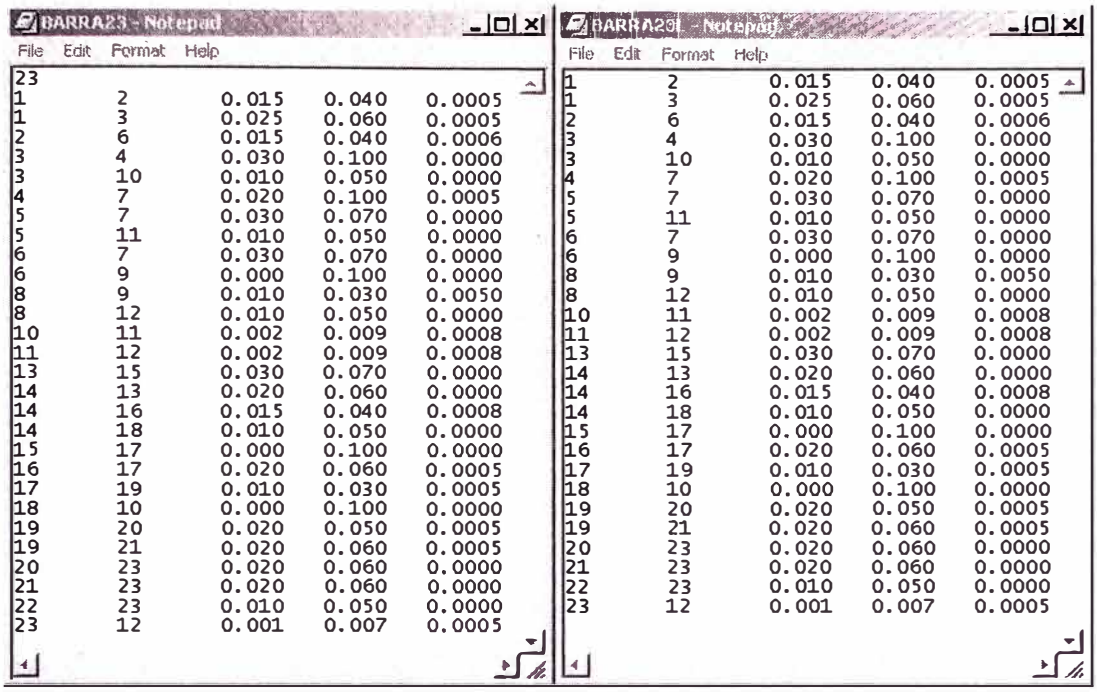
3.4 Ejemplo de aplicación y comparación a un sistema de 23 barras

Se tiene en el archivo ***BARRA23.DAT*** y ***BARRA23L.DAT*** los datos correspondientes a un sistema de potencia de 23 barras, de acuerdo a la tabla de datos que se muestra a continuación:

Tabla 3.1: Archivos BARRA23.DAT y BARRA23L.DAT

Barra de Envío	Barra de Recepción	Resistencia (p.u.)	Reactancia (p.u.)	Ys (p.u.)
1	2	0.015	0.040	0.0005
1	3	0.025	0.060	0.0005
2	6	0.015	0.040	0.0006
3	4	0.030	0.100	0.0000
3	10	0.010	0.050	0.0000
4	7	0.020	0.100	0.0005
5	7	0.030	0.070	0.0000
5	11	0.010	0.050	0.0000
6	7	0.030	0.070	0.0000
6	9	0.000	0.100	0.0000
8	9	0.010	0.030	0.0050
8	12	0.010	0.050	0.0000
10	11	0.002	0.009	0.0008
11	12	0.002	0.009	0.0008
13	15	0.030	0.070	0.0000
14	13	0.020	0.060	0.0000
14	16	0.015	0.040	0.0008
14	18	0.010	0.050	0.0000
15	17	0.000	0.100	0.0000
16	17	0.020	0.060	0.0005
17	19	0.010	0.030	0.0005
18	10	0.000	0.100	0.0000
19	20	0.020	0.050	0.0005
19	21	0.020	0.060	0.0005
20	23	0.020	0.060	0.0000
21	23	0.020	0.060	0.0000
22	23	0.010	0.050	0.0000
23	12	0.001	0.007	0.0005

La figura correspondiente a los archivos ***BARRA23.DAT*** y ***BARRA23L.DAT*** se muestra a continuación:



BARRA23 - Notepad

File Edit Format Help

```
23
1        2        0.015    0.040    0.0005
1        3        0.025    0.060    0.0005
2        6        0.015    0.040    0.0006
3        4        0.030    0.100    0.0000
3        10       0.010    0.050    0.0000
4        7        0.020    0.100    0.0005
5        7        0.030    0.070    0.0000
5        11       0.010    0.050    0.0000
6        7        0.030    0.070    0.0000
6        9        0.000    0.100    0.0000
8        9        0.010    0.030    0.0050
8        12       0.010    0.050    0.0000
10       11       0.002    0.009    0.0008
11       12       0.002    0.009    0.0008
13       15       0.030    0.070    0.0000
14       13       0.020    0.060    0.0000
14       16       0.015    0.040    0.0008
14       18       0.010    0.050    0.0000
15       17       0.000    0.100    0.0000
16       17       0.020    0.060    0.0005
17       19       0.010    0.030    0.0005
18       10       0.000    0.100    0.0000
19       20       0.020    0.050    0.0005
19       21       0.020    0.060    0.0005
20       23       0.020    0.060    0.0000
21       23       0.020    0.060    0.0000
22       23       0.010    0.050    0.0000
23       12       0.001    0.007    0.0005
```

BARRA23L - Notepad

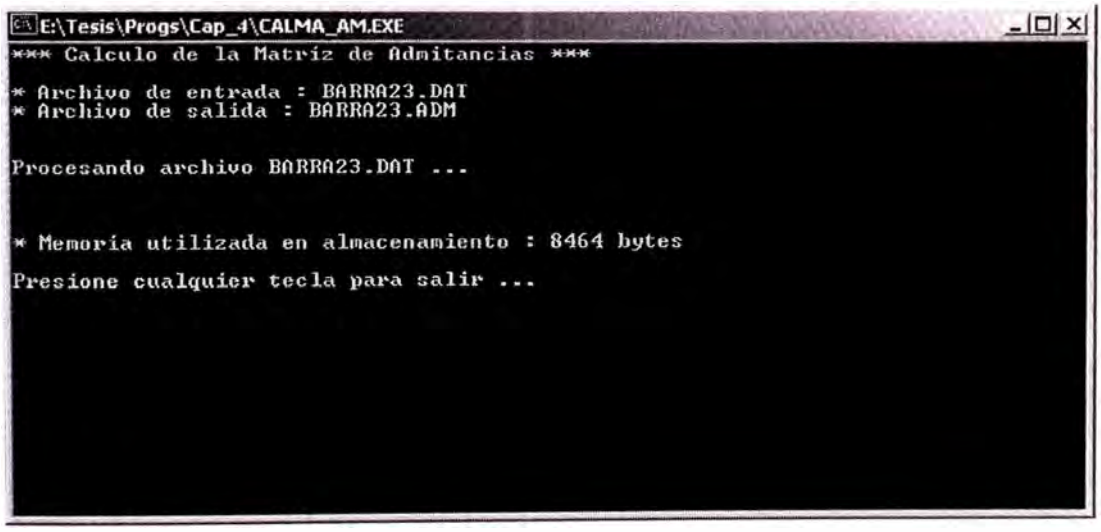
File Edit Format Help

```
1        2        0.015    0.040    0.0005
1        3        0.025    0.060    0.0005
2        6        0.015    0.040    0.0006
3        4        0.030    0.100    0.0000
3        10       0.010    0.050    0.0000
4        7        0.020    0.100    0.0005
5        7        0.030    0.070    0.0000
5        11       0.010    0.050    0.0000
6        7        0.030    0.070    0.0000
6        9        0.000    0.100    0.0000
8        9        0.010    0.030    0.0050
8        12       0.010    0.050    0.0000
10       11       0.002    0.009    0.0008
11       12       0.002    0.009    0.0008
13       15       0.030    0.070    0.0000
14       13       0.020    0.060    0.0000
14       16       0.015    0.040    0.0008
14       18       0.010    0.050    0.0000
15       17       0.000    0.100    0.0000
16       17       0.020    0.060    0.0005
17       19       0.010    0.030    0.0005
18       10       0.000    0.100    0.0000
19       20       0.020    0.050    0.0005
19       21       0.020    0.060    0.0005
20       23       0.020    0.060    0.0000
21       23       0.020    0.060    0.0000
22       23       0.010    0.050    0.0000
23       12       0.001    0.007    0.0005
```

Figura 3.1: Archivos BARRA23.DAT y BARRA23L.DAT

La diferencia entre estos dos archivos radica en que mientras que en ***BARRA23.DAT*** se tiene que especificar el número de barras, en el archivo ***BARRA23L.DAT*** prescinde de este factor, ya que la asignación de las impedancias de línea se leen directamente y se asignan dinámicamente para formar la matriz de admitancias (no se requiere una cantidad fija de memoria).

Al ejecutar el programa ***CALMA_AM.CPP*** sobre el archivo ***BARRA23.DAT*** se tiene el siguiente reporte:



E:\Tesis\Progs\Cap_4\CALMA_AM.EXE

```
*** Calculo de la Matriz de Admitancias ***

* Archivo de entrada : BARRA23.DAT
* Archivo de salida : BARRA23.ADM

Procesando archivo BARRA23.DAT ...


* Memoria utilizada en almacenamiento : 8464 bytes
Presione cualquier tecla para salir ...
```

Figura 3.2: Ejecución del programa CALMA_AM.CPP

El archivo de salida ***BARRA23.ADM***, que contiene la matriz de admitancias, se muestra parcialmente a continuación:

```
BARRA23 - Notepad
File  Edit  Format  Help
*** Matriz de Admitancias ***

 14.136+j-36.118    -8.219+j 21.918    -5.917+j 14.201     0.000+j  0.000     0.000+j  0.00
 -8.219+j 21.918    16.438+j-43.835     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
 -5.917+j 14.201     0.000+j  0.000    12.516+j-42.606    -2.752+j  9.174     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000    -2.752+j  9.174     4.675+j-18.789     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     9.019+j-31.30
  0.000+j  0.000    -8.219+j 21.918     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000    -1.923+j  9.615    -5.172+j 12.06
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000    -3.846+j 19.231     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000    -3.846+j 19.23:
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
  0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.000     0.000+j  0.00
```

Figura 3.3: Reporte del archivo BARRA23.ADM

Como se puede apreciar, se tiene una matriz de orden 23 en la cual se tienen una gran cantidad de elementos iguales a cero.

Al ejecutar el programa ***CALMA_LM.CPP*** sobre el archivo ***BARRA23L.DAT*** se tiene el siguiente reporte:

```
E:\Tesis\Progs\Cap_4\CALMA_LM.EXE
*** Calculo de la Matriz de Admitancias ***

* Archivo de entrada : BARRA23L.DAT
* Archivo de salida : BARRA23L.ADM

Procesando archivo BARRA23L.DAT ...


* Memoria utilizada en almacenamiento : 1738 bytes

Presione cualquier tecla para salir ...
```

Figura 3.4: Ejecución del programa CALMA_LM.CPP

El archivo de salida ***BARRA23L.ADM***, que contiene la matriz de admitancias, se muestra parcialmente a continuación:

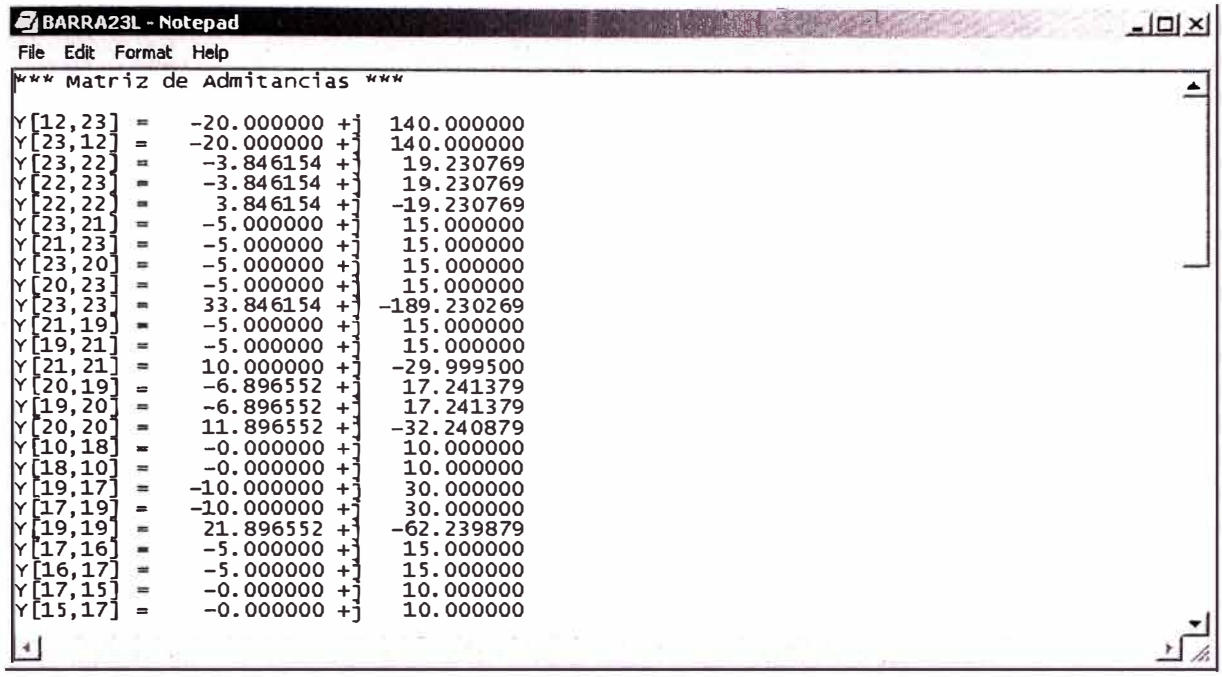


Figura 3.5: Reporte del archivo BARRA23L.ADM

Como se puede apreciar, solamente se tiene un reporte de los elementos de la matriz de admitancia diferentes de cero.

Como se puede apreciar en las figuras 3.2 y 3.4 se tiene en las líneas finales un reporte de la memoria utilizada para el almacenamiento de las matrices de admitancias respectivas.

Mientras que el programa CALMA_AM utiliza un total de 8464 bytes para el almacenamiento, el programa CALMA_LM utiliza un total de 1738 bytes para el almacenamiento única y exclusivamente de los elementos diferentes de cero.

El código del programa ***CALMA_AM.CPP*** se muestra en el Apéndice A, mientras que la librería de listas enlazadas ***LIBLISTA.H*** y el código fuente del programa ***CALMA_LM.CPP*** se muestra en el Apéndice B.

3.5 Aplicación del almacenamiento dinámico para sistemas de 5 y 6 barras

Tomando el enunciado del Ejemplo 2.5 (ver Capítulo II), utilizaremos la aplicación CALMA_LM para obtener los reportes de la matriz de admitancia.

Se tiene el archivo ***BARRA5L.DAT*** que se muestra a continuación:

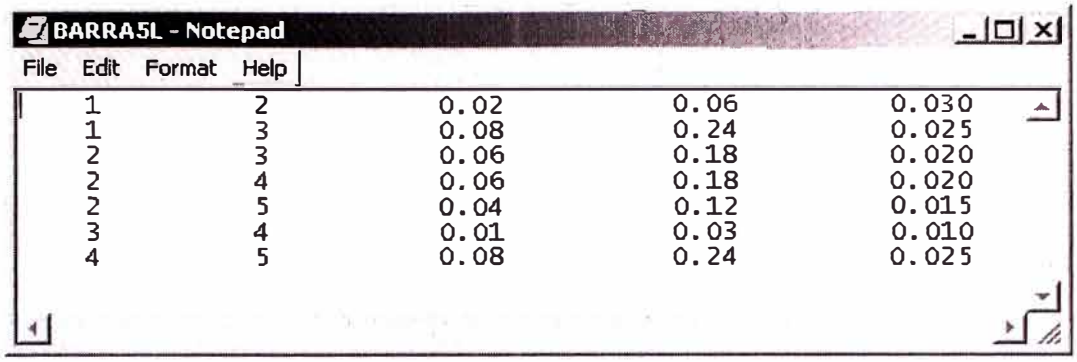


Figura 3.6: Reporte del archivo BARRA5L.DAT

Al ejecutar el programa ***CALMA_LM.CPP*** sobre el archivo ***BARRA5L.DAT*** se tiene el siguiente reporte:

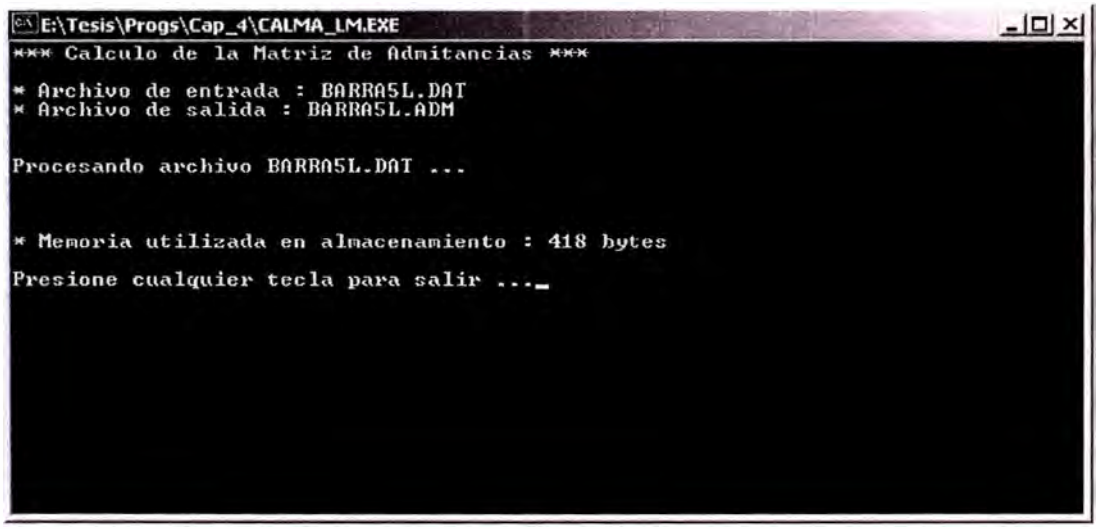


Figura 3.7: Ejecución del programa CALMA_LM.CPP

El archivo de salida ***BARRA5L.ADM***, que contiene la matriz de admitancias, se muestra parcialmente a continuación:

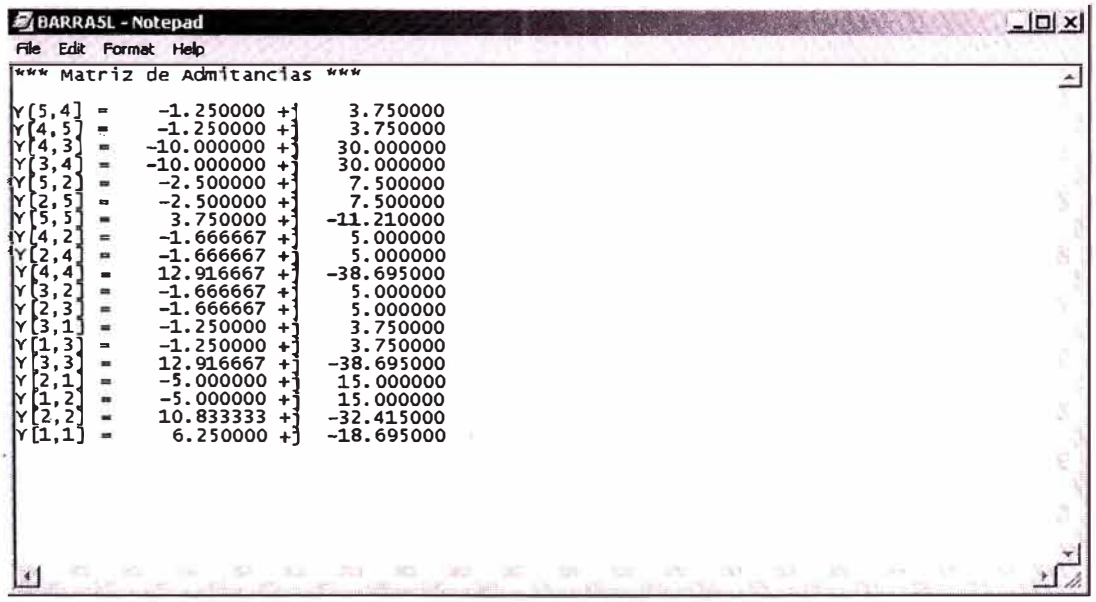


Figura 3.8: Reporte del archivo BARRA5L.ADM

Como se puede apreciar, solamente se tiene un reporte de los elementos de la matriz de admitancia diferentes de cero. Cabe resaltar que el programa CALMA_LM utiliza un total de 418 bytes para el almacenamiento única y exclusivamente de los elementos diferentes de cero.

Tomando el enunciado del Ejemplo 2.6 (ver Capítulo II), utilizaremos la aplicación CALMA_LM para obtener los reportes de la matriz de admitancia.

Se tiene el archivo ***BARRA6L.DAT*** que se muestra a continuación:

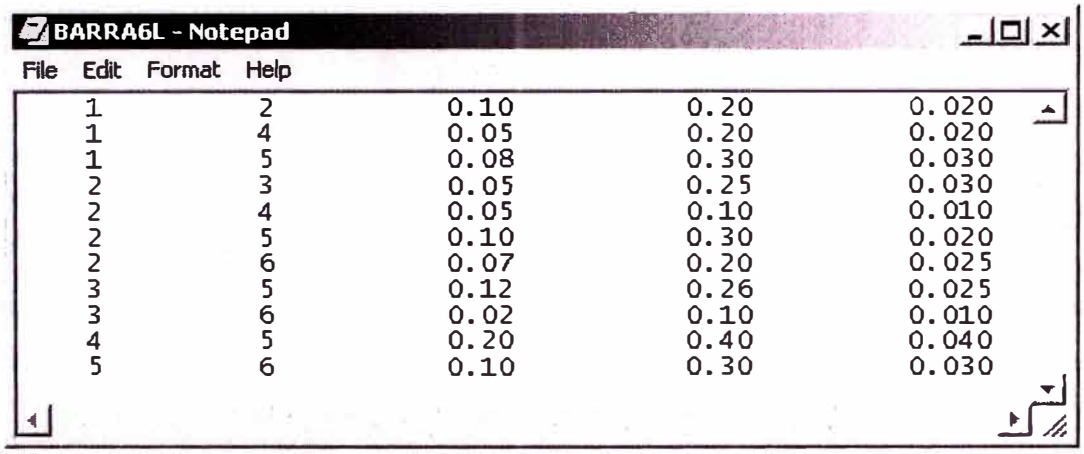


Figura 3.9: Reporte del archivo BARRA6L.DAT

Al ejecutar el programa CALMA_LM.CPP sobre el archivo ***BARRA6L.DAT*** se tiene el siguiente reporte:

Figura 3.10: Ejecución del programa CALMA_LM.CPP

El archivo de salida ***BARRA6L.ADM***, que contiene la matriz de admitancias, se muestra parcialmente a continuación:

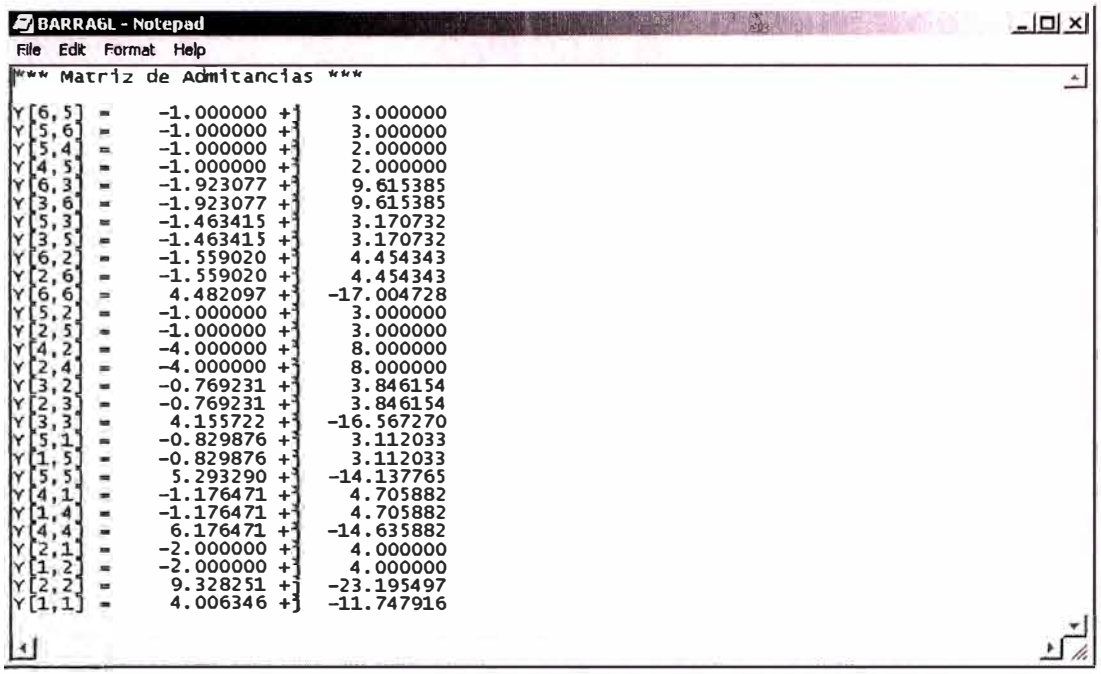


Figura 3.11: Reporte del archivo BARRA6L.ADM

Como se puede apreciar, solamente se tiene un reporte de los elementos de la matriz de admitancia diferentes de cero.

Cabe resaltar que el programa CALMA_LM utiliza un total de 616 bytes para el almacenamiento única y exclusivamente de los elementos diferentes de cero.

CAPÍTULO IV
INVERSIÓN Y FACTORIZACIÓN DE MATRICES DISPERSAS

4.1 Introducción

Para ilustrar la utilización del almacenamiento dinámico de matrices dispersas, se describirán a continuación dos aplicaciones básicas: inversión de una matriz dispersa mediante el método de Shipley-Coleman y la factorización de una matriz dispersa mediante el método LU-Dolittle (LUDM).

4.2 Inversión de una matriz dispersa

Para llevar a cabo la inversión de una matriz dispersa, emplearemos el método de ***Shipley-Coleman***, la cual es llevada a cabo utilizando un proceso de inversión "in-situ", parecido a método de Gauss-Jordan. Este método es utilizado para invertir la matriz de impedancia acoplada primitiva.

4.2.1 Derivación matemática

A partir del sistema $Ax=b$, se desearía manipular la matriz A para llegar a la forma $A^{-1}b=x$, donde A es una matriz $n \times n$, y tanto b como x son vectores de tamaño n.

Consideremos el sistema de tamaño 3×3:

$$A_{11}x_1 + A_{12}x_2 + A_{13}x_3 = b_1 \qquad \textbf{(4.1)}$$

$$A_{21}x_1 + A_{22}x_2 + A_{23}x_3 = b_2 \qquad \textbf{(4.2)}$$

$$A_{31}x_1 + A_{32}x_2 + A_{33}x_3 = b_3 \qquad \textbf{(4.3)}$$

En la primera iteración, a partir de la ecuación **(4.1)** transferimos x_1 hacia el lado derecho:

$$\frac{1}{A_{11}}b_1 - \frac{A_{12}}{A_{11}}x_2 - \frac{A_{13}}{A_{11}}x_3 = x_1 \qquad \textbf{(4.4)}$$

Sustituyendo la ecuación **(4.4)** en la ecuación **(4.2)**, obtenemos lo siguiente:

$$A_{21}\left(\frac{1}{A_{11}}b_1-\frac{A_{12}}{A_{11}}x_2-\frac{A_{13}}{A_{11}}x_3\right)+A_{22}x_2+A_{23}x_3=b_2$$

Factorizando, tenemos a continuación:

$$\frac{A_{21}}{A_{11}}b_1+\left(A_{22}-\frac{A_{21}A_{12}}{A_{11}}\right)x_2+\left(A_{23}-\frac{A_{21}A_{13}}{A_{11}}\right)x_3=b_2 \qquad \textbf{(4.5)}$$

Sustituyendo la ecuación **(4.4)** en la ecuación **(4.3)**, obtenemos lo siguiente:

$$A_{31}\left(\frac{1}{A_{11}}b_1-\frac{A_{12}}{A_{11}}x_2-\frac{A_{13}}{A_{11}}x_3\right)+A_{32}x_2+A_{33}x_3=b_3$$

Factorizando, tenemos a continuación:

$$\frac{A_{31}}{A_{11}}b_1+\left(A_{32}-\frac{A_{31}A_{12}}{A_{11}}\right)x_2+\left(A_{33}-\frac{A_{31}A_{13}}{A_{11}}\right)x_3=b_2 \qquad \textbf{(4.6)}$$

Reescribiendo las ecuaciones **(4.4)**, **(4.5)** y **(4.6)** en la forma matricial, tenemos:

$$\begin{bmatrix} \frac{1}{A_{11}} & \frac{A_{12}}{A_{11}} & \frac{A_{13}}{A_{11}} \\ \frac{A_{21}}{A_{11}} & A_{22}-\frac{A_{21}A_{12}}{A_{11}} & A_{23}-\frac{A_{21}A_{13}}{A_{11}} \\ \frac{A_{31}}{A_{11}} & A_{32}-\frac{A_{31}A_{12}}{A_{11}} & A_{33}-\frac{A_{31}A_{13}}{A_{11}} \end{bmatrix}\begin{bmatrix} b_1 \\ x_2 \\ x_3 \end{bmatrix}=\begin{bmatrix} x_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Al multiplicar la columna 1 por -1 y b_1 por -1, no se altera la matriz:

$$\begin{bmatrix} -\frac{1}{A_{11}} & -\frac{A_{12}}{A_{11}} & \frac{A_{13}}{A_{11}} \\ \frac{A_{21}}{A_{11}} & A_{22}-\frac{A_{21}A_{12}}{A_{11}} & A_{22}-\frac{A_{21}A_{13}}{A_{11}} \\ -\frac{A_{31}}{A_{11}} & A_{32}-\frac{A_{31}A_{12}}{A_{11}} & A_{33}-\frac{A_{31}A_{13}}{A_{11}} \end{bmatrix}\begin{bmatrix} -b_1 \\ x_2 \\ x_3 \end{bmatrix}=\begin{bmatrix} x_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Una vez efectuados los cálculos necesarios, cada elemento en la matriz *A* podría cambiar tal y como se indicó en la matriz líneas arriba, puede ser escrito nuevamente como:

$$A_{11}(-b_1)+A_{12}x_2+A_{13}x_3=x_1 \qquad \textbf{(4.7)}$$

$$A_{21}(-b_1)+A_{22}x_2+A_{23}x_3=b_2 \qquad \textbf{(4.8)}$$

$$A_{31}(-b_1)+A_{32}x_2+A_{33}x_3=b_3 \qquad \textbf{(4.9)}$$

donde los elementos A_{ij} son diferentes que los de las ecuaciones **(4.1)**, **(4.2)** y **(4.3)**. En la segunda iteración, a partir de la ecuación **(4.8)** resolvemos para x_2:

$$-\frac{A_{21}(-b_1)}{A_{22}}+\frac{1}{A_{22}}b_2-\frac{A_{23}}{A_{22}}x_3=x_2 \qquad \textbf{(4.10)}$$

Sustituyendo **(4.10)** en **(4.7)**:

$$A_{11}(-b_1)+A_{12}\left(-\frac{A_{21}(-b_1)}{A_{22}}+\frac{b_2}{A_{22}}-\frac{A_{23}}{A_{22}}x_3\right)+A_{13}x_3=x_1$$

Factorizando términos, tenemos lo siguiente:

$$\left(A_{11}-\frac{A_{12}A_{21}}{A_{22}}\right)(-b_1)+\frac{A_{12}}{A_{22}}b_2+\left(A_{13}-\frac{A_{12}A_{23}}{A_{22}}\right)x_3=x_1$$

Sustituyendo la ecuación **(4.10)** en la ecuación **(4.9)**, tenemos:

$$A_{31}(-b_1)+A_{32}\left(-\frac{A_{21}(-b_1)}{A_{22}}+\frac{b_2}{A_{22}}-\frac{A_{23}}{A_{22}}x_3\right)+A_{33}x_3=b_1$$

Factorizando términos:

$$\left(A_{31}-\frac{A_{32}A_{21}}{A_{22}}\right)(-b_1)+\frac{A_{32}}{A_{22}}b_2+\left(A_{33}-\frac{A_{32}A_{23}}{A_{22}}x_3\right)=b_3$$

Escribiendo estas ecuaciones en forma matricial, y multiplicando la columna 2 por -1 y b_2 por -1, tenemos lo siguiente:

$$\begin{bmatrix} A_{11}-\frac{A_{12}A_{21}}{A_{22}} & \frac{A_{12}}{A_{11}} & A_{13}-\frac{A_{12}A_{23}}{A_{22}} \\ -\frac{A_{21}}{A_{11}} & -\frac{1}{A_{22}} & -\frac{A_{23}}{A_{22}} \\ A_{31}-\frac{A_{32}A_{21}}{A_{22}} & -\frac{A_{32}}{A_{22}} & A_{33}-\frac{A_{32}A_{23}}{A_{22}} \end{bmatrix} \begin{bmatrix} -b_1 \\ -b_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ b_3 \end{bmatrix} \qquad (4.11)$$

Efectuando las mismas operaciones para la tercera iteración, la ecuación matricial presentará la siguiente forma:

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \begin{bmatrix} -b_1 \\ -b_2 \\ -b_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad (4.12)$$

donde a partir de $A^{-1}b=x$, la ecuación **(4.12)** puede ser transformada como **[-C][b]=[x]** mediante la multiplicación de **[b]** por -1 y **[C]** por -1, por lo tanto:

$$\mathbf{A^{-1} = -C}$$

4.2.2 Implementación computacional

Generalmente, mediante la inversión Shipley-Coleman se efectúa una búsqueda de la diagonal más grande para actuar como pivote, similar al procedimiento de inversión de matrices mediante el método de Gauss-Jordan. Sin embargo, en aplicaciones relacionadas con corto circuito, no existe necesidad para la búsqueda de dicha diagonal. En análisis de sistemas de potencia, los elementos diagonales de cualquier matriz se dice que es dominante. Su valor es mayor que los elementos fuera de la diagonal.

A continuación se presentará el algoritmo de inversión Shipley-Coleman, el cual asume que existe una matriz de admitancias **Y**, de orden n. Posteriormente se proporcionará la implementación de dicho algoritmo bajo la forma de procedimiento, codificado en Borland C++ 3.1.

```
01: Procedimiento Inversion_Shipley (E n:entero)
02:
03: Variables
04:
05:  i, j, k  : entero
06:
07: Inicio
08:
```

```
09:   Paso N°01:  Desde k ← 1 hasta n hacer
10:
11:      Paso N°02:  Y[k][k] ← -1/Y[k][k]
12:
13:      Paso N°03:  Desde i ← 1 hasta n hacer
14:
15:        Paso N°04:  Si (i≠k) Entonces
16:
17:                      Y[i][k] ← Y[i][k]*Y[k][k]
18:
19:                  Fin_Si
20:
21:        Paso N°05:  Desde i ← 1 hasta n hacer
22:
23:          Paso N°06:  Si (i≠k) Entonces
24:
25:            Paso N°07:  Desde j ← 1 hasta n hacer
26:
27:              Paso N°08:  Si (j≠k) Entonces
28:
29:                            Y[i][j] ← Y[i][j] + Y[i][k]*Y[k][j]
30:
31:                        Fin_Si
32:
33:                    Fin_Si
34:
35:        Paso N°09:  Desde i ← 1 hasta n hacer
36:
37:          Paso N°10:  Si (i≠k) Entonces
38:
39:                        Y[k][i] ← Y[k][i]*Y[k][k]
40:
41:                    Fin_Si
42:
43:   Paso N°11:  Desde i ← 1 hasta n hacer
44:
45:      Paso N°12:  Desde j ← 1 hasta n hacer
46:
47:        Paso N°13:  Y[i][j] ← -Y[i][j]
48:
49: Fin_Procedimiento
```

Pseudocódigo 4.1: Procedimiento *Inversion_Shipley*

```
01: /**/void Inversion_Shipley(int n)
02:     {
03:       int  i, j, k ;
04:
05:       /* Inversion Shipley-Coleman */
06:
07:           for (k=0;k<n;k++)
08:           {
09:             /* Elemento Pivote */
10:
11:                Y[k][k] = -1/Y[k][k] ;
12:
13:             /* Columna Pivote */
14:
15:                for (i=0;i<n;i++)
16:                   if (i!=k)  Y[i][k] *= Y[k][k] ;
17:
18:             /* Elementos que no estan en una fila o columna pivote */
19:
20:                for (i=0;i<n;i++)
21:                   if (i!=k)
22:                      for (j=0;j<n;j++)
23:                         if (j!=k)   Y[i][j] += Y[i][k]*Y[k][j] ;
24:
25:             /* Elementos en una fila pivote */
26:
27:                for (i=0;i<n;i++)
28:                   if (i!=k)   Y[k][i] *= Y[k][k] ;
29:           }
30:
31:       /* Cambio de Signo */
32:
33:          for (i=0;i<n;i++)
34:             for (j=0;j<n;j++)   Y[i][j] *= -1 ;
35:
36:     }
```

Pseudocódigo 4.1: Subrutina *Inversion_Shipley*

A continuación, ejecutaremos la aplicación CALMI_A.CPP (cuyo código fuente se encuentra detallado en el Apéndice H), en el cual se obtiene la matriz de impedancias, a

partir de la matriz de admitancias. Se toma como dato el archivo BARRA23.DAT, en el cual se procesa, el programa obtiene la matriz de admitancia y sobre la misma variable bidimensional que contiene dicha matriz, se efectúa el método de inversión de Shipley-Coleman para obtener la matriz de impedancia, cuyo resultado se almacena en el archivo BARRA23.IMP (Ver Figuras 4.1 y 4.2).

Figura 4.1: Ejecución del programa CALMI_A.EXE

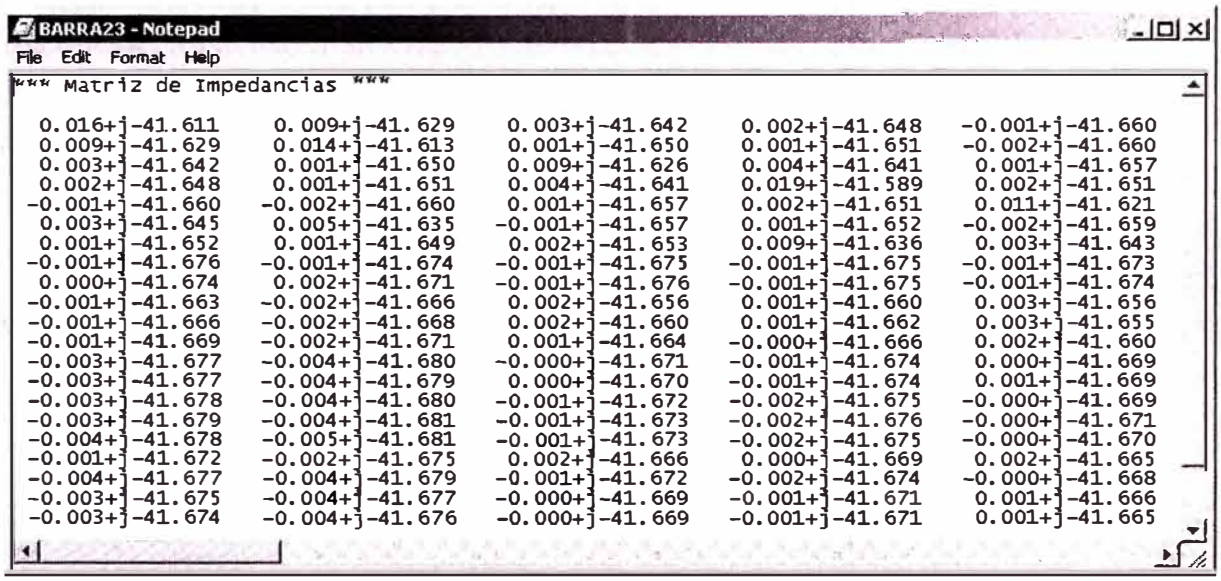


Figura 4.2: Contenido parcial de la matriz de impedancias (BARRA23.IMP)

4.3 Factorización matricial

4.3.1 Triangularización de una matriz

Triangularización es el proceso de aplicar la reducción de Gauss-Jordan para resolver un sistema de ecuaciones simultáneas de la forma $\boldsymbol{Ax=b}$. La factorización $\boldsymbol{LU}$ (donde **L** significa ***Lower*** – inferior – y **U** significa ***Upper*** – superior –) es sinónimo de trangularización. La factorización simplemente significa que cuando se multiplican las matrices $\boldsymbol{L}$ y $\boldsymbol{U}$, se obtiene la matriz $\boldsymbol{A}$. Dado $\boldsymbol{LU}$, se pueden efectuar operaciones "hacia delante" y "hacia atrás" sobre $\boldsymbol{b}$ para resolver $\boldsymbol{x}$ (esto último se hace necesario en el cálculo

de flujo de carga). Con LU también se puede obtener la inversa de A, la cual es utilizada en cálculos de corto circuito. La factorización es utilizada para resolver la matriz dispersa A. Disperso significa que existe una gran mayoría de elementos A_{ij} iguales a cero que aquellos diferentes de cero. Para sistemas de potencia, esto puede llegar a ser alrededor del 2% de elementos diferentes de cero en una matriz cuadrada A.

4.3.2 Factorización LU ordinaria

Se puede derivar la fórmula de para una factorización LU utilizando como ejemplo una matriz de tamaño 4×4. El problema es formulado a continuación.

Dada una matriz A, obtener los elementos de las matrices L y U, de tal manera que $LU=A$, donde las matrices L y U son factores de A.

Una vez que los elementos de L y U han sido obtenidos, el vector desconocido x en la ecuación $Ax=b$ puede ser determinado mediante sustitución hacia adelante o sustitución hacia atrás utilizando L y U. Existen variaciones en la estructura de L y U y otras referencias las denominan LDU. Escogeremos la forma mostrada a continuación:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ L_{21} & 1 & 0 & 0 \\ L_{31} & L_{32} & 1 & 0 \\ L_{41} & L_{42} & L_{43} & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \quad \textbf{(4.13)}$$

Al efectuar una multiplicación matricial, cada uno de los elementos A_{ij} puede ser interpretado como el producto de una fila de L y una columna de U. Hay que recordar que la matriz A es una cantidad conocida y que las matrices L y U son factores desconocidos de A.

La representación pictórica que muestra las operaciones sobre filas y columnas, se muestra a continuación en la Figura 4.3:

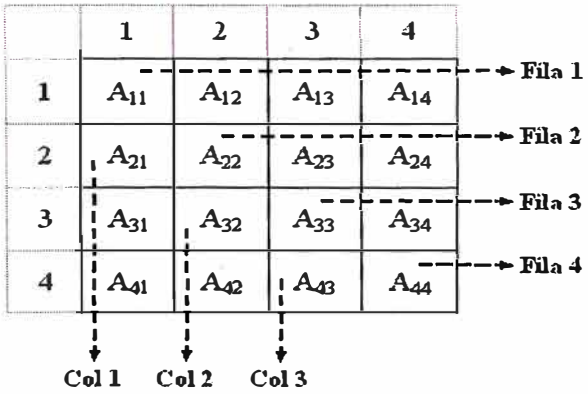


Figura 4.3: Filas y columnas a resolver

Para la Fila N°1 tenemos lo siguiente:

$$A_{11} = 1 \times U_{11} \Rightarrow U_{11} = A_{11}$$
$$A_{12} = 1 \times U_{12} \Rightarrow U_{12} = A_{12}$$
$$A_{13} = 1 \times U_{13} \Rightarrow U_{13} = A_{13}$$
$$A_{14} = 1 \times U_{14} \Rightarrow U_{14} = A_{14}$$

Para la Columna N°1 tenemos lo siguiente:

$$A_{21} = L_{21}U_{11} \Rightarrow L_{21} = A_{21}/U_{11}$$
$$A_{31} = L_{31}U_{11} \Rightarrow L_{31} = A_{31}/U_{11}$$
$$A_{41} = L_{41}U_{11} \Rightarrow L_{41} = A_{41}/U_{11}$$

Para la Fila N°2 tenemos lo siguiente:

$$A_{22} = L_{21}U_{12} + 1 \times U_{22} \Rightarrow U_{22} = A_{22} - L_{21}U_{12}$$
$$A_{23} = L_{21}U_{13} + 1 \times U_{23} \Rightarrow U_{23} = A_{23} - L_{21}U_{13}$$
$$A_{24} = L_{21}U_{14} + 1 \times U_{24} \Rightarrow U_{24} = A_{24} - L_{21}U_{14}$$

Para la Columna N°2 tenemos lo siguiente:

$$A_{32} = L_{31}U_{12} + L_{32}U_{22} \Rightarrow L_{32} = (A_{32} - L_{31}U_{12})/U_{22}$$
$$A_{42} = L_{41}U_{12} + L_{42}U_{22} \Rightarrow L_{42} = (A_{42} - L_{41}U_{12})/U_{22}$$

Para la Fila N°3 tenemos lo siguiente:

$$A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{43}U_{33} \Rightarrow U_{33} = A_{33} - L_{31}U_{13} - L_{32}U_{23}$$
$$A_{34} = L_{31}U_{14} + L_{32}U_{24} + U_{34} \Rightarrow U_{34} = A_{34} - L_{31}U_{14} - L_{32}U_{24}$$

Para la Columna N°3 tenemos lo siguiente:

$$A_{43} = L_{41}U_{13} + L_{42}U_{23} + L_{43}U_{33} \Rightarrow L_{43} = (A_{43} - L_{41}U_{13} - L_{42}U_{23})/U_{33}$$

Para un LU ordinario, se puede generalizar para una matriz de tamaño *n* la siguiente ecuación:

$$L_{ij} = \frac{1}{U_{ij}}\left(A_{ij} - \sum_{k=1}^{j-1} L_{ik}U_{kj}\right) \quad , \quad i = j+1,\ldots,N \qquad \textbf{(4.14)}$$

Para el elemento A_{44}, tenemos lo siguiente:

$$A_{44} = L_{41}U_{14} + L_{42}U_{24} + L_{43}U_{34} + U_{44} \Rightarrow U_{44} = A_{44} - L_{41}U_{14} - L_{42}U_{24} - L_{43}U_{34}$$

A continuación se generaliza la siguiente expresión:

$$U_{ij} = A_{ij} - \sum_{k=1}^{i-1} L_{ik}U_{kj} \quad , \quad j = 1,\ldots,N \qquad \textbf{(4.15)}$$

Al analizar las ecuaciones **(4.14)** y **(4.15)**, se puede concluir que no existe necesidad de almacenar L, U y A como tres matrices distintas. Se puede utilizar solamente la matriz A. Después de la factorización, la matriz A es destruída y reemplazada con L y U. La diagonal de la matriz L no se almacena.

4.3.3 Método LU-Dolittle (LUDM) para una matriz dispersa

Esta factorización es aplicable para matrices con almacenamiento intermedio. Después de procesar la primera fila y la primera columna respectivamente, se procesan aquellos elementos que se encuentran al interior del recuadro punteado, tal y como se muestra en la Figura 4.4:

	1	2	3	4
1	A_{11}	A_{12}	A_{13}	A_{14}
2	A_{21}	A_{22}	A_{23}	A_{24}
3	A_{31}	A_{32}	A_{33}	A_{34}
4	A_{41}	A_{42}	A_{43}	A_{44}

Figura 4.4: Submatriz a modificar, primera iteración

Para la primera iteración tenemos lo siguiente:

$$A'_{22} = A_{22} - L_{21}U_{12}$$
$$A'_{23} = A_{23} - L_{21}U_{13}$$
$$A'_{24} = A_{24} - L_{21}U_{14}$$
$$A'_{32} = A_{32} - L_{31}U_{12}$$
$$A'_{33} = A_{33} - L_{31}U_{13}$$
$$A'_{34} = A_{34} - L_{31}U_{14}$$
$$A'_{42} = A_{42} - L_{41}U_{12}$$
$$A'_{43} = A_{43} - L_{41}U_{13}$$
$$A'_{44} = A_{44} - L_{41}U_{14}$$

donde:

$$U_{22} = A'_{22}$$
$$U_{23} = A'_{23}$$
$$U_{24} = A'_{24}$$

Además:

$$L_{32} = A'_{32}/U_{22}$$
$$L_{42} = A'_{42}/U_{22}$$

Para la segunda iteración tenemos lo siguiente:

$$A''_{33} = A'_{33} - L_{32}U_{23}$$
$$A''_{34} = A'_{34} - L_{32}U_{24}$$
$$A''_{43} = A'_{43} - L_{42}U_{23}$$
$$A''_{44} = A'_{44} - L_{42}U_{24}$$

donde:

$$U_{33} = A''_{33}$$
$$U_{34} = A''_{34}$$
$$L_{43} = A''_{43}/U_{33}$$

Podemos verificar a partir de un LU ordinario lo siguiente:

$$U_{33} = A''_{33}$$
$$U_{33} = A'_{33} - L_{32}U_{23}$$
$$U_{33} = A_{33} - L_{31}U_{13} - L_{32}U_{23}$$

lo cual satisface el valor previamente obtenido. La representación pictórica de la segunda iteración, se muestra en la Figura 4.5:

	1	2	3	4
1	A_{11}	A_{12}	A_{13}	A_{14}
2	A_{21}	A_{22}	A_{23}	A_{24}
3	A_{31}	A_{32}	A_{33}	A_{34}
4	A_{41}	A_{42}	A_{43}	A_{44}

Figura 4.5: Submatriz a modificar, segunda iteración

Para matrices simétricas, no existe necesidad de almacenar la matriz ***L***. Al implementar computacionalmente este método se efectúa el la iteración ***k*** desde *1* hasta *N-1*.

Tal y como se muestra en la Figura 4.6 para la k-ésima iteración, se requiere la actualización de los A_{ij} al interior del recuadro punteado de color rojo. Para resolver un A_{ij} en particular, se requiere L_{ik} y U_{kj} apuntado por la pequeña línea punteada de color rojo. Sin embargo $L_{ik}=U_{ki}/U_{kk}$, por lo tanto para ahorrar memoria no existe necesidad de almacenar la matriz L debido a que su valor puede ser deducido a partir de la matriz U.

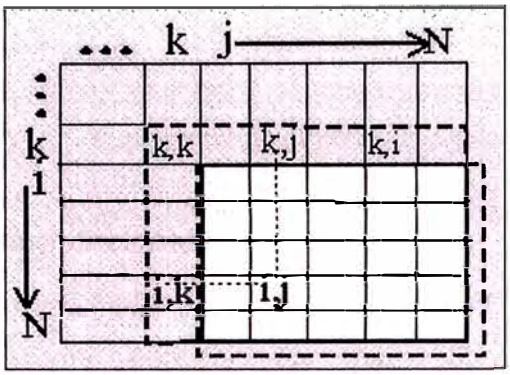


Figura 4.6: Representación pictórica de la ecuación (4.17)

Para un almacenamiento intermedio, guardando solamente los valores de U, tenemos:

$$A'_{ij} = A_{ij} - U_{kj} L_{ik} \qquad \textbf{(4.16)}$$

Pero, $L_{ik}=U_{ki}/U_{kk}$, por lo tanto:

$$A'_{ij} = A_{ij} - \frac{U_{kj} U_{ki}}{U_{kk}} \qquad \textbf{(4.17)}$$

La matriz original **[A]** es destruida. La matriz **[U]** resultante es almacenada en la misma matriz **[A]**, por lo cual solamente se utiliza una sola matriz, lo cual es similar al método de Shipley.

Al revisar la ecuación **(4.17)** y la Figura 4.7, nos encontramos en la iteración k y los rectángulos representan los nodos del lado derecho de k. En el ejemplo existen 4 y mediante la ecuación **(4.17)** se deben modificar 4 elementos diagonales y 6 elementos fuera de la diagonal a medida que se modifica solamente el triángulo superior.

Si la localización representada por ij no está en un LIFO entonces es un relleno. El propósito del ordenamiento de nodos es minimizar (sin llegar a un mínimo absoluto) el total de los rellenos introducidos durante la factorización.

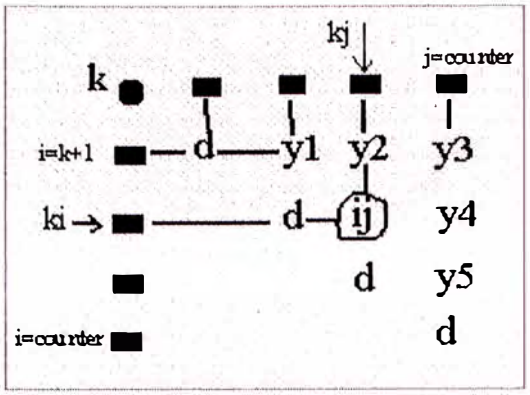


Figura 4.7: Representación dispersa de la ecuación (4.17)

4.3.4 Implementación computacional

A continuación se presentará el algoritmo de factorización por el método de Dolittle (LUDM), el cual asume que existe una matriz de admitancias **Y**, de orden n. Posteriormente se proporcionará la implementación de dicho algoritmo bajo la forma de procedimiento, codificado en Borland C++ 3.1.

```
01: Procedimiento Factorizacion_LUDM (E n:entero)
02:
03: Variables
04:
05:   i, j, k  : entero
06:
07: Inicio
08:
09:   Paso N°01:  Desde k ← 1 hasta n-1 hacer
10:
11:       Paso N°03:  Desde i ← k+1 hasta n hacer
12:
13:           Paso N°04:  Y[i][i] ← Y[i][i] - Y[k][i]*Y[k][i]/Y[k][k]
14:
15:           Paso N°05:  Desde j ← i+1 hasta n hacer
16:
17:               Paso N°06:  Y[i][j] ← Y[i][j] - Y[k][i]*Y[k][j]/Y[k][k]
18:
19: Fin_Procedimiento
```

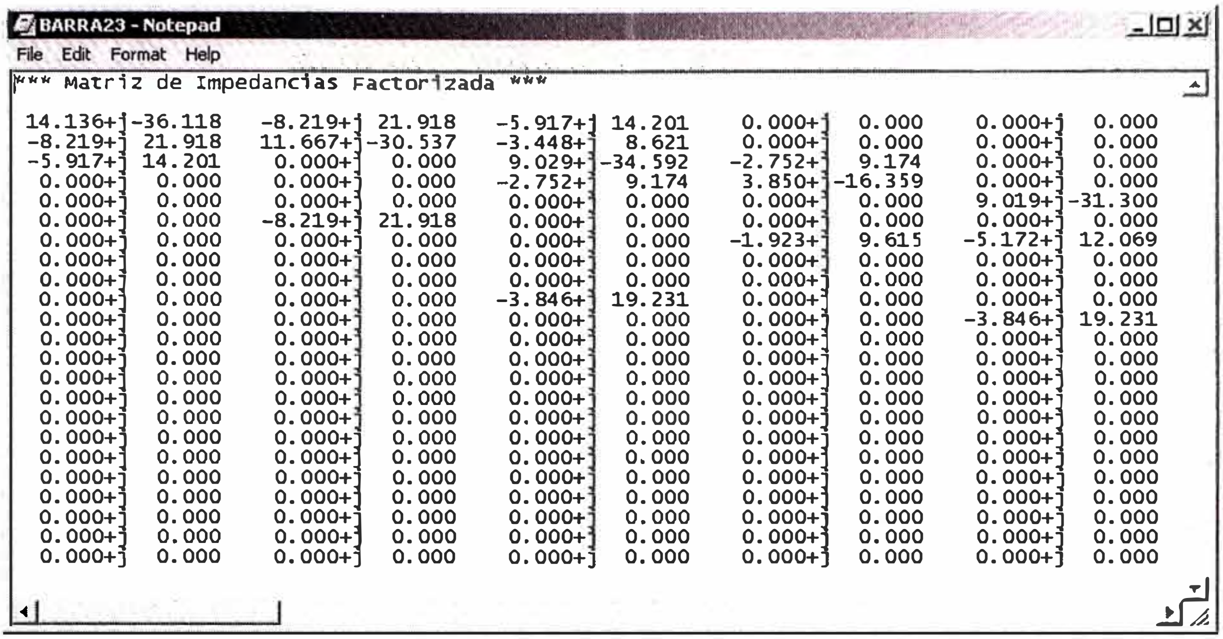
Pseudocódigo 4.2: Procedimiento *Factorizacion_LUDM*

```
01: /**/void Factorizacion_LUDM (int n)
02:     {
03:       int  i, j, k ;
04:
05:       /* Factorizacion Dolittle */
06:
07:          for (k=0;k<n-1;k++)
08:          {
09:            for (i=k+1;i<n;i++)
10:             {
11:               Y[i][i] -= Y[k][i]*Y[k][i]/Y[k][k] ;
12:
13:               for (j=i+1;j<n;j++)
14:                {
15:                  Y[i][j] -= Y[k][i]*Y[k][j]/Y[k][k] ;
16:                }
17:             }
18:          }
19:     }
```

Pseudocódigo 4.2: Subrutina *Factorizacion_LUDM*

A continuación, ejecutaremos la aplicación CALMF_A.CPP (cuyo código fuente se encuentra detallado en el Apéndice I), en el cual se obtiene la matriz de admitancias factorizada. Se toma como dato el archivo BARRA23.DAT, en el cual se procesa, el programa obtiene la matriz de admitancia y sobre la misma variable bidimensional que contiene dicha matriz, se efectúa el método de factorización de Dolittle para obtener la matriz factorizada, cuyo resultado se almacena en el archivo BARRA23.FAC (Ver Figuras 4.8 y 4.9).

Figura 4.8: Ejecución del programa CALMF_A.EXE



```
BARRA23 - Notepad
File  Edit  Format  Help
*** Matriz de Impedancias Factorizada ***

 14.136+j-36.118   -8.219+j 21.918   -5.917+j 14.201    0.000+j  0.000    0.000+j  0.000
 -8.219+j 21.918   11.667+j-30.537   -3.448+j  8.621    0.000+j  0.000    0.000+j  0.000
 -5.917+j 14.201    0.000+j  0.000    9.029+j-34.592   -2.752+j  9.174    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000   -2.752+j  9.174    3.850+j-16.359    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    9.019+j-31.300
  0.000+j  0.000   -8.219+j 21.918    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000   -1.923+j  9.615   -5.172+j 12.069
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000   -3.846+j 19.231    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000   -3.846+j 19.231
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
  0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000    0.000+j  0.000
```

Figura 4.9: Contenido parcial de la matriz de admitancias factorizada (CALMF_A.FAC)

4.4 Desarrollos posteriores

A partir de esta simple demostración se efectúan una serie de desarrollos ligados con el almacenamiento dinámico aplicado al área de sistemas de potencia:

A. Ordenamiento óptimo (Ver Apéndice C).

B. Representación de un sistema de potencia en grafos para efectuar reducción de nodos y ordenamiento óptimo.

C. Flujo de Potencia.

D. Modelamiento orientado a objetos. Tecnología empleada a partir de principios de la década de los 90s.

E. Producto de los modelos orientados a objetos, se tendrá la aplicación de algoritmos genéticos a flujo de potencia y despacho económico.

CONCLUSIONES

[01] La implementación del almacenamiento de una matriz dispersa basado en listas enlazadas tiene una naturaleza lineal, por lo cual ante sistemas eléctricos grandes se corre el riesgo de sacrificar velocidad para optimizar memoria. El problema de la velocidad puede sobrellevarse con equipos de cómputo suficientemente rápidos que lleven a cabo dicha tarea.

[02] La programación estructurada tiene sus limitaciones en cuanto al hecho de que no es lo suficientemente explícito o conveniente al momento de efectuar llamadas a subrutinas que permitan cumplir con una tarea específica. El panorama se torna un poco más complicado al momento de trabajar con estructuras de datos dinámicas. Cabe la necesidad de trabajar con técnicas de Programación Orientada a Objetos (POO), lo cual facilita que los algoritmos se relacionen de una manera más natural con la información del sistema.

[03] Las aplicaciones de computadora para sistemas de potencia permiten simplificar la preparación y procesamiento de datos, utilizar los avances logrados en ciencias de la computación (hardware, técnicas de cómputo, compiladores, sistemas operativos) lo que trae como consecuencia una reducción de costos y un incremento de flexibilidad.

[04] La algorítmica se constituye como una disciplina formal que permite el diseño de programas eficientes, así como un medio para que sea representado en diferentes lenguajes de programación.

[05] Si bien es cierto de que se tienen definiciones formales a nivel algorítmico, esto no representa una solución completa para describir el proceso de abstracción. En el caso específico de sistemas de potencia, se hace necesaria la extensión del lenguaje algorítmico para cumplir con los requerimientos de análisis y diseño de programas en ingeniería eléctrica.

[06] Se podría especular con la implementación de soluciones basadas en herramientas científicas como MATLAB. Esta propuesta puede ser válida, sin embargo, este

software básicamente se le utiliza como una herramienta de investigación debido a que su extensa librería de funciones genera una sobrecarga de operadores apreciable. A nivel de software de producción se prefiere desarrollar programas en C/C++, ya que su velocidad de ejecución es mayor, utilizándose el software científico como una herramienta de investigación, simulación y contraste de resultados.

[07] Las estructuras de datos estáticas tienen la ventaja de ser entidades más rápidas que las listas enlazadas en cuanto a su acceso a memoria. Sin embargo, para el caso de matrices dispersas no es conveniente su utilización por la gran cantidad de elementos nulos que contiene.

[08] Para complementar el esquema de listas enlazadas, se requiere de una rutina que permita ordenar secuencialmente los elementos de la matriz de admitancia, ya que se está utilizando una lista LIFO y para efectos de búsqueda no es conveniente que los elementos de la lista enlazada se encuentren desordenados.

[09] Un esquema de programación mal escogido o mal implementado puede echar a perder la mayor parte de los beneficios originados del aprovechamiento de la dispersión, así como otras características de las matrices de red, tales como la simetría, la dominancia diagonal, la simetría estructural, entre otras.

[10] Mediante un esquema dinámico de almacenamiento se consigue ampliar sustancialmente el potencial de las computadoras digitales mediante la reducción de los requisitos de memoria y tiempo de cálculo.

ANEXOS

ANEXO A
CODIGO PROGRAMA ***CALMA AM.CPP***

PROGRAMA CALMA_AM.CPP

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <complex.h>
#include <alloc.h>

/* Definicion de constantes */

   const int MAX = 23 ;

/* Definicion de variables  */

   /* Archivo de texto */

      FILE     *archivo   ;      //Manipula archivos de entrada y salida
      char     a_ent[30] ;       //Nombre de archivo de entrada
      char     a_sal[30] ;       //Nombre de archivo de salida

   /* Datos de linea */

      int      i  ,              //Barra de envio
               j    ;            //Barra de recepcion
      double   r  ,              //Resistencia de linea
               x  ,              //Reactancia de linea
               ys   ;            //Admitancia shunt de linea (Yc/2)

   /* Matriz de admitancias */

      complex  Y[MAX][MAX] ;     //Matriz de admitancias
      int      n ;               //Dimension de Y (Tama¤o maximo = MAX)


/* Definicion de procedimientos y funciones */

   void Inicializa_matriz() ;
   void Genera_reporte(int,char *) ;

void main()
 {

   /* Identificacion del archivos                        */

      clrscr() ;
      printf ("*** Calculo de la Matriz de Admitancias ***\n\n") ;
      printf ("* Archivo de entrada : ") ;
      scanf ("%s", &a_ent) ;
      printf ("* Archivo de salida : ") ;
      scanf ("%s", &a_sal) ;
      printf ("\n\n") ;

   /* Procesamiento del archivo de entrada               */

      if ((archivo=fopen(a_ent,"rt"))==NULL)
       printf ("Archivo no encontrado ... \n\n") ;
      else
       {
        printf ("Procesando archivo %s ...\n\n",a_ent) ;

        Inicializa_matriz() ;

        fscanf(archivo,"%d",&n) ;
        while(!feof(archivo))
         {
           fscanf(archivo,"%d%d%lf%lf%lf",&i,&j,&r,&x,&ys) ;
```

```
          if (i==j)
           Y[i-1][i-1] = Y[i-1][i-1] + 1/complex(r,x) + complex(0,ys) ;
          else
           {
             Y[i-1][i-1] = Y[i-1][i-1] + 1/complex(r,x) + complex(0,ys) ;
             Y[j-1][j-1] = Y[j-1][j-1] + 1/complex(r,x) + complex(0,ys) ;
             Y[i-1][j-1] = Y[j-1][i-1] = - 1/complex(r,x) ;
           }
        }

       Genera_reporte(n,a_sal) ;
      }

     fclose(archivo) ;

  /* Salida del programa                                    */

     printf ("\n\n") ;
     printf ("* Memoria utilizada en almacenamiento : %d bytes",sizeof(Y)) ;
     printf ("\n\n") ;

     printf ("Presione cualquier tecla para salir ...") ;
     getch() ;

}

/**/void Inicializa_matriz()
    {
      int  k, m ;

      for (k=0;k<MAX;k++)
      for  (m=0;m<MAX;m++)  Y[k][m] = complex(0,0) ;
    }

/**/void Genera_reporte(int n, char *nombre)
    {
      FILE  *arch ;
      int   k, m  ;

      arch = fopen(nombre,"wt") ;
      fprintf(arch,"*** Matriz de Admitancias ***\n\n") ;
      for (k=0;k<n;k++)
      {
        for (m=0;m<n;m++)
         fprintf (arch,"%7.3lf+j%7.3lf  ",real(Y[k][m]),imag(Y[k][m])) ;
        fprintf (arch,"\n") ;
      }
       fclose(arch) ;
    }
```

ANEXO B
CODIGO LIBRERÍA ***LIBLISTA.H***
CODIGO PROGRAMA ***CALMA LM.CPP***

LIBRERIA LIBLISTA.H

```
#define ListaVacia (cabecera==NULL)

typedef struct nodo puntero ;

typedef struct e_matriz {
        int      i ;
        int      j ;
        complex  y ;
       } ;

struct nodo {
        e_matriz adm ;
        puntero *next ;
       } ;


/**/ void error() ;
/**/ puntero *nuevo_elemento() ;
/**/ void insertar_adm(puntero **,e_matriz) ;
/**/ void eliminar_adm(puntero **,e_matriz) ;
/**/ puntero *buscar_adm(puntero *,e_matriz) ;
/**/ puntero *buscar_adm_pos(puntero *,int,int) ;
/**/ void ver_lista(puntero *) ;


/**/void error()
     {
       perror("error:insuficiente espacio de memoria")
       exit(1) ;
     }


/**/puntero *nuevo_elemento()
     {
       puntero *q = (puntero *) malloc(sizeof(puntero))
       if (!q) error() ;
       return(q) ;
     }



/**/void insertar_adm(puntero **cab,e_matriz dato)
     {
       puntero *cabecera=*cab ;
       puntero *actual=cabecera, *anterior=cabecera, *q

       if (ListaVacia)
       {
         cabecera = nuevo_elemento() ;
         cabecera->adm = dato ;
         cabecera->next =NULL ;
         *cab = cabecera ;
         return ;
       }

       q = nuevo_elemento() ;

       if (anterior==actual)
       {
         q->adm = dato ;
         q->next = cabecera ;
         cabecera=q;
       }
       else
       {
```

```
        q->adm = dato ;
        q->next = actual;
        anterior->next = q;
      }

      *cab = cabecera;
    }


/**/void eliminar_adm(puntero **cab,e_matriz dato)
    {
      puntero *cabecera = *cab ;
      puntero *actual = cabecera, *anterior = cabecera;
      int     boolean = 0 ;

      if (ListaVacia)
      {
        printf("Lista Vacia\n");
        return ;
      }

      boolean = dato.i!=actual->adm.i &&
              dato.j!=actual->adm.j &&
              real(dato.y)!=real(actual->adm.y) &&
              imag(dato.y)!=imag(actual->adm.y);

      while(actual!=NULL && boolean)
      {
        anterior = actual ;
        actual = actual->next;

        boolean = dato.i!=actual->adm.i &&
                dato.j!=actual->adm.j &&
                real(dato.y)!=real(actual->adm.y) &&
                imag(dato.y)!=imag(actual->adm.y);
      }

      if (actual==NULL)  return;

      if (anterior==actual)
      cabecera=cabecera->next;
      else
      anterior->next = actual->next;

      free(actual);

      *cab = cabecera;
    }


/**/puntero *buscar_adm(puntero *cabecera,e_matriz dato)
    {
      puntero *actual=cabecera;
      int     boolean = 0 ;

      boolean = dato.i!=actual->adm.i &&
              dato.j!=actual->adm.j &&
              real(dato.y)!=real(actual->adm.y) &&
              imag(dato.y)!=imag(actual->adm.y);

      while (actual!=NULL && boolean)
      {
        actual=actual->next ;

        boolean = dato.i!=actual->adm.i &&
                dato.j!=actual->adm.j &&
                real(dato.y)!=real(actual->adm.y) &&
```

```
                imag(dato.y)!=imag(actual->adm.y) ;
        }

        return(actual) ;
      }


/**/puntero *buscar_adm_pos(puntero *cabecera,int i,int j)
      {
        puntero *actual=cabecera ;
        int     boolean = 1 ;

        boolean = (actual->adm.i==i) && (actual->adm.j==j) ;

        while (actual!=NULL && !(boolean))
        {
          actual=actual->next ;
          boolean = (actual->adm.i==i) && (actual->adm.j==j) ;
        }

        return(actual) ;
      }


/**/void ver_lista(puntero *cabecera)
      {
        puntero *actual=cabecera ;
        if (ListaVacia)
        printf("Listaq Vacia\n") ;
        else
        {
          while (actual!=NULL)
           {
             printf("Y[%d,%d] = (%lf,%lf)\n",actual->adm.i,
                                        actual->adm.j,
                                        real(actual->adm.y),
                                        imag(actual->adm.y)  ) ;
             actual = actual->next ;
           }
          printf ("\n") ;
        }
      }
```

PROGRAMA CALMA_LM.CPP

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <complex.h>
#include <stdlib.h>
#include <alloc.h>
#include "LIBLISTA.H"

/* Definicion de variables  */

   /* Archivo de texto */

      FILE      *archivo   ;      //Manipula archivos de entrada y salida
      char      a_ent[30] ;       //Nombre de archivo de entrada
      char      a_sal[30] ;       //Nombre de archivo de salida

   /* Datos de linea */

      int       i  ,              //Barra de envio
                j    ;            //Barra de recepcion
      double    r  ,              //Resistencia de linea
                x  ,              //Reactancia de linea
                ys   ;            //Admitancia shunt de linea (Yc/2)

   /* Lista enlazada de admitancias */

      puntero   *inicio=NULL ;
      puntero   *q ;
      e_matriz  dato ;

   /* Variables para monitoreo de memoria */

      int       memoria=0 ;


/**/void genera_reporte(puntero *,char *) ;


void main()
 {

   /* Identificacion del archivos                        */

      clrscr() ;
      printf ("**** Calculo de la Matriz de Admitancias ***\n\n") ;
      printf ("** Archivo de entrada : ") ;
      scanf ("%s", &a_ent) ;
      printf ("** Archivo de salida : ") ;
      scanf ("%s", &a_sal) ;
      printf ("\n\n") ;

   /* Procesamiento del archivo de entrada               */

      if ((archivo=fopen(a_ent,"rt"))==NULL)
       printf ("Archivo no encontrado ... \n\n") ;
      else
       {
       printf ("Procesando archivo %s ...\n\n",a_ent) ;

        while(!feof(archivo))
         {
           fscanf(archivo,"%d%d%lf%lf%lf",&i,&j,&r,&x,&ys) ;

           if (i==j)
            {
```

```
            q = buscar_adm_pos(inicio,i,i) ;
            if (q)  q->adm.y = q->adm.y + 1/complex(r,x) + complex(0,ys) ;
            else
             {
               dato.i = i ; dato.j = i ; dato.y = 1/complex(r,x) +
                                                  complex(0,ys) ;
               insertar_adm(&inicio,dato) ;
                  memoria += sizeof(puntero) ;
             }
          }
         else
          {
            q = buscar_adm_pos(inicio,i,i) ;
            if (q)  q->adm.y = q->adm.y + 1/complex(r,x) + complex(0,ys) ;
            else
             {
               dato.i = i ; dato.j = i ; dato.y = 1/complex(r,x) +
                                                  complex(0,ys) ;
               insertar_adm(&inicio,dato) ;
               memoria += sizeof(puntero) ;
             }

            q = buscar_adm_pos(inicio,j,j) ;
            if (q)  q->adm.y = q->adm.y + 1/complex(r,x) + complex(0,ys) ;
            else
             {
               dato.i = j ; dato.j = j ; dato.y = 1/complex(r,x) +
                                                   complex(0,ys) ;
               insertar_adm(&inicio,dato) ;
               memoria += sizeof(puntero) ;
             }

            dato.i = i ; dato.j = j ; dato.y = - 1/complex(r,x) ;
            insertar_adm(&inicio,dato) ;
            memoria += sizeof(puntero) ;
            dato.i = j ; dato.j = i ;
            insertar_adm(&inicio,dato) ;
            memoria += sizeof(puntero) ;
          }
        }
      }

    genera_reporte(inicio,a_sal) ;
    fclose(archivo) ;

  /* Salida del programa                                    */

    printf ("\n\n") ;
    printf ("** Memoria utilizada en almacenamiento : %d bytes",memoria) ;
    printf ("\n\n") ;
    printf ("Presione cualquier tecla para salir ...") ;
    getch() ;

 }

/**/void genera_reporte(puntero *cabecera,char *nombre)
     {
      FILE  *arch ;
      puntero *actual=cabecera ;

      arch = fopen(nombre,"wt") ;
      fprintf(arch,"*** Matriz de Admitancias ***\n\n") ;
      if (ListaVacia)
      fprintf(arch,"Lista Vacia\n\n") ;
      else
      {
        while (actual!=NULL)
```

```
      {
        fprintf(arch,"Y[%d,%d] = %12.6lf +j%12.6lf\n",actual->adm.i,
                                                      actual->adm.j,
                                                      real(actual->adm.y),
                                                      imag(actual->adm.y)   ) ;
        actual = actual->next ;
      }
    }
    fclose(arch) ;
  }
```

ANEXO C
SOLUCIONES DIRECTAS FACTORIZADAS
DISPERSIDAD Y ORDENAMIENTO OPTIMO

SOLUCIONES DIRECTAS FACTORIZADAS

Esta sección muestra como deducir un arreglo de números a partir de una matriz no singular A que puede ser utilizada para obtener los efectos de cualquiera de las siguientes: A, A^{-1}, A^t, $(A^t)^{-1}$, y ciertas combinaciones híbridas de dos vías de estas matrices. El superíndice **–1** significa inversa y el superíndice **t** significa transpuesta. El esquema es aplicable a cualquier matriz no singular, real o compleja, dispersa o llena, simétrica o no. Inclusive los ejemplos en este en este artículo están limitados a ecuaciones nodales, el método también se aplica a ecuaciones de malla. Su mayor ventaja es verificada en problemas relacionados con grandes matrices dispersas.

El esquema a ser descrito en esta sección es similar a aquellos asociados con los nombres de Gauss, Doolittle, Choleski, Banachiewicz, y otros. Todos estos esquemas cercanamente relacionados son variaciones computacionales del proceso básico de triangularización de una matriz mediante transformaciones equivalentes. Estos fueron originalmente desarrollados para, y hasta hace muy recientemente solamente han sido descritos en términos de, procedimientos computacionales manuales. Muy poca atención se le ha proporcionado a su adaptabilidad especial para las matrices dispersas.

El esquema básico se presenta en primer término para el caso más general, una matriz llena no simétrica. La simetría es tratada como un caso especial. La dispersión con ordenamiento óptimo, el cual es el objetivo primario en el desarrollo de este artículo, es explicado en la sección III. Los ejemplos numéricos del esquema básico son dados en el Apéndice I.

1. Descomposición triangular

La descomposición de una matriz mediante eliminación gaussiana es descrita en variados libros de análisis matricial.

Ordinariamente, la descomposición es llevada a cabo mediante la eliminación de elementos que se encuentran debajo de la diagonal principal en columnas sucesivas. Desde el punto de vista de la programación de computadoras usualmente es más eficiente efectuar una eliminación por filas sucesivas. En consecuencia, el esquema ilustrado aquí es menos familiar pero matemáticamente equivalente.

El desarrollo de este esquema es basado en la ecuación:

$$Ax = b \qquad (1)$$

Donde A es una matriz no singular, x es un vector columna de incógnitas, y b es un vector conocido con al menos un elemento diferente de cero. En el algoritmo, A es aumentado por b, tal y como se muestra en **(2)** para un sistema de n-ésimo orden:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{21} & \cdots & a_{nn} & b_n \end{bmatrix} \qquad (2)$$

El primer paso es dividir los elementos de la primera fila entre a_{11}, tal y como se indica en **(3)**.

$$\begin{aligned} a_{1j}^{(1)} &= (1/a_{11})a_{1j} \qquad j = 2,\ldots,n \\ b_1^{(1)} &= (1/a_{11})b_1 \end{aligned} \qquad (3)$$

Los superíndices indican el orden del sistema deducido.

El segundo paso, tal y como se indica en **(4a)** y **(4b)**, es eliminar a_{21} de la segunda fila mediante una combinación lineal con la primera fila deducido de **(3)**, y luego dividir los elementos derivados restantes de la segunda fila entre su elemento diagonal deducido.

$$\begin{bmatrix} 1 & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & 1 & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} & b_n \end{bmatrix} \qquad (4a)$$

$$\begin{aligned} a_{2j}^{(1)} &= a_{2j} - a_{21}a_{1j}^{(1)} \qquad j = 2,\ldots,n \\ b_2^{(1)} &= b_2 - a_{21}b_1^{(1)} \\ a_{2j}^{(2)} &= \left(1/a_{22}^{(2)}\right)a_{2j}^{(1)} \qquad j = 3,\ldots,n \\ b_2^{(2)} &= \left(1/a_{22}^{(1)}\right)b_2^{(1)} \end{aligned} \qquad (4b)$$

El tercer paso, como se indica en **(5a)** y **(5b)**, es eliminar los elementos de la izquierda de la diagonal de la tercera fila y dividir los elementos deducidos restantes de la fila por el elemento diagonal derivado:

$$\begin{bmatrix} 1 & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & 1 & a_{23}^{(2)} & a_{24}^{(2)} & \cdots & a_{2n}^{(1)} & b_2^{(2)} \\ & & 1 & a_{34}^{(3)} & \cdots & a_{3n}^{(1)} & b_3^{(3)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & \cdots & a_{nn} & b_n \end{bmatrix} \qquad \textbf{(5a)}$$

$$\begin{aligned} a_{3j}^{(1)} &= a_{3j} - a_{31}a_{1j}^{(1)} \qquad j = 2,\ldots,n \\ b_3^{(1)} &= b_3 - a_{31}b_1^{(1)} \\ a_{3j}^{(2)} &= a_{3j}^{(1)} - a_{32}a_{2j}^{(2)} \qquad j = 3,\ldots,n \\ b_3^{(2)} &= b_3^{(1)} - a_{32}^{(1)}b_2^{(2)} \\ a_{3j}^{(3)} &= \left(1/a_{33}^{(2)}\right)a_{3j}^{(2)} \qquad j = 4,\ldots,n \\ b_3^{(3)} &= \left(1/a_{33}^{(2)}\right)b_3^{(2)} \end{aligned} \qquad \textbf{(5b)}$$

Procediendo de la misma manera, se obtiene el *n*-ésimo sistema deducido:

$$\begin{bmatrix} 1 & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} & b_1^{(1)} \\ & 1 & \cdots & a_{2n}^{(2)} & b_2^{(2)} \\ & & \ddots & \vdots & \vdots \\ & & & 1 & b_n^{(n)} \end{bmatrix} \qquad \textbf{(6)}$$

Debería notarse que al final del *k*-ésimo paso, se ha completado el trabajo sobre las filas 1 a *k* y que las filas *k+1* a *n* no han entrado aún en el proceso de manera alguna.

La solución ahora puede obtenerse mediante una sustitución hacia atrás:

$$\begin{aligned} x_n &= b_n^{(n)} \\ x_{n-1} &= b_{n-1}^{(n-1)} - a_{n-1,n}^{(n-1)}x_n \\ x_i &= b_i^{(i)} - \sum_{j=i-1}^{n} a_{ij}^{(i)}x_j \end{aligned} \qquad \textbf{(7)}$$

En programación, los x_i reemplazan a los b_i uno por uno a medida que son calculados, comenzando con x_n y trabajando hacia atrás hasta x_1.

Cuando A está lleno y de tamaño n, se puede demostrar que el número de operaciones de multiplicación y suma para una descomposición triangular es aproximadamente $1/3n^3$ comparado con n^3 para la inversión[5].

Puede ser fácilmente verificado que la triangularización en el mismo orden por columnas en lugar de filas podría haber producido idénticamente el mismo resultado. Cada elemento eliminado $a_{ij}^{(j-1)}$, i>j, podría haber sido el mismo así como el número de operaciones. La sustitución hacia atrás también pudo haber sido llevada a cabo por columnas en lugar de filas en el mismo número de operaciones.

2. Registro de operaciones

Si las operaciones hacia adelante en b han sido registradas de tal manera que podrían ser repetidas, es obvio que con este registro y la matriz triangular superior **(6)** para la sustitución hacia atrás, **(1)** puede ser resuelto para cualquier vector b sin tener que repetir la triangularización. Sin embargo, es trivial el registro de las operaciones hacia adelante. Cada operación hacia delante es completamente definida por las coordenadas de fila y columna y el valor de un único elemento $a_{ij}^{(j-1)}$, i ≥j, que aparece en el proceso. En consecuencia, es innecesario para llevar a cabo alguna actividad para registrar estos elementos excepto abandonarlos.

Las reglas para registrar las operaciones hacia delante en la triangularización de matrices son:

1) cuando es calculado un término $1/a_{ij}^{(j-1)}$, almacenarla en la ubicación ***ii***.

2) dejar cada término deducido $a_{ij}^{(j-1)}$, i >j, en el triángulo inferior.

Desde que tanto las operaciones de sustitución hacia delante como la sustitución hacia atrás son registradas en este esquema, ya no es necesario incluir el vector b. El resultado final de triangularizar A y registrar las operaciones hacia delante, es simbolizado en la ecuación **(8)**.

$$\begin{matrix} d_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ l_{21} & d_{22} & u_{23} & \cdots & u_{2n} \\ l_{31} & l_{32} & d_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & d_{nn} \end{matrix} \qquad (8)$$

Los elementos de **(8)**, definidos en términos de los sistemas deducidos de A en **(2)** a **(6)**, son:

$$\begin{aligned} d_{ii} &= 1/a_{ii}^{(i-1)} \\ u_{ij} &= a_{ij}^{(i)} \qquad i<j \\ l_{ij} &= a_{ij}^{(j-1)} \qquad i>j \end{aligned} \tag{9}$$

Los corchetes de la matriz son omitidos en **(8)** para enfatizar que el arreglo no es estrictamente una matriz en el mismo sentido que en el de los ejemplos precedentes, sino que representa un esquema de registro. Se hará referencia a este último como la ***tabla de factores***. En la literatura este resultado es mostrado frecuentemente como una factorización de la matriz inversa basado en el producto de una matriz triangular inferior y superior, pero es mas apropiado para esta discusión considerarlo como una tabla de factores.

3. Cálculo de soluciones directas

Es conveniente simbolizar las operaciones para obtener soluciones directas para definir algunas matrices especiales en términos de los elementos de la tabla de factores **(8)**. Las siguientes matrices no singulares difieren de la matriz principal solamente en la fila o columna indicada.

$$\begin{aligned} &D_i: \text{Fila } i = (0,0,...,0,d_{ii},0,...,0,0) \\ &L_i: \text{Columna } i = (0,0,...,0,1,-l_{i+1,i},-l_{i+2,i},...,-l_{n-1,i},-l_{n,i})' \\ &L_i^*: \text{Fila } i = (-l_{i,1},-l_{i,2},...,-l_{i,i-1},1,0,...,0,0) \\ &U_i: \text{Fila } i = (0,0,...,0,1,-u_{i,i+1},-u_{i,i+2},...,-u_{i,n-1},-u_{i,n}) \\ &U_i^*: \text{Columna } i = (-u_{1,i},-u_{2,i},...,-u_{i-1,i},1,0,...,0,0)' \end{aligned} \tag{10}$$

Las inversas de estas matrices son triviales. La inversa de la matriz D_i involucra solamente el recíproco del elemento d_{ii}. Las inversas de las matrices L_i, L_i^*, U_i y U_i^* involucran solamente la reversión de signos algebraicos de los elementos fuera de la diagonal.

Las operaciones de sustitución hacia a delante y hacia atrás sobre el vector columna b que es transformado en x puede ser expresado como premultiplicaciones por las matrices D_i, L_i o L_i^*, y U_i o U_i^*. Por lo tanto la solución de $Ax=b$ puede ser expresado tal y como se indica en **(11a)-(11d)**.

$$U_1U_2\cdots U_{n-2}U_{n-1}D_nL_{n-1}D_{n-1}L_{n-2}\cdots L_2D_2L_1D_1b = A^{-1}b = x \qquad \textbf{(11a)}$$

$$U_1U_2\cdots U_{n-2}U_{n-1}D_nL_n^*D_{n-1}L_{n-1}^*\cdots L_3^*D_2L_2^*D_1b = A^{-1}b = x \qquad \textbf{(11b)}$$

$$U_2^*U_3^*\cdots U_{n-1}^*U_n^*D_nL_{n-1}D_{n-1}L_{n-2}\cdots L_2D_2L_1D_1b = A^{-1}b = x \qquad \textbf{(11c)}$$

$$U_2^*U_3^*\cdots U_{n-1}^*U_n^*D_nL_n^*D_{n-1}L_{n-1}^*\cdots L_3^*D_2L_2^*D_1b = A^{-1}b = x \qquad \textbf{(11d)}$$

Cada una de estas cuatro ecuaciones describe una secuencia de operaciones sobre el vector *b* que es equivalente a la premultiplicación por A^{-1}. Para un sistema de n-ésimo orden, cada ecuación indica *n* multiplicaciones, n^2-*n* multiplicaciones-adiciones, y *n* sumas, excluyendo por supuesto, multiplicaciones por la unidad las cuales solamente son simbólicas. Esto corresponde exactamente con las n^2 multiplicaciones-adiciones requeridas para la premultiplicación por la inversa. Empezando con D_1 y procediendo hacia la izquierda, **(11a)** describe las operaciones de sustitución hacia adelante y hacia atrás que podrían ser efectuadas en *b* si esta aumentara *A* durante la triangularización por columnas. La ecuación **(11b)** describe el mismo resultado para la triangularización por filas. Las ecuaciones **(11c)** y **(11d)** describen otras secuencias de las mismas operaciones dando el mismo resultado. Dependiendo de las técnicas de programación, una de estas cuatro secuencias equivalentes usualmente probará ser la más conveniente.

Las soluciones directas de otros sistemas basados en A pueden ser obtenidas a partir de la tabla de factores mediante el uso de dos teoremas:

1) La inversa del producto de matrices de factores no singulares es el producto de las inversas de los factores en orden inverso.
2) La transpuesta de una matriz producto es el producto de las transpuestas de los factores en orden inverso.

A pesar de que la matriz *A* puede haberse perdido en la triangularización, su efecto es recuperable. Dado *x*, el vector *b* puede ser obtenido tal y como se indica en **(12a)** o **(12b)**. Otras dos ecuaciones análogas a **(11c)** y **(11d)** utilizando U_i^* también podrían ser escritas de la siguiente manera:

$$D_1^{-1}L_1^{-1}D_2^{-1}L_2^{-1}\cdots L_{n-1}^{-1}D_n^{-1}U_{n-1}^{-1}U_{n-2}^{-1}\cdots U_2^{-1}U_1^{-1}x = Ax = b \qquad \textbf{(12a)}$$

$$D_1^{-1}\left(L_2^*\right)^{-1} D_2^{-1}\left(L_3^*\right)^{-1}\cdots\left(L_n^*\right)^{-1} D_n^{-1}U_{n-1}^{-1}U_{n-2}^{-1}\cdots U_2^{-1}U_1^{-1}x = Ax = b \qquad \textbf{(12b)}$$

Nuevamente, el número de operaciones esenciales es n^2.

Las soluciones directas para el sistema transpuesto correspondiente $A^t y=c$ puede ser obtenido tal y como se indica en **(13)** y **(14)**.

$$D_1L_1^tD_2L_2^t\cdots L_{n-1}^tD_nU_{n-1}^tU_{n-2}^t\cdots U_2^tU_1^tc = \left(A^t\right)^{-1}c = y \qquad \textbf{(13)}$$

$$\left(U_1^t\right)^{-1}\left(U_2^t\right)^{-1}\cdots\left(U_{n-2}^t\right)^{-1}\left(U_{n-1}^t\right)^{-1} D_n^{-1}\left(L_{n-1}^t\right)^{-1}\cdots\left(L_2^t\right)^{-1} D_2^{-1}\left(L_1^t\right)^{-1} D_1^{-1}y = A^t y = c \qquad \textbf{(14)}$$

Nuevamente, el número de operaciones es n^2, y, como en los ejemplos previos, las ecuaciones podrían ser escritas utilizando L_i^* y U_i^*.

A pesar de que las ecuaciones **(11)-(14)** parecen de alguna manera complicados, ellos representan operaciones simples que pueden ser guiadas mediante la tabla de factores **(8)**. Cada ecuación indica lo siguiente:

1) Una secuencia para utilizar los elementos del arreglo para efectuar operaciones sobre el vector independiente.
2) Una regla para utilizar los subíndices de los elementos del arreglo para indicar los elementos del vector sobre el que va a ser operado.
3) Una regla de signo algebraico para los elementos l_{ij} y u_{ij}.
4) Una regla que indique si se van a multiplicar o dividir por los elementos d_{ii}.

Las operaciones pueden ser extendidas para incluir ciertas soluciones híbridas de dos vías con la matriz particionada en cualquier punto deseado. Sea $\boldsymbol{g}$ el vector columna híbrido definido como:

$$g^t = \left(b_1, b_2, \cdots, b_k, x_{k+1}, x_{k+2}, \cdots, x_n\right) \qquad \textbf{(15)}$$

Si g es dado, los primeros k elementos desconocidos de x y los $k+1$ hasta el n-ésimo elementos de b pueden ser obtenidos directamente. Primero es necesario calcular un vector z intermedio.

$$U_{n-1}^{-1}U_{n-2}^{-1}\cdots U_{k-2}^{-1}U_{k+1}^{-1}D_kL_k^*\cdots L_3^*D_2L_2^*D_1g = z \qquad \textbf{(16)}$$

La ecuación **(16)** indica la solución de un sistema triangular superior para obtener los primeros *k* elementos de *z* y la solución de un sistema triangular inferior independiente para obtener los elementos restantes de *z*.

Mediante la utilización de los elementos de *z* y *g*, el vector compuesto h^t es formado de la siguiente manera:

$$h^t = (z_1, z_2, \cdots, z_k, x_{k+1}, x_{k+2}, \cdots, x_n) \qquad \textbf{(17)}$$

Utilizando *h*, los primeros *k* elementos desconocidos de *x* son obtenidos a partir de **(18)**:

$$U_1 U_2 \cdots U_{k-1} U_k h = x \qquad \textbf{(18)}$$

La ecuación **(18)** define la sustitución hacia atrás de **(11a)** a partir de *k* hacia *1*.

Los *k+1* hasta el *n*-ésimo elemento desconocido de *b* son obtenidos mediante la siguiente expresión:

$$\left(L^*_{k+1}\right)^{-1} D^{-1}_{k+1} \cdots \left(L^*_{n-1}\right)^{-1} D^{-1}_{n-1} \left(L^*_n\right)^{-1} D^{-1}_n z = b' \qquad \textbf{(19)}$$

La ecuación **(19)** define la sustitución hacia atrás de **(12b)** a partir de *n* hasta *k+1*. El vector *b´* está compuesto de los primeros *k* elementos de *z*, los cuales no son afectados por la premultiplicación sobre el lado izquierdo de la ecuación, y los *k+1* hasta el *n*-ésimo elemento desconocido de *b*. Dado que los primeros *k* elementos de *b* fueron dados, la solución es completa. Nuevamente, el número total de operaciones es n^2.

Si solamente se desean obtener los elementos desconocidos de *x*, debe utilizarse la ecuación **(20)**

$$U_1 U_2 \cdots U_{k-1} U_k D_k L^*_k \cdots L^*_3 D_2 L^*_2 D_1 g = x \qquad \textbf{(20)}$$

Esto requiere solamente *kn* operaciones. Dado que no se requieren elementos de la tabla de factores por debajo de la *k*-ésima fila en este caso, no es necesario calcularlos y almacenarlos.

Una solución híbrida para la ecuación $A^t y = c$, en el cual los primeros *k* elementos de *c* y los *k+1* hasta el *n*-ésimo elemento de *y* son dados, puede ser desarrollada

análogamente a **(15)-(20)**. Otras soluciones híbridas también pueden ser desarrolladas, pero requieren más de n^2 operaciones y no serán consideradas.

4. Simetría

Si A es simétrica, solamente se requieren los términos d_{ii} y u_{ij} de la tabla de factores. Todas las soluciones directas precedentes pueden ser obtenidas notando que en el caso simétrico L_i puede ser definido en términos de D_i y U_i:

$$L_i = D_i U_i^t D_i^{-1} \qquad \textbf{(21)}$$

Sustituyendo **(21)** en **(11a)** y cancelando los factores adyacentes de la forma $D_i^{-1} D_i$ obtenemos:

$$U_1 U_2 \cdots U_{n-2} U_{n-1} D_n D_{n-1} U_{n-1}^t \cdots D_2 U_2^t D_1 U_1^t b = A^{-1} b = x \qquad \textbf{(22)}$$

Las expresiones más convenientes desde el punto de vista de la operación y notación pueden ser obtenidas notando que los productos de la forma $U_i^t D_j$, $i>j$, son conmutativos. Por lo tanto todos los factores D_i pueden ser agrupados en una matriz diagonal

$$D = D_1 D_2 \cdots D_n$$

Con este arreglo, la ecuación (22) se convierte en

$$U_1 U_2 \cdots U_{n-2} U_{n-1} D U_{n-1}^t \cdots U_2^t U_1^t b = A^{-1} b = x \qquad \textbf{(23)}$$

Sustituciones similares para el caso simétrico pueden efectuarse en las otras ecuaciones para soluciones directas.

La simetría inclusive permite un ahorro de casi la mitad de las operaciones de triangularización debido a que no es necesario efectuar operación alguna hacia la izquierda de la diagonal de tal forma que se obtengan los elementos derivados $a_{ij}^{(j-1)}$, $i>j$. Estos elementos pueden ser recuperados a partir de elementos previamente deducidos mediante la siguiente relación:

$$a_{ij}^{(j-1)} = \left(a_{jj}^{(j-1)}\right) a_{ji}^{(j)} \qquad i = 2,\ldots,n; \qquad j = 1,\ldots,i-1 \qquad \textbf{(24a)}$$

Un procedimiento alternativo evita un número de multiplicaciones igual a aquellos indicados en **(24a)**. En el caso no simétrico cada fila fue normalizada mediante la multiplicación de cada término a la derecha de la diagonal por el recíproco del término diagonal antes de proceder con la siguiente fila. Si este paso es diferido, al dejar los términos de una fila no normalizada hasta más adelante, la multiplicación de normalización y la recuperación del término simétrico, cuando se requiera, puede ser combinada. Por lo tanto después de procesar la fila i, cada término es dejado en la forma $a_{ij}^{(i-1)}$, $i<j$, en lugar de $a_{ij}^{(i)}$. Cuando el término $a_{ji}^{(i-1)}$, $j>i$, es requerido en el procesamiento de la fila j para ser utilizado como un multiplicador de la fila i de la misma manera como si la fila i hubiera sido normalizada, es calculada a partir de $a_{ij}^{(i-1)}$. Las operaciones que utilizan este procedimiento alternativo son de la forma que se muestra a continuación:

$$a_{jk}^{(i)} = a_{jk}^{(i-1)}\left[\left(1/a_{ii}^{(i-1)}\right)a_{ji}^{(i-1)}\right]a_{ik}^{(i-1)} \qquad i<j; \qquad k=j,\ldots,n \qquad \textbf{(24b)}$$

El término entre corchetes, el cual es $a_{ij}^{(i)}$ en la forma normalizada, reemplaza $a_{ij}^{(i-1)}$ tan pronto como haya sido utilizada en (24b). El ahorro en operaciones es trivial para una matriz completa, pero significativa para una matriz dispersa.

Si el sistema triangularizado no es normalizado por la izquierda, son requeridas n operaciones adicionales para cada solución directa. Esto es significativo para una matriz dispersa; en consecuencia, es recomendable la normalización descrita.

5. Modificaciones

Si A es alterada, es necesario modificar la tabla de factores para reflejar dicho cambio. Si un elemento $a_{k,m}$ de A es cambiado, los elementos de la tabla de factores con subíndices para fila y columna i, j son afectados de la siguiente manera:

1) Si k>m, los elementos a_{ij} con $i=k, j\geq m$ e $i>k, j\geq k$ son afectados.

2) Si k≤m, los elementos a_{ij} con $i\geq k, j=m$ e $i\geq m, j>m$ son afectados.

Dado que la mayoría de los cambios en una fila involucran un cambio en el término diagonal, el primer caso es lo más importante. El número de operaciones requerido para efectuar un cambio en la k-ésima fila es aproximadamente $1/3(n-k+1)^3$. Sin embargo, cualquier número de cambios adicionales en la submatriz inferior derecha limitado por una fila k y columna k puede ser incluido en el mismo paso sin incrementar el volumen de cálculos. Los cambios por debajo de la fila k en las columnas 1 hasta k-1

inclusive pueden efectuarse en el mismo paso con solamente un pequeño incremento en el volumen de cálculos. Si las filas en las cuales los cambios frecuentes que serán requeridos pueden ser anticipados, deberían localizarse al fondo de la matriz.

Al utilizar la inversa, cada cambio en A afecta todos los n^2 elementos de A^{-1}; en consecuencia, al menos n^2 operaciones sobre A^{-1} son requeridas para contar con cualquier cambio en A. En caso la inversa o la tabla de factores sea más fácil de modificar en una situación particular, depende de la naturaleza y el número de cambios matriciales requeridos y el esquema de la modificación inversa que está siendo utilizada.

6. Ventajas comparativas para una matriz completa

Cuando A es completa, las ventajas de la forma factorizada de la solución directa son las siguientes:

1) El arreglo de factores puede ser obtenido en un tercio del número de operaciones de la inversa.
2) La forma factorizada proporciona el efecto de A y de la matriz híbrida; esto no se aplica para la inversa.

Las ventajas de la inversa son las siguientes:

1) Las soluciones completas requieren solamente kn operaciones cuando el vector independiente tiene solamente k elementos diferentes de cero.
2) Bajo algunas circunstancias la inversa puede ser modificada para reflejar los cambios en la matriz original de manera más sencilla que la tabla de factores.

DISPERSIDAD Y ORDENAMIENTO ÓPTIMO

Cuando la matriz a ser triangularizada es dispersa, el orden en el cual las filas son procesadas afecta el número de términos diferentes de cero en el triángulo superior resultante. Si un esquema de programación es utilizado de tal manera que procesa y almacena solamente términos diferentes de cero, puede lograrse un gran ahorro en operaciones y memoria de computadora manteniendo la tabla de factores tan dispersa como sea posible. El orden óptimo absoluto de la eliminación podría resultar en la menor cantidad de posibles términos en la tabla de factores. Un algoritmo eficiente para determinar el orden óptimo absoluto no ha sido desarrollado, y aparentemente no tiene posibilidad práctica. Sin embargo, han sido desarrollados varios esquemas efectivos para determinar ordenamientos cercanamente óptimos.

1. Esquemas para el ordenamiento cuasi-óptimo

La inspección de algoritmos para un ordenamiento cercanamente óptimo a ser descritos son aplicables a matrices dispersas que son simétricas con un patrón de términos diferentes de cero fuera de la diagonal; por ejemplo, si a_{ij} es diferente de cero, entonces a_{ji} también es diferente de cero pero no necesariamente igual a a_{ij}. Estas son matrices que ocurren con mayor frecuencia en problemas de red. Desde el punto de vista de la eficiencia en programación, los algoritmos deberían ser aplicados antes, en lugar que durante, la triangularización. Se asume en adelante que las filas de la matriz son originalmente numeradas de acuerdo a algún criterio externo y reenumeradas de acuerdo al algoritmo de inspección. Las eliminaciones son llevadas a cabo en una secuencia ascendente del sistema renumerado.

A continuación se tienen las descripciones de los tres esquemas para renumerar en un orden cercanamente óptimo. Están listados en orden creciente de complejidad en programación, tiempo de ejecución, y optimalidad.

1) El número de filas de acuerdo al número de términos diferentes de cero fuera de la diagonal antes de la eliminación. En este esquema las filas con solamente un término fuera de la diagonal son numerados en primer lugar, aquellos con dos términos en segundo lugar., etc., y aquellos con la mayor cantidad de términos, al último. Este esquema no toma en cuenta cualquiera de los efectos posteriores del proceso de eliminación. La única información necesaria es una lista del número de términos diferentes de cero en cada fila de la matriz original.

2) El número de filas de tal manera que en cada paso del proceso, la siguiente fila sobre la que se va a operar es la que introducirá la menor cantidad de elementos diferentes de cero. Si más de una fila cumple con este criterio seleccione cualquiera. Este esquema requiere una simulación de los efectos sobre la acumulación de términos diferentes de cero del proceso de eliminación. La información de entrada es una lista por filas de los números de columna de los términos diferentes de cero fuera de la diagonal.

3) Número de filas de tal manera que en cada paso del proceso la siguiente fila sobre la que se va a operar es aquella que introducirá la menor cantidad de términos diferentes de cero. Si más de una fila cumple con este criterio, seleccione cualquiera. Esto involucra una simulación de prueba de cada alternativa factible

del proceso de eliminación en cada paso. La información de entrada es la misma como en el esquema (2).

Las ventajas comparativas de estos esquemas son influenciados por la topología de la red y el tamaño y número de soluciones directas requeridas. La única virtud del esquema (1) es su simplicidad y velocidad. Para ecuaciones nodales de una red eléctrica el esquema (2) es suficientemente mejor que el esquema (1) para justificar el tiempo adicional requerido para su ejecución. El esquema (3) no parece ser suficientemente mejor que el esquema (2) para justificar su uso en redes eléctricas, pero se cree que puede ser más efectivo para otras redes. Dado que la experiencia del autor está limitada a redes eléctricas, estas conclusiones pueden ser inválidas para otras redes. Otros algoritmos pueden necesitar ser desarrollados. Es posible la implementación de esquemas intermedios entre (1) y (2) o entre (2) y (3), así como otros esquemas distintos.

Puede demostrarse que existen algunas matrices para las cuales ninguno de los esquemas dados será suficientemente efectivo. Sin embargo, es poco probable su ocurrencia en la mayoría de ecuaciones de red.

2. Otros factores que influencian el ordenamiento

Bajo determinadas condiciones puede ser ventajoso o necesario numerar al último ciertas filas a pesar de que esto afecta adversamente la dispersidad. Entre estas condiciones se encuentran las siguientes:

1) Es conocido que los cambios en la matriz original ocurrirán solamente en pocas y determinadas filas. Si estas filas son numeradas al último, solamente estas filas en la tabla de factores necesitan ser modificados para reflejar estos cambios.
2) Es conocido que los cambios en el vector independiente ocurrirán solamente en pocos y determinados elementos. Si las filas son numeradas de tal manera que no ocurra cambio alguno en el vector sobre la fila k, las operaciones hacia adelante que preceden la fila k necesitan ser llevadas a cabo solamente una vez, y no son repetidas para cada caso subsecuente.
3) La matriz es solo ligeramente no simétrica. Si las filas son numeradas de tal manera que la porción no simétrica de la matriz es la última, puede tomarse ventaja de la simetría hasta este punto.
4) La operación híbrida va a ser utilizada y es necesario tener las filas apropiadas al último.

Cuando la red en la cual la matriz se basa, se compone de las subredes con relativamente pocas interconexiones.

El patrón de una matriz de admitancias nodal de una red eléctrica grande compuesta de tres subredes. El patrón de la tabla de factores derivado de esta matriz podría ser similar. Las submatrices *A*, *B* y *C*, las cuales presumiblemente son bastante dispersas, representan los nodos dentro de las subredes *A*, *B* y *C* respectivamente; y *S* representa los nodos asociados con sus interconexiones. Todas las submatrices fuera de *A*, *B*, *C* y *S* son cero excepto aquellos indicados por las *X* los cuales contienen los pocos términos fuera de la diagonal que interconectan el sistema en *S*. El algoritmo de renumeración debería ser aplicado a cada submatriz independientemente. Dentro de cada una de las submatrices *A*, *B* y *C*, las filas con términos diferentes de cero en las columnas de *S* deberían ser ubicadas al último. Este arreglo no será conseguido ordinariamente por cualquiera de los algoritmos de renumeración dados sin la adición de código lógico o externo.

El arreglo ofrece las siguientes ventajas:

1) Si es efectuado un cambio en una de las submatrices, afecta solamente a *B* y *S* en las tabla de factores, mas no a *A* y *C*.
2) Si después de obtener la solución, es efectuado un cambio en el vector independiente en las filas de únicamente una partición, digamos *B*, solamente las operaciones sobre el vector definidas por *B* y *S* necesitan ser repetidas para obtener una nueva solución.
3) Si la memoria de la computadora es limitada, el procedimiento puede ser implementado mediante la operación de una sola submatriz por vez.
4) A medida que más subredes sean añadidas a un sistema, la relación entre el tamaño del sistema y la tabla de factores tiende a ser casi lineal.

3. Efectividad del ordenamiento óptimo

La efectividad del ordenamiento óptimo puede ser expresado como el cociente entre el número de términos diferentes de cero fuera de la diagonal en la tabla de factores y el número de términos similares en la matriz original. Mientras más próximo sea este cociente a la unidad, más efectivo será el ordenamiento. El cociente es influenciado en mayor medida por la topología de la red antes que por el tamaño del sistema. Obviamente, puede ser diferente para matrices basadas en otros tipos de redes.

Cuando la matriz es dispersa, las operaciones aritméticas requeridas para calcular la tabla de factores para un sistema de n-ésimo orden son como siguen:

$$\text{divisiones} = n$$

$$\text{multiplicaciones} = s = \sum_{i=1}^{n-1} r_i \qquad \textbf{(26a)}$$

El número de multiplicaciones-sumas es dependiente en si la matriz es simétrica o no. Para matrices no simétricas con patrón simétrico de elementos diferentes de cero fuera de la diagonal:

$$\text{multiplicaciones-sumas} = s = \sum_{i=1}^{n-1} r_i^2 \qquad \textbf{(26b)}$$

Para matrices simétricas:

$$\text{multiplicaciones-sumas} = s = \sum_{i=1}^{n-1} 2\left(r_i^2 + r_i\right) \qquad \textbf{(26c)}$$

donde r_i es el número de términos diferentes de cero a la derecha de la diagonal en la fila i en la tabla de factores y s es el número total de dichos términos.

Al determinar el conteo de operaciones para cambiar el arreglo de factores para reflejar los cambios en la matriz original, es algo complicado y dependiente del esquema de programación actual. En general, si los cambios en A comienzan en la fila k, **(26a)-(26c)** puede ser utilizado con las sumas comenzando en k en lugar de 1 para obtener un conteo aproximado.

Las operaciones requeridas para calcular cualquiera de las posibles soluciones directas completas basadas en A son:

$$\text{multiplicaciones o divisiones} = n$$

$$\text{sumas} = n \qquad \textbf{(26c)}$$

$$\text{multiplicaciones-sumas} = 2s$$

Las soluciones parciales pueden ser obtenidas en una menor cantidad de operaciones.

4. Programación

Una presentación exhaustiva de las diversas técnicas de programación que han sido desarrolladas para el método se encuentra más allá de el ámbito de este artículo. Algunos han sido discutidos en otros artículos. [1], [4], [10]

Cuando se trabaja con una matriz completa, la dirección en la memoria de la computadora de cada elemento de la matriz puede ser relacionado con los índices de fila y columna de tal manera que la programación se haga menos complicada. Sin embargo, a fin de obtener los beneficios de la dispersidad, el esquema de programación debe almacenar y procesar solamente elementos diferentes de cero. Esto requiere, adicionalmente a la localización en memoria para los mismos elementos de la matriz, de tablas para indexar información para identificar los elementos y facilitar su direccionamiento. La programación es más complicada en este caso y mucho del potencial del método puede perderse a lo largo de una programación pobremente planeada. La necesidad de programación experta no puede ser sobre enfatizada.

Con las técnicas más efectivas de programación, las operaciones son llevadas a cabo como si estuviesen siendo efectuadas mediante inspección visual y cálculo manual. En la finalización del algoritmo de ordenamiento óptimo [esquema (2) o (3)], la forma exacta de la tabla de factores es establecida y su información es registrada en varias tablas para guiar el proceso de eliminación actual. Durante la eliminación no se efectúa operación alguna que pudiera conducir a un resultado nulo predecible y no se efectúa localización en memoria alguna para un elemento cero predecible.

La matriz original, la tabla de factores, y todas las tablas de índices contienen solamente elementos diferentes de cero.

Las filas son transferidas a partir de la matriz original hacia una fila de trabajo compacta en las cuales los elementos a la izquierda de la diagonal son eliminadas mediante una apropiada combinación lineal con filas previamente procesadas a partir de una tabla de factores parcialmente completada. Cuando ha sido completado el trabajo en la fila, se le añade a la tabla de factores. El procedimiento actual varía dependiendo de la naturaleza de la aplicación. Si la matriz no es simétrica, los elementos derivados a la izquierda de la diagonal deben ser almacenados. El arreglo de las tablas depende en algún grado de las necesidades subsecuentes para las soluciones directas.

Un ejemplo de una posibilidad para arreglar la tabla de factores es indicado en la Tabla N°1. Se representa en el resultado final para una matriz simétrica de sétimo orden.

Las columnas indicadas con **Loc** hace referencia a las direcciones relativas en la memoria de la computadora. La i-ésima localización en la tabla D contiene el elemento d_{ii} y la localización en la tabla U del primer elemento u_{ij} de la fila i. La columna de la tabla etiquetada J contienen el subíndice de la columna j de u_{ij}. Por lo tanto la fila 2 empieza en la localización 3 de la tabla U y contiene u_{23}, u_{26}, y u_{27} en las localizaciones 3, 4 y 5 respectivamente. La fila 3 empieza en la localización 6.

Tabla D		Tabla U			
Loc	**Factores D**	**Loc en la tabla U**	**Loc**	**Factores U**	**J**
1	d_{11}	1	1	u_{12}	2
2	d_{22}	3	2	u_{17}	7
3	d_{33}	6	3	u_{23}	3
4	d_{44}	8	4	u_{26}	6
5	d_{55}	10	5	u_{27}	7
6	d_{66}	11	6	u_{36}	6
7	d_{77}	12	7	u_{37}	7
8	-	12	8	u_{45}	5
			9	u_{46}	6
			10	u_{56}	6
			11	u_{67}	7

Tabla N°1: Ejemplo de almacenamiento y esquema de indexación para la tabla de factores.

5. Ventajas comparativas para una matriz dispersa

Cuando A es dispersa, las ventajas de la forma factorizada, adicionalmente a las aquellas previamente listadas son:

1) La tabla de factores puede ser obtenida en una pequeña fracción de tiempo requerido por la inversa.
2) El requerimiento de almacenamiento es pequeño, permitiendo la solución de sistemas más grandes.
3) Las soluciones directas pueden ser obtenidas más rápidamente a menos que el vector independiente sea extremadamente disperso.
4) El error de redondeo es reducido.
5) Las modificaciones debidas a los cambios en la matriz puede ser efectuado de manera más rápida.

La única desventaja del método es que requiere técnicas de programación mucho más sofisticadas.

ANEXO D
PUNTEROS Y LISTAS ENLAZADAS

PUNTEROS Y LISTAS ENLAZADAS

Una *lista lineal* es un conjunto de elementos de un tipo de dato que se encuentran ordenados y pueden variar en número. Ésta es una definición muy general, que incluye los archivos y los vectores.

Los elementos de una lista lineal se almacenan normalmente continguos – un elementos detrás de otro – en posiciones consecutivas de la memoria. Las sucesivas entradas en una guía o directorio telefónico, por ejemplo, están en líneas sucesivas, excepto en las partes superior e inferior de cada columna. Una lista lineal se almacena en la memoria principal de una computadora en posiciones sucesivas de memoria; cuando se almacenan en cinta magnética, los elementos sucesivos se presentan en sucesión en la cinta. Esta asignación de memoria se denomina *almacenamiento secuencial*. Posteriormente, se verá que existe otro tipo de almacenamiento denominado *encadenado o enlazado.*

Las líneas así definidas se denominan *contiguas.* Las operaciones que se pueden realizar con listas lineales contiguas son:

1) Insertar, eliminar o localizar un elemento.
2) Determinar el tamaño – número de elementos – de la lista.
3) Recorrer la lista para localizar un determinado elemento.
4) Clasificar los elementos de la lista en orden ascendente o descendente.
5) Unir dos o más listas en una sola.
6) Dividir una lista en varias sublistas.
7) Copiar la lista.
8) Borrar la lista.

Una lista lineal se almacena en la memoria de la computadora en posiciones sucesivas o adyacentes y se procesa como un array unidimensional. En este caso, el acceso a cualquier elemento de la lista y la adición de nuevos elementos es fácil; sin embargo, la inserción o borrado requiere un desplazamiento de lugar de los elementos que le siguen y, en consecuencia, el diseño de un algoritmo específico.

Para permitir operaciones con listas como arrays se deben dimensionar éstos con tamaño suficiente para que contengan todos los posibles elementos de la lista.

Listas enlazadas

Los inconvenientes de las listas contiguas se eliminan con las listas enlazadas. Se pueden almacenar los elementos de una lista lineal en posiciones de memoria que no sean contiguas o adyacentes.

Una *lista enlazada o encadenada* es un conjunto de elementos en los que cada elemento contiene la posición – o dirección – del siguiente elemento de la lista. Cada elemento de la lista enlazada debe tener al menos dos campos: un campo que tiene el valor del elemento y un campo (enlace, link) que contiene la posición del siguiente elemento, es decir, su conexión, enlace o encadenamiento. Los elementos de una lista son enlazados por medio de los campos enlaces.

Las listas enlazadas tienen una terminología propia que se suele utilizar normalmente. Primero, los valores se almacenan en un *nodo* (Figura N°1).

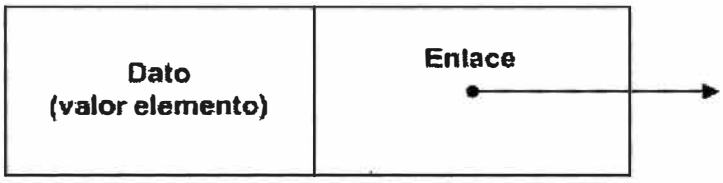


Figura 1: Nodo con dos campos

Una lista enlazada se muestra en la Figura N°2

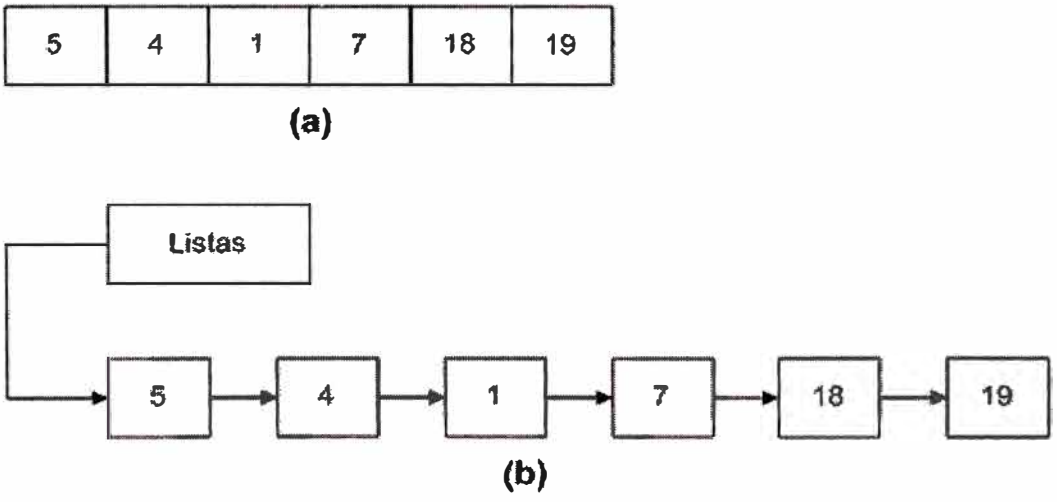


Figura 2: (a) Array representado por una lista; (b) lista enlazada representada por enteros

Los componentes de un nodo se llaman *campos*. Un nodo tiene al menos un campo *dato* o *valor* y un *enlace* (indicador o puntero) con el siguiente campo. El campo enlace apunta (proporciona la dirección de) al siguiente nodo de la lista. El último nodo de la lista

enlazada, por convenio se suele representar por un enlace con la palabra reservada **nil** (nulo), una barra inclinada (/) y, en ocasiones, el símbolo eléctrico de tierra o masa (Figura N°3)

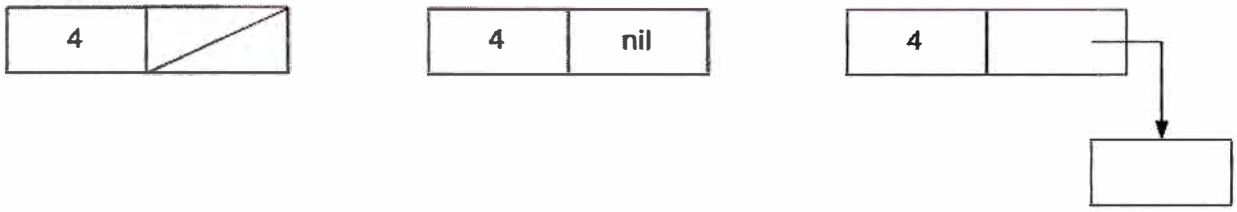


Figura 3: Representación del último nodo de una lista

La implementación de la lista enlazada depende del lenguaje. Pascal, C, C++, Ada y Modula-2 utilizan como enlace una *variable puntero*, o *puntero* simplemente. Los lenguajes como FORTRAN y BASIC no disponen de este tipo de datos y se debe simular con una variable entera que actúa como indicador o cursor.

Un **puntero** (apuntador) es una variable cuyo valor es la dirección o posición de otra variable. En las listas enlazadas no es necesario que los elementos de la lista sean almacenados en posiciones físicas adyacentes, ya que el puntero indica donde se encuentra el siguiente elemento de la lista, tal como se indica en la Figura N°4.

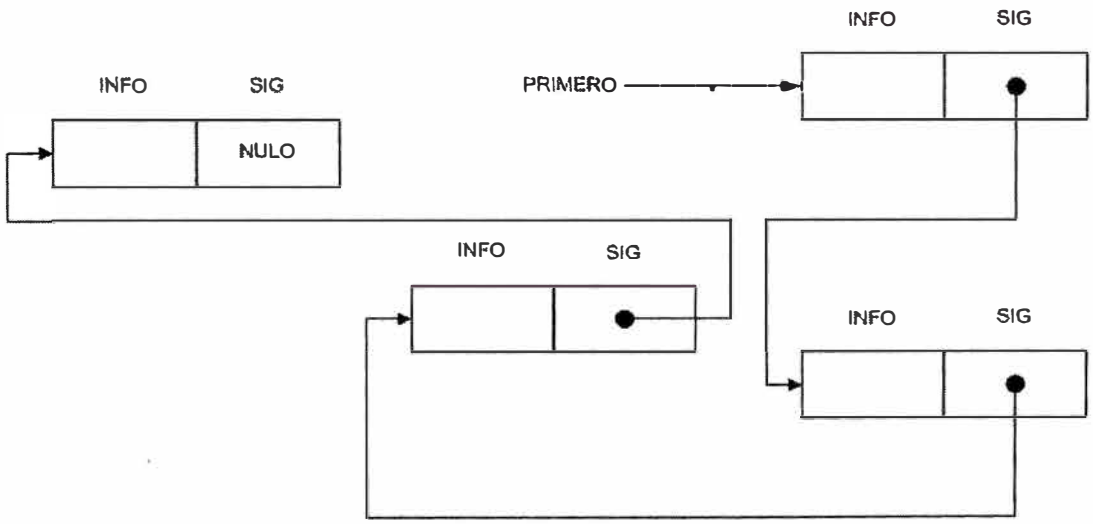


Figura 4: Elementos no adyacentes de una lista enlazada

Por consiguiente, la inserción y borrado no exigen desplazamiento como en el caso de las listas contiguas.

Una lista enlazada sin ningún elemento se llama *lista vacía*. Su puntero inicial o de cabecera tiene el valor nulo (nil).

Una lista enlazada se define por:

- El tipo de sus elementos: campo de información (datos) y campo de enlace (puntero).
- Un puntero de cabecera que permite acceder al primer elemento de la lista.
- Un medio para detectar el último elemento de la lista: *puntero nulo* (nil).

Procesamiento de listas enlazadas

Para procesar una lista enlazada son imprescindibles las siguientes informaciones:

- El primer nodo (cabecera de la lista)
- El tipo de sus elementos

Las operaciones que normalmente se ejecutan con listas incluyen:

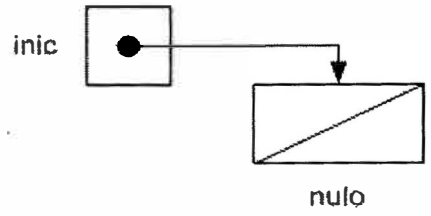
1) Recuperar información de un nodo específico (acceso a un elemento).
2) Encontrar el nodo que contiene una información específica (localizar la posición de un elemento dado).
3) Insertar un nuevo nodo en un elemento de la lista.
4) Insertar un nuevo nodo en relación a una información particular.
5) Borrar (eliminar) un nodo existente que contiene información específica.

La creación de la lista conlleva la inicialización a nulo del puntero (Inc.), que apunta al primer elemento de la lista.

```
tipo
  puntero_a nodo : punt
  registro:  tipo_elemento
        ... : ...
        ... : ...
  fin registro
  registro: nodo
    inicio
      tipo_elemento : elemento
      punt          : sig
  fin registro

var  punt           : inic
     punt           : posic
     punt           : anterior
     tipo_elemento  : elemento
     logico         : encontrado

inicio
  inicializar (inic)
fin
```



La inserción tiene dos casos particulares:

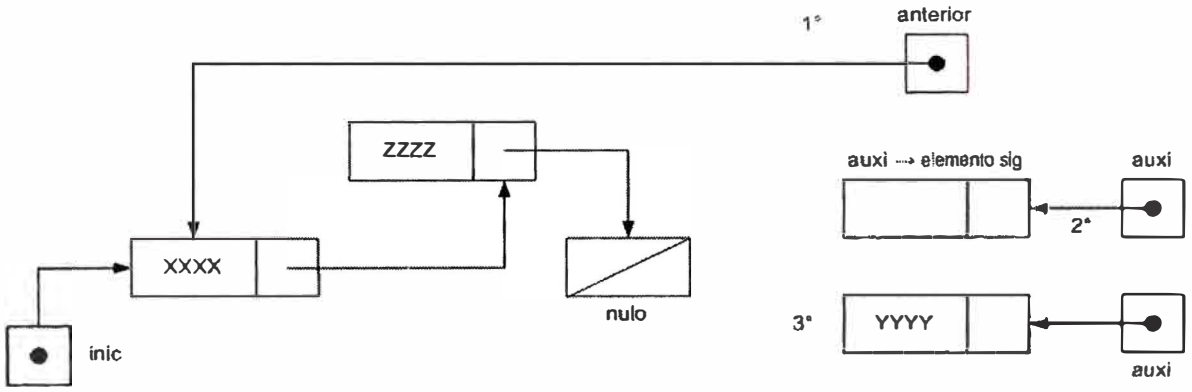
- Insertar el nuevo nodo en el frente, principio de la lista.
- Insertar el nuevo nodo en cualquier otro lugar de la lista.

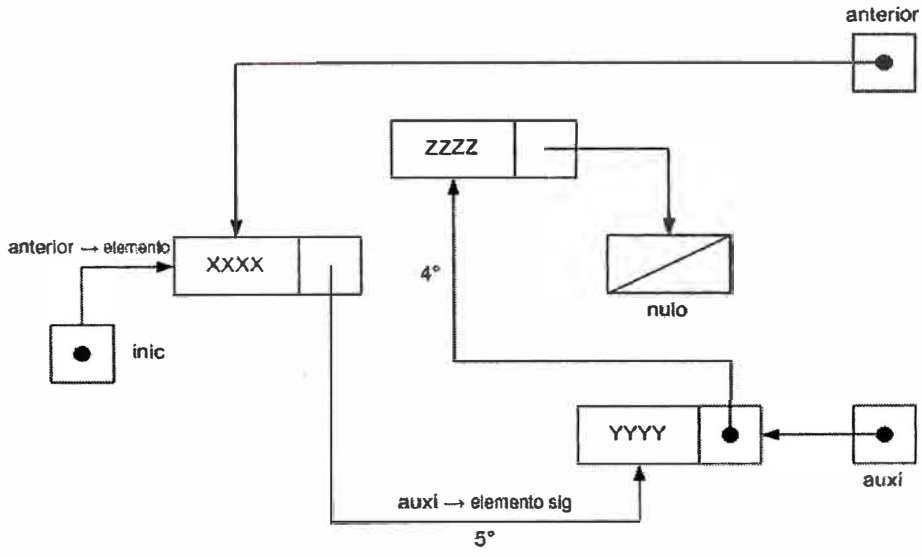
El procedimiento insertar, inserta un nuevo elemento al comienzo de anterior; si anterior fuera nulo significa que ha de insertarse al comienzo de la lista.

```
procedimiento insertar(E/S punt: inic, anterior E tipo_elemento:
elemento)

  var punt : auxi

inicio
  reservar(auxi)
  auxi → elemento ← (elemento)
  si anterior=nulo entonces
    auxi → sig ← (inic)
    inic ← (auxi)
  si_no
    auxi → sig ← (anterior → sig)
    anterior → sig ← (auxi)
  fin_si
  anterior → (auxi) // Opcional
fin_procedimiento
```





1° Situación de partida

2° Reservar(auxi)

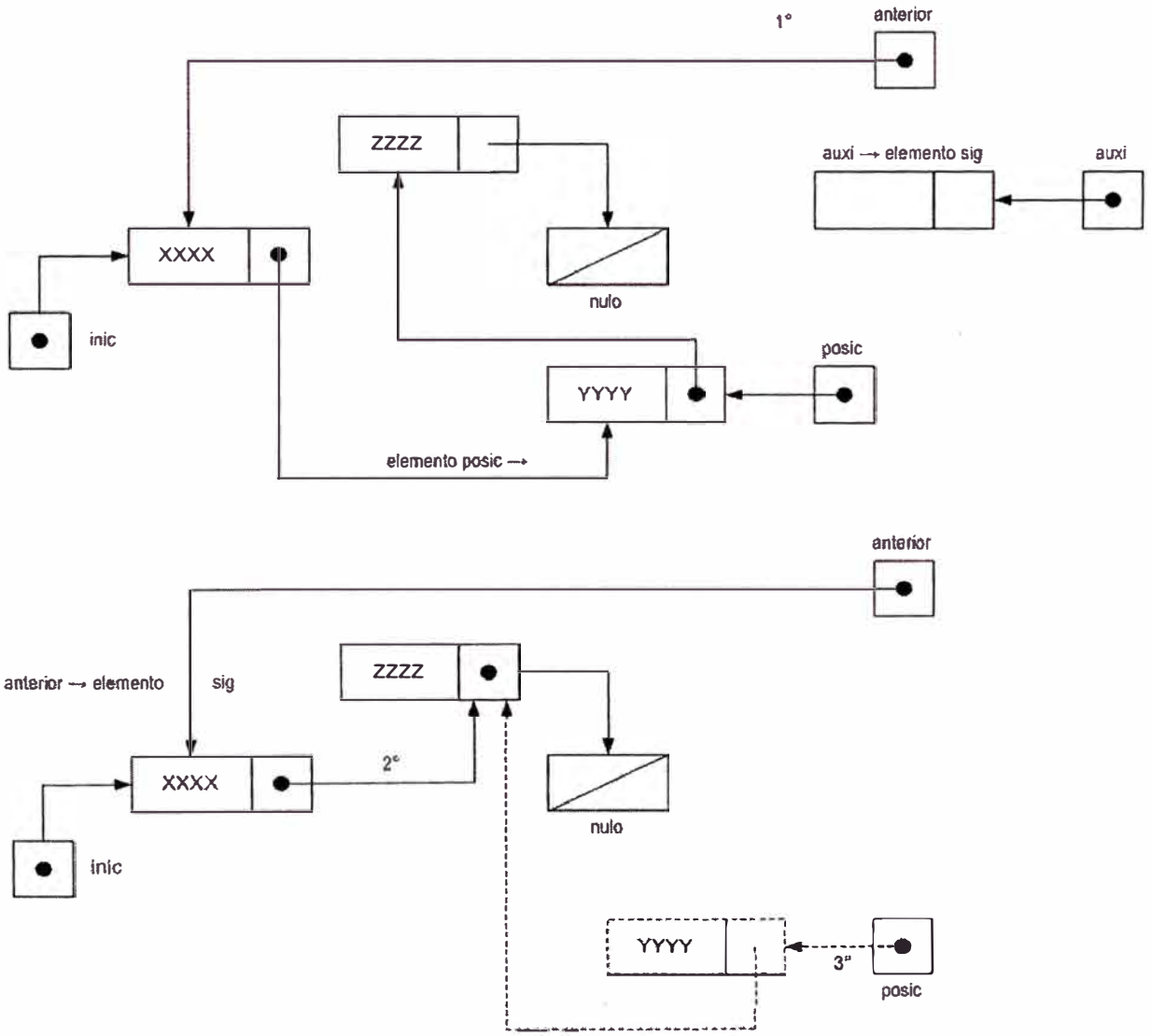
3° Introducir la nueva información en `auxi → elemento`

4° Hacer que `auxi → sig` apunte a donde lo hacía `anterior → sig`

5° Conseguir que `anterior → sig` apunte a donde lo hacía `auxi`

Antes de proceder a la eliminación de un elemento de la lista, deberemos comprobar que no está vacía. Al suprimir un elemento de la lista consideraremos dos casos particulares:

- El elemento a suprimir está al principio de la lista.
- El elemento se encuentra en cualquier otro lugar de la lista.



1° Situación de partida

2° `anterior → sig` apunta a donde `posic → sig`

3° liberar(posic)

```
procedimiento suprimir(E/S punt: inic, anterior, posic)
inicio
  si anterior=nulo entonces
    inic ← (posic → sig)
  si_no
    anterior → sig ← (posic → sig)
  fin_si
  liberar(posic)
  anterior ← (nulo)  // Opcional
  posic ← (inic)     // Opcional
fin_procedimiento
```

ANEXO E
ESPECIFICACIONES DEL LENGUAJE ALGORITMICO UPSAM

APÉNDICE A

ESPECIFICACIONES DEL LENGUAJE ALGORÍTMICO UPSAM[1]

A.1. ELEMENTOS DEL LENGUAJE

Identificadores

- Longitud: máximo 32 caracteres.
- Se admiten todos los caracteres alfabéticos (estándar y extendidos), dígitos y el símbolo de subrayado.
- No se hace distinción entre mayúsculas y minúsculas.
- Se debe empezar siempre por un carácter alfabético.

Comentarios

- Se recomienda el uso de la doble barra inclinada (//) al comienzo del comentario. Puede ir al principio de una línea del pseudocódigo, o después de alguna instrucción.
- Para comentarios multilínea, se puede encerrar el comentario entre llaves ({}).

Tipos de datos estándar

- Numérico (`entero` y `real`).
- Lógicos (`logico`).

[1] Las especificaciones del pseudolenguaje UPSAM (su denominación original fue UPS) han sido experimentadas desde el año 1990, en el Departamento de Lenguajes y Sistemas Informáticos de la Facultad y Escuela Universitaria de Informática de la Universidad Pontificia de Salamanca en Madrid y, con anterioridad, desde 1986 en el CESIES de la misma Universidad.

Las primeras especificaciones del pseudocódigo se publicaron en 1986 en nuestro libro *Metodología de la Programación*, editado por McGraw-Hill. El resultado que ahora se publica es fruto de todos los profesores que han impartido la asignatura de *Fundamentos de Programación*, y esta síntesis ha sido redactada por los profesores Luis Rodríguez, Matilde Fernández y Luis Joyanes, con las sugerencias y mejoras de los profesores Luis Villar y Joaquín Abeger. Han colaborado también los profesores Antonio Muñoz, Ángel Hermoso, Isabel Torralvo, Ángela Carrasco, Miguel Sánchez, Víctor Martín, M.ª del Mar García, Rafael Ojeda, Lucas Sánchez y Joaquín Abeger.

- Caracteres (`caracter`).
- Cadenas (`cadena`). Se considera como un dato estándar, pero estructurado, es decir, como un array de caracteres acabado en un carácter nulo.
- Se admiten cadenas abiertas o limitadas a un número determinado de caracteres, de la forma `cadena nombre_variable [n]`.

Constantes estándar

- Numéricas enteras: están compuestas por los dígito (`0..9`), y el signo (`+,-`).
- Reales en coma fija: están compuestas por los dígitos (`0..9`), el signo (`+.-`) y la coma decimal (`,`).
- Reales en coma flotante: están compuestas por los dígitos (`0..9`), el signo (`+,-`) y la coma decimal (`,`) de la mantisa, la letra **E**, antes del exponente y los dígitos (`0..9`), el signo (`+,-`) y la coma decimal (`,`) del exponente.
- Lógicos: son las constantes que representan a verdadero (`Verdad`) y falso (`Falso`).
- Caracteres: cualquier carácter del juego de caracteres utilizado, delimitado por el separador de caracteres (`'`).

Operadores

Tabla A.1. Operadores aritméticos

-	Menos unario
*	Multiplicación
/	División real
↑	Exponenciación
+	Adición
-	Resta
mod	Módulo de la división entera
div	División entera

Tipos de resultados de las expresiones aritméticas:

La división real siempre da un resultado real. Los operadores mod y `div` sólo operan con enteros y el resultado es entero. El menos unario devuelve el mismo tipo que el operando. En el resto de operadores aritméticos, si alguno de los operadores es real, el resultado es real.

Tabla A.2. Operadores de relación

=	Igual a
<>	Distinto de
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que

Tipos de resultados de expresiones de relación:

Siempre devuelven un dato de tipo lógico. Los operandos deben ser del mismo tipo.

Tabla A.3. Operadores lógicos

no	Negación lógica
y	Multiplicación lógica
o	Suma lógica
o_x	O exclusivo
eqv	Equivalencia
imp	Implicación

Tipos de resultados de expresiones lógicas:

Siempre devuelven un dato de tipo lógico. Los operandos deben ser del mismo tipo lógico.

Tabla A.4. Prioridad de los operandos

Paréntesis	()
Exponenciación	↑
Operadores unarios	no, -
Operadores multiplicativos	*,/, div, mod, y, ↑
Operadores auditivos	+,-,o,o_x
Operadores de relación	=,<, >, < >, >=, <=

Las operaciones se evalúan siempre de izquierda a derecha.

Operador de asignación: (←). Sólo se pueden asignar datos del mismo tipo.

Definición de tipos

- Registros:

```
registro: <nombre_del_tipo>
    <tipo>: <nombre_del_campo>
    ....
fin_registro
```

  Una línea por campo.

- Archivos de organización secuencial

```
archivo_s de <tipo_de_dato> : <nombre_del_tipo>
```

- Archivos de organización relativa

```
archivo_d de <tipo_de_dato> : <nombre_del_tipo>
```

- Arrays

```
array[dimensiones] de <tipo_de_dato>: <nombre_del_tipo>
```

dimensiones *sería un subrango.*

- Conjunto

```
conjunto de <tipo_de_dato> : <nombre_del_tipo> [es (<lista de valores>)]
```

- Enumerados

```
<nombre_del_tipo> es (<lista de valores>)
```

- Subrangos

```
subrango de <tipo_de_dato> : <nombre_del_tipo> es [<inf>..<sup>]
```

Definición de constantes

- `const <nombre_de_constante> = <expresion>`

Definición de variables

- `<tipo_de_dato> : <lista_de_variables>`

Funciones aritméticas de biblioteca

`abs()`	*valor absoluto de un número*
`ent()`	*valor entero (se redondea al entero menor)*
`trunc()`	*valor entero (se truncan los decimales)*
`redondeo()`	*redondea un número*
`aleatorio()`	*valor aleatorio*
`exp()`	e^x
`ln()`	*logaritmo neperiano (base e)*
`log10()`	*logaritmo decimal (base 10)*
`raiz2()`	*raíz cuadrada*
`cuadrado()`	*cuadrado de*
`sen()`	*seno*
`cos()`	*coseno*
`arctan()`	*arco tangente*
`tipo()`	*convierte la expresión en el tipo base especificado*

Funciones de cadena de biblioteca

`longitud()`	*Longitud de la cadena*
`posicion(<c>, <cb>)`	*Posición de* `cb` *dentro de* `c`
`subcadena(<c>, <ini>, [<long>])`	*Extrae de* `c`*, a partir del carácter* `ini` *una subcadena de longitud* `long`

Funciones de conversión de número a cadena

`codigo()`	*Devuelve el código ASCII de un carácter*
`car()`	*Devuelve el carácter asociado a un código ASCII*
`valor()`	*Convierte una cadena en un valor numérico*
`cad()`	*Convierte un número en cadena*

Funciones de información

`tamaño_de()`	*Devuelve la longitud en bytes de una variable o tipo de dato*

Estructura de un programa

```
algoritmo <nombre_algoritmo>
[incluir <modulo>]…
// declaraciones de constantes, tipos y variables
[const <declaraciones de constante]
[tipo <declaraciones de tipos de datos>]
[var <declaraciones de variables>]
// Se recomienda declarar primero las constantes,
// luego los tipos y por ultimo las variables
// Cualquier objeto debe ser declarado antes de ser utilizado
inicio
...
fin
[declaraciones de procedimientos y funciones]
```

A.2. ESTRUCTURAS DE CONTROL

Estructuras selectivas

```
si condicion entonces
   <acciones>
[si_no
   <acciones>]
fin_si
segun_sea <expresion_ordinal> hacer
   <lista_de_valores_ordinales> : <acciones>
   ...
```

```
[si_no
   <acciones>]
fin_segun
```

- La expresión que se evalúa debe ser de tipo ordinal.
- Las listas de valores, son uno o más valores ordinales separados por comas.

Estructuras repetitivas

```
mientras <expresion_logica> hacer
   <acciones>
fin_mientras

repetir
   <acciones>
hasta_que <expresion_logica>

desde <variable_ordinal> ← <valor_final>
                         incremento|decremento <valor> hacer
   <acciones>
fin_desde

iterar
   <acciones>
   [salir_si]
fin_iterar
```

A.3. PROGRAMACIÓN MODULAR

A.3.1. Inclusión de archivos o modulos

- `incluir <modulo>`, permite incluir archivos con subprogramas.

A.3.2. Procedimientos

- Declaración

```
procedimiento <nombre_proc>[lista_de_parametros_formales)]
[declaraciones locales]
inicio
...
fin_procedimiento
```

`<lista_de_argumentos>` *es uno o más argumentos de la siguiente forma:*

`[(E|S|E/S)`[1] `<tipo_de_dato> : <lista_de_argumentos>]...`

[1] E: Entrada; S: Salida; E/S: Entrada/Salida.

- El tipo de dato debe estar definido por anticipado.
- Llamada a un procedimiento:

```
[llamar_a] <nom_proc>[lista_parametros_actuales)]
```

- Los parámetros deben coincidir en posición, tipo y número.

A.3.3. Funciones

- Declaración

```
<tipo_de_dato> funcion
<nombre_fun>[(lista_de_parametros_formales)]
[declaraciones locales]
inicio
  ...
  devolver (<expresion>)
fin_funcion
```

- Lista de parámetros es igual que en los procedimientos.
- `devolver` nos dice el valor de retorno de la función.
- Sólo se pueden devolver datos estándar.

A.3.4. Cuestiones generales

- El ámbito de las variables es igual que en Pascal.
- Todas las variables locales, para el uso de variables globales utilizar el modificador `global`, `global entero a`.
- No se admiten subprogramas anidados.
- Los archivos se pasan siempre como E/S.

A.3.5. Procedimientos de entrada/salida

- `leer (<lista_de_variables>)`
- `escribir ([<lista_de_expresiones>])`
- `leercar (<variable_tipo_caracter>)`

A.4. ARCHIVOS

A.4.1. Archivos secuenciales (texto)

- Crear y abrir es distinto. Para abrir es necesario tener creado el archivo (esto también es válido para indexados y directos). Crear inicializa el archivo; abrir, permite acceder a un archivo existente.
- `crear (<var_tipo_archivo>, <nombre_fisico>)`, nombre físico es una expresión de cadena.
- `abrir (<var_tipo_archivo>, <modo>, <nombre_fisico>)`, nombre físico es una expresión de cadena.
- <modo> en archivos secuenciales puede ser: `lectura` (coloca el puntero de datos al inicio

del archivo y abre para la lectura) o **escritura** (coloca el puntero de datos al final del archivo y abre para escritura).

- Escritura: **escribir** *(<var_tipo_archivo>, <lista_de_expresiones>)*.
- Lectura: **leer** *(<var_tipo_archivo>, <lista_de_variables>)*.
- Lectura: **leercar** *(<var_tipo_archivo>, <var_tipo_caracter>)*.
- Fin de línea: **fdl** [*(var_tipo_archivo)*]
- Fin de archivo: **fda** [*(var_tipo_archivo)*]. Es verdadero si se ha intentado la lectura después del último registro.
- Cerrar un archivo **cerrar** *(<lista var_tipo archivo>)*.
- Otros procedimientos útiles pueden ser **borrar** *(<nombre_fisico>)*, que borra un archivo que debe estar cerrado, y **renombrar** *(<nombre fisico>, <nombre_fisico>)*, que renombra el archivo primero por el nombre del segundo. Ambos deben estar cerrados.
- Otra función útil puede ser **lda** *(<nombre_fisico>)*, que devuelve la longitud del archivo en bytes.

A.4.2. Archivos directos (relativos)

- **crear** *(<var_tipo_archivo>, <nombre_fisico>)*, nombre físico es una expresión de cadena.
- **abrir** *(<var_tipo_archivo>, <modo>, <nombre_fisico>)*, nombre físico es una expresión de cadena.
- <modo> en archivos directos sólo es **l_e** para lectura y escritura.
- Escritura: **escribir** *(<var_tipo_archivo>, <var_tipo_base>, <posicion>)*. Escribe una variable del tipo base del archivo en la posición relativa especificada.
- Lectura: **leer** *(<var_tipo_archivo>, <var_tipo_base>, <posicion>)*. Lee una variable del tipo base del archivo en la posición relativa especificada.
- Fin de línea y Fin de archivo, no tienen efecto en archivos relativos (*si consideramos la lectura secuencia, Fin de archivo sí tendría sentido).*
- También se admiten los procedimientos **borrar** y **renombrar**, así como la función **lda**.
- El número de registros del archivo se puede obtener con

```
lda(<var_tipo_archivo>) / tamaño_de(var_tipo_base_del_archivo).
```

A.4.3. Archivos indexados

- Definición:

```
archivo_i de <tipo_dato>: <nombre_del_tipo>
             clave_p <lista_de_campos>//Clave primaria sin duplicados
             [clave_s <lista_de_campos> [duplicada]]...
```

- **crear** *(<var_tipo_archivo>, <nombre_fisico>)*, nombre físico es una expresión de cadena.
- **abrir** *(<var_tipo_archivo>, <modo>, <nombre_fisico>)*, nombre físico es una expresión de cadena.
- <modo> en archivos indexados sólo puede ser **l_e**.
- Escritura: **escribir** (<var_tipo_archivo>, <var_tipo_base>). Escribe el registro en la posición que tenga en ese momento la clave primaria. Da error si existe, o si tiene claves secundarias sin duplicados.

- Lectura directa: **leer** *(<var_tipo_archivo>, <var_tipo_base>, [<campo clave]]*. Por omisión lee el registro cuyo valor coincida con el que en ese momento tenga la clave primaria. Para leer por claves alternativas se incluirá el argumento <campo_clave>.
- Una función **Existe** *(<var_tipo_archivo>)* devolvería verdadero si la lectura ha sido correcta.
- Lectura secuencial: **leersec** *(<var_tipo_archivo>, <var_tipo_base>, [<campo clave]]*. Lectura secuencial del siguiente registo de la clave especificada. Por omisión, lee la siguiente clave secuencial.
- Fin de archivo: **fda** *[(var_tipo_archivo)]*. Es verdadero si se ha intentado la lectura después del último registro.
- Otros procedimientos útiles pueden ser: **borrar** *(<nombre_fisico>)*, que borra un archivo que debe estar cerrado, y **renombrar** *(<nombre fisico>, <nombre_fisico>)*, que renombra el archivo primero por el nombre del segundo. Ambos deben estar cerrados.
- Otra función útil puede ser **ida** *(<nombre_fisico>)*, que devuelve la longitud del archivo en bytes.

A.5. VARIABLES DINÁMICAS

- Declaración de tipos dinámicos

```
puntero_a <tipo_de_dato> : <nombre_del_tipo>
```

- Acceso al contenido de una variable

```
<var_tipo_ptr> ->
```

- Asignar memoria a una variable de tipo puntero: **reservar** (<var_tipo_ptr>).
- Liberar memoria de una variable de tipo puntero: **liberar**(<var_tipo_ptr>).
- Puntero nulo: **nulo**.

A.6. PALABRAS RESERVADAS, OPERADORES, CARACTERES ESPECIALES, FUNCIONES Y PROCEDIMIENTOS ESTÁNDAR

Tabla A.5. Palabras reservadas, símbolos y operadores

Carácter / Palabra reservada	Significado / Traducción
'	delimitador de caracteres
→	variable apuntada
←	asignación
*	multiplicación
+	suma, concatenación
+	signo positivo
-	signo negativo
, (coma decimal)	
-	menos unario
-	resta

Tabla A.5. Palabras reservadas, símbolos y operadores.*(Cont.)*

Carácter / Palabra reservada	Significado / Traducción
/	división real
//	inicio de comentario
:	dos puntos
<	menor que
< =	menor o igual que
< >	distinto de
=	igual a
>	mayor que
> =	mayor o igual a
^	exponenciación
(	inicio de comentario
)	fin de comentario
&	concatenación
abrir	*open*
abs()	*abs*
aleatorio()	*random*
algoritmo	*program*
arctan()	*atn*
archivo_d	definición de archivos directos
archivo_i	definición de indexados
archivo_s	definición de archivos secuenciales
array	*array*
borrar	*delete*
cad()	*str*
cadena	*string*
car()	*chr*
caracter	*char*
clave_p	clave primaria de archivos indexados
clave_s	clave secundaria de archivos indexado
codigo()	*asc*
conjunto	*set*
const	*const*
cos()	*cos*
crear()	inicializa un archivo
cuadrado()	*sqr*
de	*of*
decremento	decremento para bucles desde
desde	*for*
devolver	*return*
div	*div*
duplicada	se admiten duplicados en claves secundarias de archivos indexados
E	inicio del exponente en un real de coma flotante
e	modo de apertura de archivos para escritura
E	parámetro de entrada
E/S	parámetro de entrada/salida

Tabla A.5. Palabras reservadas, símbolos y operadores.*(Cont.)*

Carácter / Palabra reservada	Significado / Traducción
ent()	*int*
entero	*integer*
entonces	*then*
eqv	*eqv*
es	*is*
escribir	*write*
existe()	indica en archivos indexados si una operación de lectura ha sido correcta
exp()	*exp*
Falso	*false*
fda	*eof*
fdl	*eoln*
fin	*end*
fin_desde	*next, end for*
fin_funcion	*end function*
fin_iterar	*loop*
fin_mientras	*end while*
fin_procedimiento	*end sub*
fin_segun	*end select*
fin_si	*end if*
funcion	*function*
hacer	*do*
hasta	*to*
hasta_que	*until*
imp	*imp*
incluir	*include*
incremento	para bucles desde
inicio	*begin*
iterar	*do*
l/d	*I-O*
lda	*lof*
l	*input*
leer	*read*
leercar	*readkey, input$*
leersec	lectura secuencial en archivos indexados
liberar	*dispose*
ln()	*log*
log10()	*log10*
logico	*boolean*
longitud()	*len, length*
llamar_a	*call*
mientras	*while*
mod	*mod*
no	*not*
nulo	*nil*
o	*or*
o_x	*xor*

Tabla A.5. Palabras reservadas, símbolos y operadores.*(Cont.)*

Carácter / Palabra reservada	Significado / Traducción
posicion (<c>, <cb>)	*pos, index, instr*
procedimiento	*procedure*
puntero_a	declaración de tipos de puntero
raiz2	*sqrt*
real	*real*
redondeo()	*round()*
registro	*record, struct*
renombrar	*rename*
repetir	*repeat*
reservar	*new*
S	parámetro de salida
salir	*exit*
segun_sea	*case of*
sen ()	*sin*
si	*if*
si_no	*else*
subcadena (<c>, <ini>, [lond>])	*mid, copy*
subrango	declaración de subrangos
Tamaño_de	*sizeOf*
tipo	*type*
trunc ()	*trunc*
var	*var*
valor ()	*val*
verdad	*true*
Y	*and*

ANEXO F

GUIA DE REFERENCIA LENGUAJE C ANSI

APÉNDICE **D**

GUÍA DE REFERENCIA LENGUAJE C ANSI

D.1. ELEMENTOS BÁSICOS DE UN PROGRAMA

El lenguaje C fue desarrollado en *Bell Laboratories* para su uso en investigación, y se caracteriza por un gran número de propiedades que lo hacen ideal para usos científicos y de gestión.

Una de las grandes ventajas del lenguaje C es ser *estructurado*. Se pueden escribir bucles que tienen condiciones de entrada y salida claras y se pueden escribir funciones cuyos argumentos se verifican siempre para su completa exactitud.

Su excelente biblioteca estándar de funciones convierten a C en uno de los mejores lenguajes de programación que los profesionales informáticos pueden utilizar.

D.2. ESTRUCTURA DE UN PROGRAMA C

Un programa típico en C se organiza en uno o más *archivos fuentes* o *módulos*. Cada archivo tiene una estructura similar con comentarios, directivas de preprocesador, declaraciones de variables y funciones y sus definiciones. Normalmente se sitúa cada grupo de funciones y variables relacionadas en un único archivo fuente. Dentro de cada archivo fuente, los componentes de un programa suelen colocarse en un determinado modo estándar. La Figura D.1 muestra la organización típica de un archivo fuente en C.

Los componentes típicos de un archivo fuente de programa son:

1. El archivo comienza con algunos comentarios que describen el propósito del módulo e información adicional, tal como el nombre del autor y fecha, nombre del archivo. Los comentarios comienzan con `/*` y terminan con `*/`.
2. Órdenes al preprocesador, conocidas como *directivas del preprocesador*. Normalmente incluyen archivos de cabecera y definición de constantes.
3. Declaraciones de variables y funciones son visibles en todo el archivo. En otras palabras, los nombres de estas variables y funciones se pueden utilizar en cualquiera de las funciones de este archivo. Si se desea limitar la visibilidad de las variables y funciones *sólo* a ese módulo,

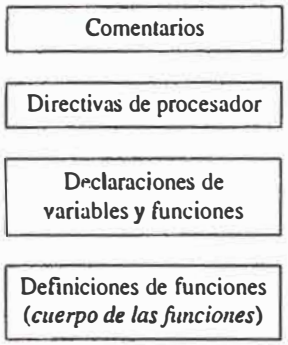


Figura D.1. Organización de un programa C.

ha de poner delante de sus nombres el prefijo `static`; por el contrario, la palabra reservada `extern` indica que los elementos se declaran y definen en otro archivo.

4. El resto del archivo incluye definiciones de las funciones (su cuerpo). Dentro de un cuerpo de una función se pueden definir variables que son locales a la función y que sólo existen en el código de la función que se está ejecutando.

D.3. EL PRIMER PROGRAMA C ANSI

```
#include <studio.h>

main ()
{
   printf ("¿Hola mundo!");
   return 0;
}
```

D.4. PALABRAS RESERVADAS ANSI C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Las palabras reservadas `const`, `enum`, `void` y `volatile` son nuevas en C ANSI.

D.5. DIRECTIVAS DEL PREPROCESADOR

El *preprocesador* es la parte del compilador que realiza la primera etapa de traducción o compilación de un archivo C ANSI en instrucciones de máquina. El preprocesador procesa el archivo fuente y

actúa sobre las órdenes, denominadas *directivas de preprocesador*, incluidas en el programa. Estas directivas comienzan con el signo de libra # (almohadilla). Normalmente, el compilador invoca automáticamente al preprocesador antes de comenzar la compilación. Se puede utilizar el preprocesador de tres formas distintas para hacer sus programas más modulares, más legibles y más fáciles de personalizar:

1. Mediante la directiva `#include` para insertar el contenido de un archivo en su programa.
2. Mediante la directiva `#define`, se pueden definir macros que permiten reemplazar una cadena por otra. Se puede utilizar la directiva `#define` para dar nombres significativos a constantes numéricas, mejorando la legibilidad de sus archivos fuente.
3. Con directivas tales como `#if`, `#ifdef`, `#else` y `#endif`, pueden compilar sólo partes de su programa. Se puede utilizar esta característica para escribir archivos fuente con código para dos o más sistemas, pero compilar sólo aquellas partes que se aplican al sistema informático en que se compila el programa.

D.6. ARCHIVOS DE CABECERA

Directivas tales como `#include <stdio.h>` indican al compilador que lea el archivo `studio.h` de modo que sus líneas se sitúan en la posición de la directiva. C ANSI soporta dos formatos para la directiva `#include`:

```
1. #include <stdio>
2. #include "demo.h"
```

El primer formato de `#include`, lee el contenido de un archivo (el archivo estándar de C, `stdio.h`). El segundo formato visualiza el nombre del archivo encerrado entre las dobles comillas que está en el directorio actual.

D.7. DEFINICIÓN DE MACROS

Una macro define un símbolo equivalente a una parte de código C y se utiliza para ello la directiva `#define`. Por ejemplo, se puede representar constantes tales como `PI`, `IVA` y `BUFFER`.

```
#define PI 3.14159
#define IVA 16
#define BUFFER 1024
```

que toman los valores `3.14159`, `16` y `1024`, respectivamente. Una macro también puede aceptar un parámetro y reemplazar cada ocurrencia de ese parámetro con el valor proporcionado cuando la macro se utiliza en un programa. Por consiguiente, el código que resulta de la expansión de una macro puede cambiar dependiendo del parámetro que se utilice cuando se ejecuta la macro. El código que resulta de la expansión de una macro puede cambiar dependiendo del parámetro que se utilice cuando se ejecuta la macro. Por ejemplo, la macro siguiente acepta un parámetro y expande a una expresión diseñada para calcular el cuadrado del parámetro:

```
#define cuadrado (x) ((x) * (x))
```

D.8. COMENTARIOS

El compilador ignora los comentarios encerrados entre los símbolos **/*** y ***/**.

```
/*Mi primer prog rama*/
```

Se pueden escribir comentarios multilínea:

```
/* Mi seg undoprog ramaC
escrito el dia 15 de Agos tde 1985 durante las fiestas
de Carchelejo - Jaen - España */
```

Los comentarios no pueden anidarse. La línea siguiente no es legal:

```
/*Comentario /* com entariointerno */ externo */
```

D.9. TIPOS DE DATOS

Los tipos de datos básicos incorporados a C son *enteros, reales* y *carácter.*

Tabla D.1. Tipos de datos enteros

Tipo de dato	Tamaño en bytes	Tamaño en bits	Valor mínimo	Valor máximo
signed char	1	8	-128	127
unsigned char	1	8	0	255
signed short	2	16	-32.768	32.767
unsigned short	2	16	0	65.535
signed int	2	16	-32.768	32.767
unsigned int	2	16	0	65.535
signed long	4	32	-2.147.483.648	2.147.483.647
unsigned long	4	32	0	4.294.967.295

El tipo `char` se utiliza para representar caracteres o valores integrales. Las constantes de tipo `char` pueden ser caracteres encerrados entre comillas (`'A'`, `'b'`, `'p'`). Caracteres no imprimibles (tabulación, avance de página...) se pueden representar con secuencias de escape (`'\t'`, `'\f'`).

Tabla D.2. Secuencias de escape

Carácter	Significado	Código ASCII
\a	Carácter de alerta (timbre)	7
\b	Retroceso de espacio	8
\f	Avance de página	12
\h	Nueva línea	10

Carácter	Significado	Código ASCII
\r	Retorno de carro	13
\t	Tabulación (horizontal)	9
\v	Tabulación (vertical)	11
\\	Barra inclinada	92
\?	Signo de interrogación	63
\'	Comilla	39
\''	Doble comilla	34
\nnn	Número octal	—
\xnn	Número hexadecimal	—
'\0'	Carácter nulo (terminación de cadena)	—

Tabla D.3. Tipos de datos de coma flotante

Tipo de dato	Tamaño en bytes	Tamaño en bits	Valor mínimo	Valor máximo
float	4	32	3.4E-38	3.4E+38
double	8	64	1.7E-308	1.7E+308
long double	10	80	3.4E-4932	1.1E+4932

Todos los números sin un punto decimal en programas C se tratan como enteros y todos los números con un punto decimal se consideran reales de coma flotante de doble precisión. Si se desea representar números en base 16 (hexadecimal) o en base 8 (octal), se precede al número con el carácter `'OX'` para hexadecimal y `'0'` para octal. Si se desea especificar que un valor entero se almacena como un entero largo se debe seguir con una `''L''`.

```
025   /* oc tal25 o decimal 21 */
0x25  /* hexadecimal 25 o decimal 37 */
250L  /* entero largo 250 */
```

D.10. VARIABLES

Todas las variables en C se declaran o definen antes de que sean utilizadas. Una *declaración* indica el tipo de una variable. Si la declaración produce también almacenamiento (se inicia), entonces en una *definición.*

D.10.1. Nombres de variables en C

Los nombres de variables en C constan de letras, números y carácter subrayado. Pueden ser mayúsculas o minúsculas o una mezcla de tamaños. El tamaño de la letra es significativo, las variables siguientes son *todas diferentes.*

```
Temperatura TEMPERATURA temperatura
```

A veces se utilizan caracteres subrayados y mezcla de mayúsculas y minúsculas para aumentar la legibilidad:

```
Dia_de_Semana    DiaDeSemana    Nombre_ciudad
PagaMes

int x;                          //declara x variable entera
char nombre, conforme;          //declara nombre, conforme de tipo char

int x=, no = 0;                 //definen las variables x y no
flotat total = 42.125           //define la variable total
```

Se pueden declarar variables múltiples del mismo tipo en dos formas: Así una declaración

```
int v1; int v2; int v3; int v4;
```

o bien:

```
int v1;
int v2;
int v3;
int v4;
```

pudiéndose declarar también de la forma siguiente:

```
int v1, int v2, int v3, int v4;
```

C no soporta tipos de datos lógicos, pero mediante enteros se pueden representar: 0, significa *falso*; distinto de cero, significa *verdadero* (cierto).

La palabra reservada `const` permite definir determinadas variables con valores constantes, que no se pueden modificar. Así, se declara

```
const int z = 4350
```

y si se trata de modificar su valor

```
z = 3475
```

el compilador emite un mensaje de error similar a "`Cannot modify a const object in function main`" ("`No se puede modificar un objeto const en la funcion main`"). Las variables declaradas como `const` pueden recibir valores iniciales, pero no puede modificarse su valor con otras sentencias.

D.10.2. Variables tipo `char`

Las variables de tipo `char` (carácter) pueden almacenar caracteres individuales. Por ejemplo la definición

```
char car = 'M';
```

declara una variable `car` y le asigna el valor ASCII del carácter `M`. El compilador convierte la constante carácter `'M'` en un valor entero (`int`), igual al código ASCII de `'M'` que se almacena a continuación en el byte reservado para `car`.

Dado que los caracteres literales se almacenan internamente como valores `int`, se puede cambiar la línea.

```
char car;
```

por

```
int car;
```

y el programa funcionará correctamente.

D.10.3. Constantes de cadena

Las cadenas de caracteres constan de cero o más caracteres separados por dobles comillas. La cadena se almacena en memoria como una serie de valores ASCII de tipo `char` de un solo byte y se termina con un byte cero, que se llama carácter nulo

```
'Sierra Magina en Jaen
```

Además de los caracteres que son imprimibles, se pueden guardar en constantes cadena, códigos de escape, símbolos especiales que representan códigos de control y otros valores ASCII no imprimibles. Los códigos de escape se representan en la tabla como un carácter único, almacenado internamente como un valor entero y compuesto de una barra inclinada seguida por una letra, signo de puntuación o dígitos octales o hexadecimales. Por ejemplo, la declaración

```
char c = '\n';
```

asigna el símbolo nueva línea a la variable `c`. En los PC, cuando se envía un carácter `\n` a un dispositivo de salida, o cuando se escribe `\n` en un archivo de texto, el símbolo nueva línea se convierte en un retorno de carro y un avance de línea.

D.10.4. Tipos enumerados

El tipo `enum` es una *«lista ordenada»* de elementos como constantes enteras. A menos que se indique lo contrario, el primer miembro de un conjunto enumerado de valores toma el valor 0, pero se pueden especificar valores. La declaración:

```
enum diasSemana {Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
```

significa que `Lunes = 0, martes = 1`, etc. Sin embargo, si se hace `Viernes = 10`, entonces `Lunes` sigue siendo 0, `Martes` es igual a 2, etc.; pero ahora `Viernes = 11, Sabado = 12`, etc. Un tipo enumerado se puede utilizar para declarar una variable

```
enum diasSemana Laborable;
```

y a continuación utilizarla con

```
Laborable = Jueves;
```

o bien:

```
Laborable = Sabado;
if (Laborable >= Viernes)
  printf ("Hoy no es laborable /n",Laborable);
```

D.11. EXPRESIONES Y OPERADORES

Las expresiones son operaciones que realiza el programa.

```
a + b + c;
```

Tabla D.4. Operadores aritméticos

Operador	Descripción	Ejemplo
*	Multiplicación	(a * b)
/	División	(a / b)
+	Suma	(a + b)
-	Resta	(a - b)
%	Módulo	(a % b)

Tabla D.5. Operadores relacionales

Operador	Descripción	Ejemplo
<	Menor que	(a < b)
<=	Menor que o igual	(a <= b)
>	Mayor que	(a > b)
>=	Mayor o igual que	(a >= b)
==	Igual	(a = = b)
!=	No igual	(a != b)
++	Incremento en i	++i, i++
--	Decremento en i	-- j, j --

Ejemplos:

```
i = i + 1;      //Sumar uno a i
i++;            //Igual que anterior

i = i - 1:      //Resta uno a i
i--;            //Igual que anterior
```

Tabla D.6. Operadores de manipulación de bits (bitwise)

Operador	Descripción	Ejemplo
&	AND bit a bit	C = A&B;
|	OR inclusiva bit a bit	C = A | B;
^	OR exclusiva bit a bit	C = A ^ B;
<<	Desplazar bits a izquierda	C = A << B;
>>	Desplazar bits a derecha	C = A >> B;
~	Complemento a uno	C = ~B

D.11.1. Operadores de asignación

Los operadores de asignación son binarios y combinaciones de operadores y del signo = utilizado para abreviar expresiones:

```
A = B       /* asigna el valor de B a A
C = (A = B) /* C y A son iguales a B
C = A = B   /* asigna B a A y a C
```

`A = A + 45;` *equivale a* `A += 45;`

El compilador puede generar código más eficiente, recurriendo a operadores de asignación compuestos del tipo `*=`, `+=`, etc.; cada operador compuesto (*op*) reduce la expresión en pseudocódigo:

```
O = a op b
```

a la forma abreviada

```
a op = b;
```

Tabla D.7. Operadores de asignación

Operador	Descripción y ejemplo		
=	Operación de asignación simple		a = b;
*=	z *= 10;	*equivale a*	z = z * 10;
/=	z /= 5;	*equivale a*	z = z / 5;
%=	z %= 2;	*equivale a*	z = z % 2;
+=	z += 4;	*equivale a*	z = z + 4;
-=	z -= 5;	*equivale a*	z = z - 5;
<<=	z <<= 3;	*equivale a*	z = z << 3;
>>=	z >>= 4;	*equivale a*	z = z > 3;
&=	z &= j;	*equivale a*	z = z $ j;
^=	z ^= j;	*equivale a*	z = z ^ j;
|=	z |= j;	*equivale a*	z = z | j;

D.11.2. Operador serie

El operador en serie, la coma, indica una serie de sentencias ejecutadas de izquierda a derecha. Se utilizan normalmente en bucles `for`. Por ejemplo:

```
for (cuenta=1; cuenta<100; ++cuenta, ++lineasporpagina);
```

produce el incremento de la variable cuenta y de la variable `lineasporpagina` cada vez que se ejecuta el bucle (se realiza una iteración).

D.11.3. Prioridad (precedencia) de operadores

Las expresiones C constan de diversos operandos y operadores. En expresiones complejas, las subexpresiones con operadores de prioridad (precedencia) más alta se evalúan antes que las subexpresiones con operadores de menor prioridad.

Tabla D.8. Orden de evaluación y prioridad de operadores (asociatividad)

Nivel	Operadores	Orden de evaluación
1	( ) . [ ] →	izquierda-derecha
2	* & ! ~ ++ -- + - (conversion de tipo sizeof)	derecha-izquierda
3	* / %	izquierda-derecha
4	+ -	izquierda-derecha
5	<< >>	izquierda-derecha
6	< <= > >=	izquierda-derecha
7	== !=	izquierda-derecha
8	&	izquierda-derecha
9	^	izquierda-derecha
10	|	izquierda-derecha
11	&&	izquierda-derecha
12	||	izquierda-derecha
13	? :	derecha-izquierda
14	= *= /= += -= %= <<= >>= &= ^= |=	derecha-izquierda
15	,	izquierda-derecha

D.12. FUNCIONES DE ENTRADA Y SALIDA

Las funciones `printf()` y `scanf()` permiten comunicarse con un programa y se denominan funciones de E/S. `printf()` es una *función de salida* (S) y `scanf()` es una *función de entrada* y ambas utilizan una cadena de control y una lista de argumentos.

D.12.1. `printf`

La función escribe en el dispositivo de salida, los argumentos incluidos en la lista de argumentos. Requiere el archivo de cabecera `stdio.h` La salida de `printf` se realiza con formato, y el mismo consta de una cadena de control y una lista de variables.

```
printf (cadena de control [, item1, item2,…item]);
```

El primer argumento es la *cadena de control* (o formato propiamente dicho) y determina el formato de escritura de los datos. Los argumentos restantes son los datos o variables de datos a escribir

```
printf ("Esto es una prueba %d\n", prueba);
```


La cadena de control tiene tres componentes: texto, identificadores y secuencias de escape. Se puede utilizar cualquier texto, identificadores y secuencias de escape. El número de identificadores ha de corresponder con el número de variables o valores a escribir.

Los identificadores de la cadena de formato determinan cómo se escriben cada uno de los argumentos:

```
printf ("Mi pueblo favorito es Cazorla%s", msg);
```

Cada identificador comienza con un signo porcentaje (%) y un código que indica el formato de salida de la variable.

Tabla D.9. Códigos de identificadores

Identificador	Formato
%d	Entero decimal
%c	Carácter simple
%s	Cadena de caracteres
%f	Coma flotante (decimal)
%e	Coma flotante (notación exponencial)
%g	Usa el %f o el %e más corto
%u	Entero decimal sin signo
%o	Entero octal sin signo
%x	Entero hexadecimal sin signo

Las secuencias de escape son las indicadas en la tabla

```
printf ("Mi flor favorita es la %s\n", msg);
printf ("La temperatura es %f grados centigrados \n", centigrados);
```

Ejemplo D.1

```
#include <stdio.h>
#define PI 3.141593
#define SIERRA "Sierra Magina"

int main (void)
{
    printf ("El valor de pi es %f.\n", PI);
    printf ("/%2s/ \n", SIERRA);
    printf ("/%22s/ \n", SIERRA);
    return 0;
}
```

La salida producida al ejecutar el programa es:

```
El valor de PI es 3.141593.
/Sierra Magina/
/              Sierra Magina/
```

D.12.2. scanf

La función `scanf()` es la función de entrada con formato. Esta función se puede utilizar para introducir números con formato de máquina, caracteres o cadena de caracteres, a un programa

```
scanf ("%f", &fahrenheit);
```

El formato general de la función `scanf()` es una cadena de formato y uno o más variables de entrada. La cadena de control consta sólo de identificadores.

Tabla D.10. Identificadores de formato de scanf

Identificador	Formato
%d	Entero decimal
%c	Carácter simple
%s	Cadena de caracteres
%f	Coma flotante
%e	Coma flotante
%u	Entero decimal sin signo
%o	Entero octal sin signo
%x	Entero hexadecimal sin signo
%h	Entero corto

Un ejemplo típico de uso de `scanf` es:

```
printf ("Introduzca ciudad y provincia :");
scanf ("%s %s", ciudad, provincia);
```

otros ejemplos son:

```
scanf ("%d", &cuenta);
scanf ("%20s", direccion)
scanf ("%d%s", &r, &c);
scanf ("%d*c%d", &x, &y);
```

D.13. SENTENCIAS DE CONTROL

Una *sentencia* consta de palabras reservadas, expresiones y otras sentencias; cada sentencia termina con su punto y coma (`;`).

Un tipo especial de sentencia, la *sentencia compuesta* o *bloque*, es un grupo de sentencias encerradas entre llaves (`{...}`). El cuerpo de una función es una sentencia compuesta. Una sentencia compuesta puede tener variables locales.

D.13.1. Sentencia if

1.
```
if (expresion)
    sentencia;
```
o bien:
```
if (expresion) sentencia;
```


2. Si la sentencia es *compuesta*

```
if (expresion) {
    sentencia1;
    sentencia2;
    ...
}
```
o bien:
```
if (expresion)
{
    sentencia1;
    sentencia2;
    ...
}
```

3.
```
if (expresion == valor)
    sentencia;
```

4.
```
if (expresion != 0)
    sentencia;
```
equivale a
```
if (expresion)
```

NOTA: `(expresion != 0)` y `(expresion)` son equivalentes, ya que cualquier valor distinto de cero representa *cierto*.

D.13.2. Sentencia if-else

1.
```
if (expresion)
    sentencia1;
else
    sentencia2;
```

2.
```
if (expresion) {
    sentencia1;
    sentencia2;
    ...
} else {
    sentencia3;
    sentencia4;
    ...
}
```
o bien:
```
if (expresion)
{
    sentencia1;
    sentencia2;
}
else
{
    sentencia3
    sentencia4;
}
```

D.13.2.1. Sentencias if anidadas

1.
```
if (expresion1)
    sentencia1;
else if (expresion2)
    sentencia2;
else
    sentencia3;
```

```
2. if (expresion1)
       sentencia1;
   else if (expresion2)
       sentencia2;
   else if (expresion3)
       sentencia3;
   else if (expresionN)
       sentenciaN;
   else        /* opcional */
       sentenciaPorOmision;    /* opcional */
```

D.13.3. Expresión condicional (?:)

Expresión condicional es una simplificación de una sentencia `if-else`. Su sintaxis es:

```
expresion1 ? expresion2 : expresion3;
```

que equivale a

```
if (expresion1)
    expresion2;
else
    expresion3;
```

Ejemplo D.2

```
if (opcion == 'S')
   premio = 1000
else
   premio = 0
```

equivale a

```
Premio=(opcion =='S')? 1000 : 0
```

D.13.4. Sentencia `switch`

La sentencia `switch` realiza una bifurcación múltiple, dependiendo del valor de una expresión

```
switch (expresion) {
  case valor1:
    sentencia1; /* se ejecuta si expresion igual a 1*/
    break;      /* salida de sentencia switch       */
  case valor2:
    sentencia2;
    break;
  case valor3:
    sentencia3;
    break;
  ...
  default:
  sentencia por omision /* se ejecuta si ningun valor coincide
                           con expresion */
  }
```


Ejemplo D.3

```
switch (op)
{
    case 'a':
       func1();
       break;
    case 'b':
       func2 ();
       break;
    case 'H':
       printf ("Hola \n");
    default:
       printf ("Salida \n");
}
```

D.13.5. Sentencia `while`

```
while (expresion)
  sentencia;
```

o bien:

```
while (expresion) {
  sentencia1;
  sentencia2;
  ...
}
```

D.13.6. Sentencia `do-while`

```
do {
  sentencia;
  ...
} while (expresion);
```

o bien:

```
do {
  sentencia1;
  sentencia2;
  ...
} while (expresion)
```

Las sentencias `do-while` monosentencias se pueden escribir también así:

```
do sentencia; while  (expresion);
```

D.13.7. Sentencia `for`

```
1. for (expresion1; expresion2; expresion3) {
   sentencia;
   }

2. for (expresion1; expresion2; expresion3) sentencia;
```

La sentencia `for` equivale a

```
expresion1;
while (expresion2) {
   sentencia;
   expresion3;
}
```

Ejemplo D.4

a)
```
for (i = 1; i <= 100; i++)
    printf ("i == %d\n", i);
```

b)
```
for (i = 0, suma = 0; i <= 100; suma + = i, i++);
```

sentencia nula

D.13.8. Sentencia nula

La *sentencia nula* representada por punto y coma no hace nada. Se utilizan sentencias nulas en bucles, cuando todo el proceso se hace en las expresiones del bucle en lugar del cuerpo. Por ejemplo, localizar el byte cero que marca el final de una cadena:

```
char cad[80] = "Prueba";
int i;

for (i = o; cad [i] != '\0'; i++)
    ;   /* sentencia nula */
```

D.13.9. Sentencia `break`

La sentencia `break` se utiliza para salir incondicionalmente de un bucle `for`, `while`, `do-while` o de una sección `case` de una sentencia `switch`.

```
for (;;) {
...
  if (expresion)
  break;
}
```

D.13.10. Sentencia `continue`

La sentencia `continue` salta en el interior del bucle hasta el principio del bucle para proseguir con la *ejecución*, dejando sin ejecutar las líneas restantes después de la sentencia `continue` hasta el final del bucle; `continue` es similar a la sentencia `break`.

```
while (expresion1) {
  ...
  if (expresion2)
    continue;
  ...
}
```


D.13.11. Sentencia `goto`

Una sentencia `goto` con una etiqueta dirige el programa a una sentencia específica que contiene dicha etiqueta. Las etiquetas deben terminar con un símbolo de dos puntos.

```
salto:
  ...
  if (expresion) goto salto;
```

D.14. FUNCIONES

Las funciones son los bloques de construcción de programas C. Una *función* es una colección de declaraciones y sentencias. Cada programa C tiene al menos una función: la función `main`; ésta es la función donde comienza la ejecución de un programa C. La biblioteca C ANSI contiene gran cantidad de funciones estándar.

D.14.1. Sentencia `return`

La sentencia `return` detiene la ejecución de la función actual y devuelve al control a la función llamadora. La sintaxis es:

```
return expresion
```

en donde el valor de *expresion* se devuelve como valor de la función.

```
#include <stdio.h>

main ()
{
   printf ("Sierra de Cazorla y \n");
   printf ("Sierra Magina \n")N
   printf ("son dos bonitas sierras andaluzas \n");
   return 0;
}
```

D.14.2. Prototipos de funciones

En C ANSI se debe declarar una función antes de utilizarla. La declaración de la función indica al compilador el tipo de valor que devuelve la función y el número y tipos de argumentos que acepta.

El *prototipo de una función* es el nombre de la función, la lista de sus argumentos y el tipo de dato que devuelve.

```
int SeleccionarMenu(void);
double Area(int x, int y);
void salir(int estado);
```

D.14.3. El tipo void

C ANSI ha incorporado el tipo de dato `void`, para declarar funciones que no devuelven *nada* ni aceptan parámetros, así como para describir punteros que pueden apuntar a cualquier tipo de dato.

```
void exit (int estado);
void CuentaArriba (void);
void CuentaAbajo (void);
```

Errores típicos de funciones

1. *Ningún retorno de valor.* La función carece de una sentencia `return`. Si una función termina sin ejecutar `return`, devuelve un valor impredecible, que puede producir errores serios.
2. *Retornos omitidos.* Si hay funciones que tienen sentencias `if`, hay que asegurarse de que existe una sentencia `return` por cada camino de salida posible.
3. *Ausencia de prototipos.* Las funciones que carecen de prototipos se considera que devuelven `int`, incluso aunque estén definidas para devolver valores de otro tipo. Como *regla general, declarar prototipos para todas las funciones.*
4. *Efectos laterales.* Este problema se produce normalmente por una función que cambia el valor de una o más variables globales.

D.14.4. Detener un programa con exit

Un medio para terminar un programa sin mensajes de error en el compilador es utilizar la sentencia

```
return valor;
```

donde *valor* es cualquier expresión entera.

Otro medio para detener un programa es llamar a `exit`. La ejecución de la sentencia

```
exit (0);
```

termina el programa inmediatamente y cierra todos los archivos abiertos.

D.15. ESTRUCTURAS DE DATOS

Los tipos complejos o estructuras de datos en C son: *arrays, estructuras, uniones, cadenas* y *campos de bits.*

D.15.1. Arrays

Todos los arrays en C comienzan con el índice [0].

```
int notas[25];          /* declara array de 25 elementos */
float lista[10][25];    /* declara array de 10 por 25 dimensiones */
```


El número de elementos de un array se puede determinar dividiendo el tamaño del array completo por el tamaño de uno de los elementos

```
noelementos = sizeof(lista) / sizeof(lista[0])
```

Un *array estático* se puede iniciar cuando se declara con

```
static char nombre[] ="Sierra Magina";
```

Un *array auto* se puede iniciar sólo con una expresión constante en C ANSI.

```
int listamenor [2][2] = {{25, 4}, {100, 75}};
int digitos[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, };
```

Un *array de punteros* a caracteres se puede iniciar con

```
char *colores[]= {"verde", "rojo", "amarillo", "rosa", "azul"};
```

D.16. CADENAS

Las *cadenas* son simplemente arrays de caracteres que terminan siempre con un valor nulo (`'\0'`)

```
char cadena[80];  /* declara una cadena de 80 caracteres */
char mensaje[] = "Carchelejo esta en Sierra Magina";
char frutas[][10] = {"naranja", "platano", manzana"};
```

D.17. ESTRUCTURAS

Las *estructuras* son colecciones de datos, normalmente de tipos diferentes que actúan como un todo. Puede contener tipos de datos simples (*carácter, float, array* y *enumerado*) o compuestos (estructuras, arrays o uniones)

```
struct coordenada {
  int x;
  int y;
};

struct signatura {
  char titulo[40];
  char autor[30];
  int paginas;
  int anyopubli;
};
```

Una variable de tipo estructura se puede declarar así:

```
struct coordenada punto;0
struct signatura librosinfantiles;
```

para asignar valores a los miembros de una estructura se utiliza la notación punto (`.`)

```
punto.x = 12
punto.y = 15;
```

Existe otro modo de declarar estructuras y variables de tipo estructura.

```
struct coordenada {
  int x;
  int y;
} punto;
```

Para evitar tener que escribir `struct` cada vez que se declara la estructura, se puede usar `typedef` en la declaración:

```
typedef struct coordenada {
  int x;
  int y;
} Coordenadas;
```

y, a continuación, se puede declarar la variable `Punto`:

```
Coordenadas punto;
```

y utilizar la notación punto (.)

```
punto.x =12
punto.y = 15;
```

D.18. UNIONES

Las *uniones* son casi idénticas a las estructuras en su sintaxis; proporcionan un medio de almacenar más de un tipo en una posición de memoria. Se define un tipo unión

```
union tipodemo {
  short x;
  long l;
  float f;
};
```

y se declara una variable con

```
union tipodemo demo;
```

pudiendo utilizarse mediante

```
demo.x = 345;
```

o bien mediante

```
demo.y = 324567;
```

La unión anterior se puede iniciar así:

```
union tipodemo { short x; long l; float f; } = {75};
```


D.19. CAMPOS DE BITS

Los *campos de bits* se utilizan con frecuencia para poner enteros en espacios más pequeños de los que el compilador pueda normalmente utilizar y son, por consiguiente, dependientes de la implementación. Una estructura de campos de bits especifica el número de bits que cada miembro ocupa.

Una declaración de una estructura `Persona`

```
struct persona {
  unsigned edad;    /* 0::99 */
  unsigned sexo;    /* 0 = varon, 1 = hembra */
  unsigned hijos;   /* 0 .. 15 */
};
```

y de una estructura de campos de bits

```
struct persona {
  unsigned edad: 7;   /* 0..1127 */
  unsigned sexo: 1;   /* 0 = varon, 1 = hembra */
  unsigned hijos: 4;  /* 0..15 */
}
```

D.20. PUNTEROS (APUNTADORES [1])

El *puntero* es un tipo de dato especial que contiene la dirección de otra variable. Un puntero se declara utilizando el asterisco (*) delante de un nombre de variable

```
float *longitudCuenta;    /* puntero a datos float */
char * indice;            /* puntero a datos char */
int *p;                   /* puntero a datos int */
```

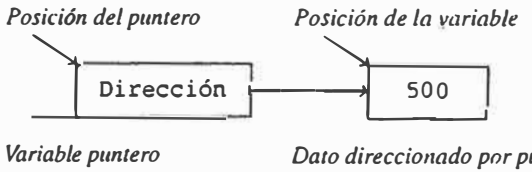


Figura D.2. Un puntero es una variable que contiene una dirección.

D.20.1. Declaración e indirección de punteros

```
int m;  /* una variable entera un */
int *p  /* un puntero p apunta a un valor entero */
p = &i; /* se asigna a p la direccion de la variable m*/
```

[1] En Latinoamérica se utiliza el término *apuntador*.

El operador de indirección (&) se utiliza para acceder al valor de la dirección contenida en un puntero.

```
float *longitudOnda;
longitudOnda = &cable1;
*longitudOnda = 40.5;
```

D.20.2. Punteros nulos y void

Un puntero *nulo* apunta a ninguna parte específica; es decir, no direcciona a ningún dato válido en memoria:

```
fp = NULL;   /* asigna Nulo a fp */
fp = 0;      /* asigna Nulo a fp */

if (fp != NULL)  /* el programa verifica si el puntero es valido */
```

Al igual que una función void que no devuelve ningún valor, un puntero void apunta a un tipo de dato no especificado

```
void *noapuntafijo;
```

D.20.3. Punteros a valores

Para utilizar un puntero que apunte a un valor, se utiliza el operador de indirección (*) delante de la variable puntero

```
double *total;
*total = 34750.75;
```

El operador de dirección (&) asigna el puntero a la dirección de la variable:

```
double *total;
double sctotal;
total = &ctotal;
```

D.20.4. Punteros y arrays

Si un programa declara

```
float *punterolista;
float listafloat [50];
```

entonces a punterolista se puede asignar la dirección del principio de listafloat (el primer elemento).

```
punterolista = listafloat;
```


o bien:

```
punterolista = &listafloat [0];

listafloat[4]    es lo mismo que    *(listafloat + 4)
listafloat[2]    es lo mismo que    *(listafloat + 2)
```

D.20.5. Punteros a punteros

Los punteros pueden apuntar a otros punteros (*doble indirección*):

```
char **lista;   /* se declara un puntero a otro puntero char */
```

equivale a

```
char *lista[];
```

y

```
char ****ptr   /* un puntero a un puntero a char */
```

D.20.6. Punteros a funciones

La declaración

```
int (*ptrafuncion ((void);
```

crea un puntero a una función que devuelve un entero, y la declaración

```
float (*ptrafuncion1) (int x, int y);
```

declara ptrafuncion1 como un puntero a una función que devuelve un valor float y requiere dos argumentos enteros. Otros ejemplos:

```
float *ptr1;         /* ptr apunta a un valor float */
float *mifun(void); /* funcion mifun devuelve un puntero a un valor float */
```

ANEXO G

GUIA DE SINTAXIS DEL LENGUAJE C++ (ESTÁNDAR C++ ANSI)

APÉNDICE **E**

GUÍA DE SINTAXIS DEL LENGUAJE C++ (ESTANDAR C++ ANSI)

C++ es considerado un C más grande y potente. La sintaxis de C++ es una extensión de C, al que se han añadido numerosas propiedades, fundamentalmente orientada a objetos. ANSI C ya adoptó numerosas características de C++, por lo que la emigración de C a C++ no suele ser difícil.

En este apéndice se muestran las reglas de sintaxis del estándar clásico de C++ recogidas en al *Annotated Manual* (ARM) de Stroustrup y Ellis, así como las últimas propuestas incorporadas al nuevo borrador de C++ ANSI, que se incluyen en las versiones 3 (actual) y 4 (futura de AT&T C++).

E.1. ELEMENTOS DEL LENGUAJE

Un programa en C++ es una secuencia de caracteres que se agrupan en componentes léxicos (*tokens*) que comprenden el vocabulario básico del lenguaje. Estos componentes de léxico son: palabras reservadas, identificadores, constantes, constantes de cadena, operadores y signos de puntuación.

E.1.1. Caracteres

Los caracteres que se pueden utilizar para construir elementos del lenguaje (componentes léxicos o *tokens*) son:

```
a b c d e f g     j k l m n o p q r s t u v w x y z
A B C D E F G .. I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
  - * / . ( )     [ ] < >     ! | / $      ;
```

caracteres espacio (blancos y tabulaciones)

E.1.2. Comentarios

C++ soporta dos tipos de comentarios. Las líneas de comentarios al estilo C y C ANSI, tal como:

```
/* Comentario estilo C */
/* Comentario mas extenso, pero tambien es estilo C y ANSI C */
```

El otro tipo de comentarios se pueden utilizar por los programadores de C++: La versión `/*...*/` se utiliza para comentarios que excedan una línea de longitud y la versión `//...` se utiliza, sólo, para comentarios de una línea. *Los comentarios no se anidan.*

E.1.3. Identificadores

Los *identificadores* (nombres de variables, constantes,...) deben comenzar con una letra del alfabeto (mayúscula o minúscula) o con un carácter subrayado, y pueden tener uno o más caracteres. Los caracteres segundo y posteriores pueden ser: letras, dígitos o un subrayado, no permitiéndose caracteres no alfanuméricos ni espacios.

```
test_prueba    //legal
X123           //legal
multi_palabra  //legal
var25          //legal
15var          //no legal
```

C++ es sensible a las mayúsculas.

```
Paga_mes          es un identificador distinto      a paga_mes
```

Una buena práctica de programación aconseja utilizar identificadores significativos que ayudan a documentar un programa.

```
nombre  apellidos  salario  precio_neto
```

E.1.4. Palabras reservadas

Las palabras reservadas o claves no se deben utilizar como identificadores, debido a su significado estricto en C++; tampoco se deben redefinir. La Tabla E.1 enumera las palabras reservadas de C++ según el ARM[1].

Tabla E.1. Palabras reservadas de C++

asm*	continue	float	new*	signed	try
auto	default	for	operator*	sizeof	typedef
break	delete*	friend*	private*	static	union
case	do	goto	protected*	struct	unsigned
catch*	double	if	public*	switch	virtual*
char	else	inline*	register	template*	void
class*	enum	int	return	this*	volatile
const	extern	long	short	throw*	while

* Estas palabras no existen en ANSI C.

[1] Siglas del libro de BJARNE STROUSTRUP en el que se definen las reglas de sintaxis del lenguaje C++ estándar. *Annotated Reference Manual*, Addison-Wesley, 1992.

Los diferentes compiladores comerciales de C++ pueden incluir, además, nuevas palabras reservadas. Éstos son los casos de Borland, Microsoft y Sysmantec.

Tabla E.2. Palabras reservadas de Turbo/Borland C++

asm	_ds	interrup	short
auto	else	_loadds	signed
break	enum	long	sizeof
case	_es	_near	_ss
catch	_export	near	static
_cdecl	extern	new	struct
cdecl	_far	operator	switch
char	far	pascal	template
class	float	pascal	this
const	for	private	typedef
continue	friend	protected	union
_cs	goto	pubic	unsigned
default	huge	register	virtual
delete	if	return	void
do	inline	_saveregs	volatile
_double	int	_seg	while

Tabla E.3. Palabras reservadas de Microsoft Visual C/C++ 1.5/2.0

_asm	else	int	signed
auto	enum	_interrupt	sizeof
based	_except	_leave	static
break	_export	_loadds	_stcall
case	extern	long	struct
_cdecl	_far	maked	switch
char	_fastcall	_near	thread
const	_finally	_pascal	_try
continue	float	register	typedef
_declspec	for	return	union
default	_fortran	_saveregs	unsigned
dllexport	goto	_self	void
dllimport	_huge	_segment	volatile
do	if	_segname	while
double	_inline	short	

El comité ANSI ha añadido nuevas palabras reservadas (Tabla E.4).

Tabla E.4. Nuevas palabras reservadas de ANSI C++

bool	false	reinterpret_cast	typeid
cons_cast	mutable	static_cast	using
dynamic_cast	namespace	true	wchar_t

E.2. TIPOS DE DATOS

Los tipos de datos simples en C++ se dividen en dos grandes grupos: integrales (datos enteros) y de coma flotante (datos reales). La Tabla E.5. muestra los diferentes tipos de datos en C++.

Tabla E.5. Tipos de datos simples en C++

char	signed char	unsigned char
short	int	long
usigned short	unsigned	unsigned long
float	double	long double

Los tipos derivados en C++ pueden ser:

- enumeraciones (enum),
- estructuras (struct),
- uniones (union),
- arrays,
- clases (class y struct),
- uniones y enumeraciones anónimas,
- punteros,

E.2.1. Verificación de tipos

La verificación o comprobación de tipos en C++ es más rígida (estricta) que en C. Algunas consideraciones a tener en cuenta son:

- *Usar funciones declaradas.* Esta acción es ilegal en C++, y está permitida en C:

```
int main ()
{
    //...
    printf(x);       //C: int printf();
                     //C++ es ilegal, ya que printf no esta declarada
    return 0;
}
```

- *Fallo al devolver un valor de una función.* Una función en C++ declarada con un tipo determinado de retorno, ha de devolver un valor de ese tipo. En C, está permitido no seguir la regla.
- *Asignación de punteros* void. La asignación de un tipo void* a un puntero de otro tipo se debe hacer con una conversación explícita en C++. En C, se realiza implícitamente.
- *Inicialización de constantes de cadena.* En C++ se debe proporcionar un espacio para el carácter de terminación nulo cuando se inicializan constantes de cadena. En C, se permite la ausencia de ese carácter.

```
int main ()
{
     //...
     char car [7] = "Cazorla";   //legal en C
                                 //error en C++
    //...
    return 0;
}
```


Una solución al problema que funciona tanto en C como en C++ es:

```
char car[] = "Cazorla";
```

E.3. CONSTANTES

C++ contiene constantes para cada tipo de dato simple (integer, char,...). Las constantes pueden tener dos sufijos, u, l y f, que indican tipos unsigned, long y float, respectivamente. Así mismo, se pueden añadir los prefijos o y ox, que representan constantes octales y hexadecimales.

```
456  0456 0x456        //constantes enteras:decimal, octal, hexadecimal

1231 123ul             //constantes enteras:long, unsigned long

'B'  'b'  '4'          //constantes tipo char

3.1415f  3.14159L      //constantes reales de diferente precision

"cadena de caracteres" //constante de cadena
```

Las cadenas de caracteres se encierran entre comillas, y las constantes de un solo carácter se encierran entre comillas simples.

```
"" //cadena vacia, \0
```

E.3.1. Declaración de constantes

En C++, los identificadores de variables/constantes se pueden declarar *constantes*, significando que su valor no se puede modificar. Esta declaración se realiza con la palabra reservada const.

```
const double PI = 3.1416;
const char BLANCO = ' ';
const double PI_EG = -I;
const double DOBLE_I = 2 * PI;
```

El modificador de tipos const se utiliza en C++, también para proporcionar protección de sólo lectura para variables y parámetros de funciones. Las funciones miembro de una clase que no modifican los miembros dato a que acceden pueden ser declarados const. Este modificador evita también que parámetros por referencia sean modificados:

```
void copy (const char* fuente, char* destino);
```

E.4. CONVERSIÓN DE TIPOS

Las conversiones explícitas se fuerzan mediante *moldes* (casts). La conversión forzosa de tipos de C tiene el formato clásico:

```
(tipo) expresion
```

C++ ha modificado la notación anterior por una notación funcional como alternativa sintáctica:

```
nombre del tipo (expresion)
```

Las notaciones siguientes son equivalentes:

```
z = float(x);    //notacion de moldes en C++
z = (float)x;    //notacion de moldes en C
```

E.5. DECLARACIÓN DE VARIABLES

En ANSI C, todas las declaraciones de variables y funciones se deben hacer al principio del programa o función. Si se necesitan declaraciones adicionales, el programador debe volver al bloque de declaraciones al objeto de hacer los ajustes o inserciones necesarios. Todas las declaraciones deben hacerse antes de que se ejecute cualquier sentencia. Así, la declaración típica en C++,

```
NombreTipo, NombreVariable1, NombreVariable2, ...
```

proporciona declaraciones tales como:

```
int saldo, meses;
double clipper, salario;
```

Al igual que en C, se pueden asignar valores a las variables en C++:

```
int mes = 4, dia, anyo = 1995;
double salario = 45.675;
```

En C++, las declaraciones de variables se pueden situar en cualquier parte de un programa. Esta característica hace que el programador declare sus variables en la proximidad del lugar donde se utilizan las sentencias de su programa. El siguiente programa es legal en C++, pero no es válido en C:

```
#include <stdio.h>
int main ()
{
   int i;
   for (i = 0; i < 100; ++i)
      printf ("%d\n",i);

   double j;
   for (j = 1.7547; j < 25.4675; j += .001)
       printf ("%lf\n",y);
}
```

El programa anterior se podría reescribir, haciendo la declaración y la definición dentro del mismo bucle:

```
int main ()
{
  for (int i = 0; i < 100; ++i)
      cout << i << endl;

  for (double j = 1.7545; j < 25.4675; j += .001)
      cout << j << endl;
}
```


E.6. OPERADORES

C++ es un lenguaje muy rico en operadores. Se clasifican en los siguientes grupos:

- Aritméticos.
- Relacionales y lógicos.
- Asignación.
- Acceso a datos y tamaño.
- Manipulación de bits.
- Varios.

Como consecuencia de la gran cantidad de operadores, se producen también una gran cantidad de expresiones diferentes.

E.6.1. Operadores aritméticos

C++ proporciona diferentes operadores que relacionan operaciones aritméticas.

Tabla E.6. Operadores aritméticos en C++

Operador	Nombre	Propósito	Ejemplo
+	Más unitario	Valor positivo de x	x = +y, +5
-	Negación	Valor negativo de x	x = -y;
+	Suma	Suma x e y	z = x + y;
-	Resta	Resta y de x	z = x - y;
*	Multiplicación	Multiplica x por y	z = x * y;
/	División	Divide x por y	z = x / y;
%	Módulo	Resto de x dividido por y	z = x % y;
++	Incremento	Incrementa x después de usar	x++
--	Decremento	Decrementa x antes de usar	--x

Ejemplo:

```
-i + w;          //menos unitario mas unitario
a * b/c % c;     //multiplicacion, division, modulo
a - b a -b       //suma y resta binaria
a = 5/2;         //a toma el valor 2, si se considera a entero
a = 5/2;         //a toma el valor 2.5, si a es real
```

Los operadores de incremento y decremento sirven para incrementar y decrementar en uno los valores almacenados en una variable.

```
variable ++      //postincremento
++variable       //preincremento
variable --      //postdecremento
-- variable      //predecremento
++a;             equivale a      a = a + 1;
--b;             equivale a      b = b - 1;
```

Los formatos postfijos se conforman de modo diferente según la expresión en que se aplica:

```
b = ++a;        equivale a      a = a + 1; b = a
b = a++;        equivale a      b = a; a = a + 1;

int i, j, k = 5;
k++;            //k vale 6, igual efecto que ++k
--k;            //k vale ahora 5, igual efecto que k--
k = 5;
i = 4 * k++;    //k es ahora 6 e i es 20
k = 5;
j = 4* ++k;     //k es ahora 6 e i es 24
```

E.6.2. Operadores de asignación

El operador de asignación (=) hace que el valor situado a la derecha del operador se adjudique a la variable situada a su izquierda. La asignación suele ocurrir como parte de una expresión de asignación y las conversiones se producen implícitamente.

```
z = b + 5;      //asigna (b + 5) a variable z
```

C++ permite asignaciones múltiples en una sóla sentencia. Así:

```
a = b +(c=10);
```

equivale a:

```
c = 10;
a = b + c;
```

otros ejemplos de expresiones válidas y no válidas son:

```
//expresiones legales     //expresiones no legales
a = 5 * (b + a);            a + 3 = b;
double x = y;               PI = 3;
a = b = 6;                  x++ = y;
```

C++ proporciona operadores de asignación que combinan operadores de asignación y otros diferentes, produciendo operadores tales como +=, /=, -=, * = y % =. C++ soporta otros tipos de operadores de asignación para manipulación de bits.

Tabla E.7. Operadores aritméticos de asignación

Operador	Formato largo	Formato corto
+=	x = x + y;	x += y;
-=	x = x - y;	x -= y;
*=	x = x * y;	x *= y;
/=	x = x / y;	x /= y;
%=	x = x % y;	x %= y;

Ejemplos

```
a + b;          equivale a  a = a + b;
a * = a + b;    equivale a  a = a * (a + b);
v += e;         equivale a  v = v + e;
v - = e;        equivale a  v = v%e;
```

> Expresiones equivalentes:
>
> ```
> n = n + 1
> n += 1;
> n++;
> ++n;
> ```

E.6.3. Operadores lógicos y relacionales

Los operadores lógicos y relacionales son los bloques de construcción básicos para constructores de toma de decisión en un lenguaje de programación. La Tabla E.8. muestra los operadores lógicos y relacionales:

Tabla E.8. Operadores lógicos y relacionales

Operador	Descripción	Ejemplo
&&	AND (*y*) lógico	a && b
||	OR (*o*) lógico	c || b
!	NOT (*no*) lógico	! c
<	Menor que	i < 0
<=	Menor o igual que	i <= 0
>	Mayor que	j > 50
>=	Mayor o igual que	j >=8.5
==	Igual a	x =='\0'
!=	No igual a	c != '\n'
?:	Asignación condicional	k = (i < 5) ? 1 = i;

El operador ?: se conoce como *expresión condicional*. La expresión condicional es una abreviatura de la sentencia condicional if_else. La sentencia if

```
if (condicion)
   variable = expresion1;
else
   variable = expresion2;
```

es equivalente a

```
variable = (condicion) ? expresion1 : expresion2;
```

La expresión condicional comprueba la condición. Si esa condición es verdadera, se asigna *expresion1* a *variable*; en caso contrario, se asigna *expresion2* a *variable*.

Reglas prácticas

Los operadores lógicos y relacionales actúan sobre valores lógicos: el valor *falso* puede ser o bien 0, o bien el puntero nulo, o bien 0.0; el valor *verdadero* puede ser cualquier valor distinto de cero. La siguiente tabla muestra los resultados de diferentes expresiones.

`x > y`	1, si x excede a y, sino 0
`x >= y`	1, si x es mayor o igual a y, sino 0
`x < y`	1, si x es menor que y, sino 0
`x <= y`	1, si x es menor que o igual a y, sino 0
`x == y`	1, si x es igual a y, sino 0
`x != y`	1, si x e y son distintos, sino 0
`!x`	1, si x es 0, sino 0
`x || y`	0, si ambos x e y son 0, sino 0

Evaluación en cortocircuito

C++, igual que C, admite reducir el tiempo de las operaciones lógicas; la evaluación de las expresiones se reduce cuando alguno de los operandos toman valores concretos.

1. *Operación lógica AND (&&).* Si en la expresión `expr1&& expr2`, `expr1` toma el valor cero y la operación lógica AND (y) siempre será cero, sea cual sea el valor de `expr2`. En consecuencia, `expr2` no se evaluará nunca.
2. *Operación lógica OR (||).* Si `expr1` toma un valor distinto de cero, la expresión `expr1 || expr2` se evaluará a 1, cualquiera que sea el valor de `expr2`; en consecuencia, `expr2` no se evaluará.

E.6.4. Operaciones de manipulación de bits

C++ proporciona operadores de manipulación de bits, así como operadores de asignación de manipulación de bits.

Tabla E.9. Operadores de manipulación de bits (*bitwise*)

Operador	Descripción	Ejemplo
`&`	AND bit a bit	`x & 128`
`|`	OR bit a bit	`j | 64`
`^`	XOR bit a bit	`j ^ 12`
`~`	NOT bit a bit	`~ j`
`<<`	Desplazar a la izquierda	`i << 3`
`>>`	Desplazar a la derecha	`j >> 4`

Tabla E.10. Operadores de asignación de manipulación de bits

Operador	Formato largo	Formato corto
`&=`	`x = x & y;`	`x &= y;`
`|=`	`x = x | y;`	`x |= y;`
`^=`	`x = x ^ y;`	`x ^= y;`
`<<=`	`x = x << y;`	`x <<= y;`
`>>=`	`x = x >> y;`	`x >>= y;`

Ejemplo:

```
~ x      Cambia los bits 1 a 0 y los bits 0 a 1
x & y    Operacion logica AND (y) bit a bit de x e y
x | y    Operacion logica OR (o) bit a bit de x e y
x << y   x se desplaza a la izquierda (en y posiciones)
x >> y   x se desplaza a la derecha (en y posiciones)
```

E.6.5. El Operador *sizeof*

El operador `sizeof` proporciona el tamaño en bites de un tipo de dato o variable `sizeof` toma el argumento correspondiente (tipo escalar, *array*, *record*, etc.). La sintaxis del operador es:

```
sizeof (nombre_variable | tipo_de_dato)
```

Ejemplo:

```
int m, n[12];
sizeof(m)        proporciona 4, en maquinas de 32 bits
sizeof(n)       //proporciona 48
sizeof(15)      //proporciona 4
Tamanyo = sizeof(long) - sizeof(int);
```

E.6.6. Prioridad y asociatividad de operadores

Cuando se realizan expresiones en las que se mezclan operadores diferentes es preciso establecer una *precedencia* (prioridad) de los operadores y la *dirección* (o secuencia) de evaluación (orden de evaluación: izquierda-derecha, derecha-izquierda), denominada *asociatividad*.

La Tabla E.11 muestra la precedencia y asociatividad de operadores.

Ejemplo:

```
a * b/c * d    equivale a    (a * b) / (c * d)
```

E.6.7. Sobrecarga de operadores

La mayoría de los operadores de C++ pueden ser sobrecargados o redefinidos para trabajar con nuevos tipos de datos. La Tabla E.12. lista los operadores que pueden ser sobrecargados.

Tabla E.11. Precedencia y asociatividad de operadores

Operador	Asociatividad	Prioridad
:: () [] . ->	Izquierda-Derecha	1
++ --	Derecha-Izquierda	2
& (*direccion*) ~ (*tipo*) ! - + sizeof (*tipo*) new delete * (*indireccion*)		
.* ->*	Izquierda-Derecha	3
* / %	Izquierda-Derecha	4
+ -	Izquierda-Derecha	5
<< >>	Izquierda-Derecha	6
< <= > >=	Izquierda-Derecha	7
== !=	Izquierda-Derecha	8
&	Izquierda-Derecha	9
^	Izquierda-Derecha	10
|	Izquierda-Derecha	11
&&	Izquierda-Derecha	12
||	Izquierda-Derecha	13
?:	Derecha-Izquierda	14
= += -= *= /= %= >>= <<= &= |= ^=	Derecha-Izquierda	15
, (*operador coma*)	Izquierda-Derecha	16

Tabla E.12. Operadores que se pueden sobrecargar

+	-	*	\	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	,	->*	->( )	[]			

E.7. ENTRADAS Y SALIDAS BÁSICAS

Al contrario que muchos lenguajes, C++ no tiene facilidades incorporadas para manejar entrada o salida. En su lugar, se manejan por rutinas de bibliotecas. Las clases que C++ utiliza para entrada y salida se conocen como *flujos*. Un *flujo (stream)* es una secuencia de caracteres junto con una colección de rutinas para insertar caracteres en flujos (a pantalla) y extraer caracteres de un flujo (de teclado).

E.7.1. Salida

El flujo `cout` es el flujo de salida estándar que corresponde a `stdout` en C. Este flujo se deriva de la clase `ostream` construida en `isostream`.

Figura E.1. Uso de flujos para salida de caracteres.

Si se desea visualizar el valor del objeto `int` llamado `i`, se escribe la sentencia:

```
cout << i;
```

El siguiente programa visualiza en pantalla una frase:

```
#include <isostream.h>
int main ()
{
   cout <<"hola, mundo\n";
}
```

Las salidas en C++ se pueden conectar en cascada, con una facilidad de escritura mayor que en C.

```
#include <isostream.h>
int main ()
{
   int i;
   i = 1099;
   cout <<"El valor de i es" << i <<'\n';
}
```

Otro programa que muestra la conexión en cascada es:

```
#include <iostream.h>
int main ()
{
   int x = 45;
   double y = 495.125;
   char *c = "y multiplicada por x=";
   cout << c << y * x << "\n";
}
```

E.7.2. Entrada

La entrada se maneja por la clase `istream`. Existe un objeto predefinido `istream`, llamado `cin`, que se refiere al dispositivo de entrada estándar (el teclado). El operador que se utiliza para obtener un valor del teclado es el *operador de extracción* >>. Por ejemplo, si `i` era un objeto `int`, se escribirá:

```
cin >> i;
```

que obtiene un número del teclado y lo almacena en la variable `i`.

Un programa simple que lee un dato entero y lo visualiza en pantalla es:

```
#include <iostream.h>
int main ()
{
   int i;
   cin >> i; '\n';
   cout << i <<
```

Al igual que en el caso de `cout`, se pueden introducir datos en cascada:

```
#include <iostream.h>
int main ()
{
   char c[60];
   int x,y;

   cin >> c >> x >> y;
   cout << c << " "<< x <<" "<< y << "\n";
}
```

E.7.3. Manipuladores

Un método fácil de cambiar la anchura del flujo y otras variables de formato es utilizar un operador especial denominado *manipulador*. Un manipulador acepta una referencia de flujo como un argumento y devuelve una referencia al mismo flujo.

El siguiente programa muestra el uso de manipuladores específicamente para conversiones de número (`dec`, `oct`, y `hex`)

```
#include <iostream.h>
int main ()
{
   int i = 36;
   cout << dec << i << oct << i << " "<< hex << i << "\n";
}
```

La salida de este programa es:

```
36   44   24
```

Otro manipulador típico es `endl`, que representa al carácter de nueva línea (*salto de línea*), es equivalente a `'\n'`. El programa anterior se puede escribir también así:

```
#include <isostream.h>
int main ()
{
   int i = 36;

   cout << dec << i <<" " << oct << i << " " << hex << i << endl;
}
```

E.8. SENTENCIAS

Un programa en C++ consta de una secuencia de sentencias. Existen diversos tipos de sentencias. El punto y coma se utiliza como elemento terminal de cada sentencia.

E.8.1. Sentencia de declaración

Se utilizan para establecer la existencia y, opcionalmente, los valores iniciales de objetos identificados por nombre.

```
NombreTipo identificador, ...;
NombreTipo identificador = expresion, ...;
const NombreTipo identificador = expresion, ...;
```

Algunas sentencias válidas en C++ son:

```
char c1;
int p, q = 5, r = a + b;   //suponiendo que a y b han
                           //sido declaradas e
                           //inicializadas antes
const double IVA = 16.0;
```

E.8.2. Sentencias expresión

Las sentencias de expresiones hacen que la expresión sea evaluada. Su formato general es:

```
expresion;
```

Ejemplo:

```
n++;
425;                    //legal, pero no hace nada
a + b;                  //legal, pero no hace nada
n = a < b || b != 0;
a += b = 3;             //sentencia compleja
```

C++ permite asignaciones múltiples en una sentencia.

```
m = n + (p = 5);     equivale a      p = 5;
                                     m = n + p;
```

E.8.3. Sentencias compuestas

Una *sentencia compuesta* es una serie de sentencias encerradas entre llaves. Las sentencias compuestas tienen el formato:

```
{
   sentencia
   sentencia
   sentencia
   ...
}
```

E.9.2. Sentencias de alternativa múltiple: *switch*

La sentencia switch ofrece una forma de realizar decisiones de alternativas múltiples. El formato de switch es:

```
switch (expresion)
{
   case constante 1:
     sentencias
     break;
   case constante 2:
     sentencias
     .
     .
     .
     break;
   case constante n:
     sentencias
     break;
   default: //opcional
     sentencias
}
```

La sentencia switch requiere una expresión cuyo valor sea entero. Este valor puede ser una constante, una variable, una llamada a función o una expresión. El valor de *constante* ha de ser una constante. Al ejecutar la sentencia se evalúa *expresion* y si su valor coincide con una *constante* se ejecutan las sentencias a continuación de ella, en caso contrario se ejecutan las sentencias a continuación de default.

```
switch (Puntos)
{
   case 10:
    nota = 'A';
    break;
   case 9:
    nota = 'B';
    break;
   case 7,8:
    nota = 'C';
    break;
   case 5,6:
    nota = 'D';
    break;
   default:
   nota = 'F';
```


E.10. BUCLES: SENTENCIAS REPETITIVAS

Los *bucles* sirven para realizar tareas repetitivas. En C++, existen tres diferentes tipos de sentencias repetitivas:

- while
- do
- for

E.10.1. Sentencia *while*

La sentencia while es un bucle condicional que se repite mientras la condición es verdadera. El bucle while nunca puede iterar si la condición comprobada es inicialmente falsa. La sintaxis de la sentencia while es:

```
while (expresion)
   sentencia;
```

o bien:

```
while (expresion) {
  < secuencia de sentencias >
}
```

Ejemplo:

```
int n, suma = 0;
int i = 1;
while (i <= 100)
{
   cout << "Entrar";
   cin >> n;
   suma += n;
   i++;
}
cout << "La media es" << double (suma)/100.0;
```

E.10.2. Sentencia *do*

La sentencia do actúa como la sentencia while. La única diferencia real es que la evaluación y la prueba de salida del bucle se hace después que el cuerpo del bucle se ha ejecutado, en lugar de antes. El formato es:

```
do
   sentencias
while (expresion);
sentencia siguiente
```

Se ejecuta `sentencia` y a continuación se evalúa `expresion` y, si es verdadero (distinto de cero), el control se pasa de nuevo al principio de la sentencia do y el proceso se repite, hasta que `expresion` es falso (cero) y el control pasa a la `sentencia siguiente`.

Ejemplo:

```
int n, suma = 0;
int i = 1;
do
{
    cout <<"Entrar";
    cin >>n;
    suma += n;
    i++;
}while (i <= 100);
cout << "La media es" << double (suma)/100.0;
```

El siguiente ejemplo visualiza los cuadrados de 2 a 10:

```
int i = 2;
do
{
    cout << i << "12 = " << i * i++ << endl;
}while (i < 11);
```

E.10.3. La sentencia *for*

Una sentencia `for` ejecuta la iteración de un bucle un número determinado de veces. `for` tiene tres componentes: *expresion1*, inicializa las variables de control del bucle; *expresion2*, es la condición que determina si el bucle realiza otra iteración; la última parte del bucle `for` es la cláusula que incrementa o decrementa las variables de control del bucle. El formato general de `for` es:

```
for (expresion1; expresion2; expresion3)
    sentencia;|<secuencia de sentencias>;
```

expresion1 se utiliza para inicializar la variable de control de bucle; a continuación *expresion2* se evalúa, si es verdadera (distinta de cero), se ejecuta la sentencia y se evalúa *expresion3* y el control pasa de nuevo al principio del bucle. La iteración continúa hasta que *expresion2* es falsa (cero), en cuyo momento el control pasa a la sentencia siguiente al bucle.

Ejemplos:

```
1. for (int i = 0; i < N; i++)  //se relaciona N iteraciones
       sentencias1

2. Suma de 100 numeros
   int n, suma = 0;
   for (int i = 0; i < 100; i++)
   {
      cout << "Entrar";
      cin >> n;
      suma += n;
   }
```


E.10.4. Sentencias *break* y *continue*

El flujo de control ordinario de un bucle se puede romper o interrumpir mediante las sentencias `break` y `continue`.

La sentencia `break` produce una salida inmediata del bucle `for` en que se encuentra situada:

```
for (i = 0; i < 100; ++i)
{
   cin >> x;
   if (x < 0.0) )
     cout << "salir del bucle" <<endl;
     break;
   }
   cout << sqrt(x) << endl;
}
```

La sentencia `break` también se utiliza para salir de la sentencia `switch`.

La sentencia `continue` termina la iteración que se está realizando y comenzará de nuevo la siguiente iteración:

```
for (i = 0; i < 100; ++i)
   cin >> x;
   if (x < 0.0)
     continue;
}
```

> ***Advertencia:***
>
> - Una sentencia `break` puede ocurrir únicamente en el cuerpo de una sentencia `for`, `while`, `do` o `switch`.
> - Una sentencia `continue` sólo puede ocurrir dentro del cuerpo de una sentencia `for`, `while` o `do`.

E.10.5. Sentencia nula

La `sentencia nula` se representa por un punto y coma, y no hace ninguna acción.

```
char cad[80]="Cazorla";
int i;

for (i = 0; cad[i] !='\0'; i++)
    ;
```

E.10.6. Sentencia *return*

La *sentencia return* detiene la ejecución de la función actual y devuelve el control a la función llamada. Su sintaxis es:

```
return expresion;
```

donde el valor de *expresion* se devuelve como el valor de la función.

E.11. PUNTEROS (APUNTADORES)[2]

Un puntero o apuntador es una referencia indirecta a un objeto de un tipo especificado. En esencia, un puntero contiene la posición de memoria de un tipo dado.

E.11.1. Declaración de punteros

Los punteros se declaran utilizando el operador unitario *. En las sentencias siguientes se declaran dos variables: n es un entero, y p es un puntero a un entero.

```
int n;  //n es un tipo de dato entero
int *p; //p es un puntero a un entero
```

Una vez declarado un puntero, se puede fijar la dirección o posición de memoria del tipo al que apunta.

```
p = &n; //p se fija a la direccion de a
```

> Un puntero se declara escribiendo:
>
> *NombreTipo * Nombre Variable*

Una vez que se ha declarado un puntero, p, el objeto al que apunta se escribe *p y se puede tratar como cualquier otra variable de tipo NombreTipo.

```
int *p, *q, n;   //dos punteros a int, y un int
n = -25;         //n se fija a -16
*p = 105;        //*p a 101
*q = n + *p;     //*q a 80
```

C++ trata los punteros a tipos diferentes como tipos diferentes,

```
int *ip;
double *dp;
```

Los punteros ip y dp son incompatibles, de modo que es un error escribir

```
dp = ip;          //Error, no se pueden asignar punteros a tipos diferentes
```

Se pueden, sin embargo, realizar asignaciones entre contenidos, ya que se realizaría una conversión explícita de tipos.

```
*dp = *ip;
```

[2] El termino *puntero* es el más utilizado en España, mientras que en Latinoamérica se suele utilizar *apuntador*.

Existe un puntero especial (*nulo*) que se suele utilizar con frecuencia en programas C++. El puntero NULL tiene un valor cero, que lo diferencia de todas las direcciones válidas. El conocimiento nos permite comprobar si un puntero p es el puntero NULL evaluando la expresión (p==0). Los punteros NULL se utilizan sólo como señales de que ha sucedido algo. En otras palabras, si p es un puntero NULL, es correcto referenciar *p.

E.11.2. Punteros a arrays

A los arrays se accede a través de los índices:

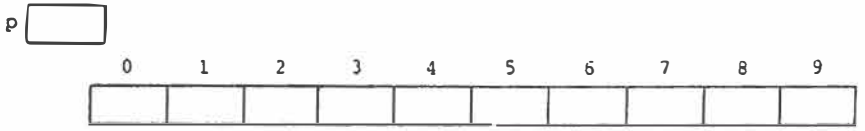
```
int lista [5];
lista [3] = 5;
```

A los arrays también se puede acceder a través de punteros:

```
int lista [5]; //array de 5 elementos
int *ptr;      //puntero a entero
ptr = lista;   //fija puntero al primer elemento del array
ptr += 3;      //suma 3 a ptr; ptr apunta al 4º elemento
*ptr = 5;      //establece el 4º elemento a 5.
```

> El nombre de un array se puede utilizar también como si fuera un puntero al primer elemento del array.

```
double a [10];
double *p = a;    //p y a se refieren al mismo array
```



Este elemento se puede llamar por: a[0], *p *o bien* p[0]

Este elemento se puede llamar por: a[6], *(p + 6) *o bien* p[0]

Si nombre apunta al primer elemento del array, entonces nombre + 1 apunta al segundo elemento. El contenido de lo que se almacena en esa posición se obtiene por la expresión:

```
*(nombre + 1)
```

> Aunque las funciones no pueden modificar sus argumentos, si un array se utiliza como un argumento de una función, la función puede modificar el contenido del array.

E.11.3. Punteros a estructuras

Los punteros a estructuras son similares y funcionan de igual forma que los punteros a cualquier otro tipo de dato.

```
struct familia
{
   char *marido;
   char *esposa;
   char *hijo;
};

familia Mackoy;     //Mackoy estructura de tipo familia
familia *p;         //p, un puntero a familia
p = &Mackoy         //p, contiene direccion de mackoy

p marido = "Luis Mackoy";     //inicializacion
p esposa = "Vilma Gonzalez"; //inicializacion
p hijo = "Luisito Mackoy";    //inicializacion
```

E.11.4. Punteros a objetos constantes

Cuando se pasa un puntero a un objeto grande, pero se trata de que la función no modifique el objeto (por ejemplo, el caso de que sólo se desea visualizar el contenido de un array), se declara el argumento correspondiente de la función como puntero a un objeto constante.

La declaración:

```
const NombreTipo *v
```

establece v como un puntero a un objeto que no puede ser modificado. Un ejemplo puede ser:

```
void Visualizar(const ObjetoGrande *v);
```

E.11.5. Punteros a *void*

El tipo de dato `void` representa un valor nulo. En C++, sin embargo, el tipo de puntero `void` se suele considerar como un puntero a cualquier tipo de dato. La idea fundamental que subyace en el puntero `void` en C++ es la de un tipo que se puede utilizar adecuadamente para acceder a cualquier tipo de objeto, ya que es más o menos independiente del tipo.

Un ejemplo ilustrativo de la diferencia de comportamiento en C y C++ es el siguiente segmento de programa:

```
int main()
{
   void *vptr;
   int *iptr;

   vptr = iptr;
   iptr = vptr;                 //Incorrecto en C++, correcto en C
   iptr = (int *) vprr;         //Correcto en C++
   ...
}
```


E.11.6. Punteros y cadenas

Las cadenas en C++ se implementan como arrays de caracteres, como constantes de cadena y como punteros a caracteres.

Constantes de cadena

Su declaración es similar a

```
char *Cadena = "Mi profesor";
```

o bien su sentencia equivalente

```
char VarCadena[] = "Mi profesor";
```

Si desea evitar que la cadena se modifique, añada `const` a la declaración:

```
const char *VarCadena = "Mi profesor";
```

> Los puntos a cadena se declaran:
>
> char a [] *o bien:* char *a

Punteros a cadenas

Los punteros a cadenas no son cadenas. Los punteros que localizan el primer elemento de una cadena almacenada.

```
char *varCadena;
const char *Cadenafija;
```

Consideraciones prácticas

Todos los arrays en C++ se implementan mediante punteros:

```
char cadena1[16]= "Concepto Objeto";
char * cadena2 = cadena1;
```

Las declaraciones siguientes son equivalentes y se refieren al carácter 'C':

```
cadena1[0]  cadena1  cadena2
```

E.11.7. Aritmética de punteros

Dado que los punteros son números (direcciones), puden ser manipulados por los operadores aritméticos. Las operaciones que están permitidas sobre punteros son: suma, resta y comparación. Así, si las sentencias siguientes se ejecutan en secuencia:

```
char *p;    //p, contiene la direccion de un caracter
char a[10]; //array de diez caracteres
p = &a[0];  //p, apunta al primer elemento del array
p++;        //p, apunta al segundo elemento del array
p++;        //p, apunta al tercer elemento del array
p--;        //p, apunta al segundo elemento del array
```

Un elemento de comparación de punteros, es el siguiente programa:

```
#include <iostream.h>
main (void)
{
  int *ptr1, *ptr2;
  int a[2] = {10,10};
  ptr1 =a;
  cout << "ptr1 es" << "ptr1<< "*ptr1 es" << *ptr1 <<endl;
  ptr2 = ptr1 + 1;
  cout << "ptr2 es" << ptr2 << "*ptr2 es" << *ptr2 <<endl;

  //comparar dos punteros
  if (ptr1 == ptr2)
      cout << " ptr1 no es igual a ptr2 \n";

  if (*ptr1 == *ptr2)
      cout <<"*ptr1 es igual a *ptr2 \n";
  else
      cout <<"*ptr1 no es igual a *ptr2 \n";
}
```

E.12. LOS OPERADORES *NEW* Y *DELETE*

C++ define un método para realizar asignación dinámica de memoria, diferente del utilizado en C, mediante los operadores **new** y **delete**.

El operador new sustituye a la función `malloc` tradicional en C, y el operador `delete` sustituye a la función `free` tradicional, también en C. new, asigna memoria, y devuelve un puntero al objeto últimamente creado; su sintaxis es:

```
new NombreTipo
```

y un ejemplo de su aplicación es:

```
int *ptr1
double *ptr2;
ptr1 = new int;     //memoria asignada para el objeto ptr1
ptr2 = new double; //memoria ampliada para el objeto ptr2
*ptr1 = 5;
*ptr2 = 6.55;
```

Dado que new devueve un puntero, se puede utilizar ese puntero para inicializar el puntero de una sola definición, tal como:

```
int* p = new int;
```

Si new no puede ocupar la cantidad de memoria solicitada devuelve un valor NULL. El operador `delete` libera la memoria asignada mediante new.

```
delete ptr1;
```


Un pequeño programa que muestra el uso combinado de new y `delete` es:

```
#include <iostream.h>
void main (void)
{
   char *c;

  . c = new char[512];
   cin >> c;
   cout << c << endl;

   delete c;
}
```

Los operadores new y `delete` se pueden utilizar para asignar memoria a arrays, clases y otro tipo de datos.

```
int *i;
i = new int[2][35];     // crear el array
//asignar el array
...
delete i;               // destruir el array
```

Sintaxis de new y `delete`

```
new nombre-tipo                    new Int       new char[100]
new nombre-tipo inicializador      new int(99)   new char('C')
new nombre-tipo                    new (char*)
delete expresion                   delete p
delete[] expresion                 delete []p
```

E.13. ARRAYS

Un *array*[3] (*matriz, tabla*) es una colección de elementos dados del mismo tipo que se identifican por medio de un índice. Los elementos comienzan con el índice 0.

Declaración de arrays

Un declaración de un array tiene el siguiente formato:

```
nombreTipo   nombreVariable[n]
```

Algunos ejemplos de arrays unidimensionales:

```
int ListaNum[2];        //array de dos enteros
char ListaNombres[10]; //.array de 10 caracteres
```

3 En Latinoamérica, este término se traduce al español por la palabra *arreglo*.

Arrays *multidimensionales* son:

```
nombretipo      nombreVariable[n1][n2]...[nx];
```

El siguiente ejemplo declara un array de enteros 4 x 10 x 3

```
int multidim [4][10][3];
```

El ejemplo tabla declara un array de 2 x 3 elementos

```
int tabla [2][3]; //array de enteros de 2 x 3 = 6 elementos
```

E.13.1. Definición de arrays

Los arrays se inicializan con este formato:

```
int a [3]       = {5, 10, 15};
char cad[5]     = {'a', 'b', 'c', 'd', 'e'};
int tabla [2][3] = {{1,2,3} {3,4,5}};
```

Las tres siguientes definiciones son equivalentes:

```
char saludo [5] = "hola";
char saludo []  = "hola";
char saludo [5] = {'h', 'o', 'l', 'a', '\0'};
```

1. Los arrays se pueden pasar como argumentos a funciones.
2. Las funciones no pueden devolver arrays.
3. No está permitida la asignación entre arrays. Para asignar un array a otro, se debe escribir el código para realizar las asignaciones elemento a elemento.

E.14. ENUMERACIONES, ESTRUCTURAS Y UNIONES

En C++, un nombre de una enumeración, estructura o unión es un nombre de un tipo. Por consiguiente, la palabra reservada `struct`, `union`, o `enum` no son necesarias cuando se declara una variable.

El tipo de dato *enumerado* designa un grupo de constantes enteros con nombres. La palabra reservada `enum` se utiliza para declarar un tipo de dato enumerado o *enumeración*. La sintaxis es:

```
enum nombre
{
  lista-simbolos
}
```

donde *nombre* es el nombre de la variable declarada enumerada, *lista-simbolos* es una lista de tipos enumerados, a los que se asigna valores cuando se declara la variable enumerada y puede tener un valor de inicialización. Se puede utilizar el nombre de una enumeración para declarar una variable de ese tipo (variable de enumeración).

```
nombre var;
```


Considérese la siguiente sentencia:

```
enum color {Rojo, Azul, Verde, Amarillo};
```

Una variable de tipo enumeración color es:

```
color pantalla = Rojo; //Estilo C++
```

Una *estructura* es un tipo de dato compuesto que contiene una colección de elementos de tipos de datos diferentes combinados en una única construcción del lenguaje. Cada elemento de la colección se llama `miembro` y puede ser una variable de un tipo de dato diferente. Una estructura representa un nuevo tipo de dato en C++.

La sintaxis de una estructura es:

```
struct nombre
{
   miembros
};
```

Un ejemplo y una variable tipo estructura se muestran en las siguientes sentencias:

```
  struct cuadro {
    int i;
    float f;
  };

struct cuadro nombre; //Estilo C
cuadro nombre;        //Estilo C++
```

Una *unión* es una variable que puede almacenar objetos de tipos y tamaños diferentes. Una unión puede almacenar tipos de datos diferentes, sólo puede almacenar uno cada vez, en oposición a una estructura que almacena simultáneamente una colección de tipos de datos. La sintaxis de una unión es:

```
union nombre {
  miembros
};
```

Un ejemplo de una unión es:

```
union alfa {
  int x;
  char c;
};
```

Una declaración de una variable estructura es:

```
alfa w;
```

El modo de acceder a los miembros de la estructura es mediante el operador punto.

```
u.x = 145;
u.c = 'z';
```

C++ admite un tipo especial de unión llamada *unión anónima*, que declara un conjunto de miembros que comparten la misma dirección de memoria. La unión anónima no tiene asignado un nombre y, en consecuencia, se accede a los elementos de la unión directamente.

La sintaxis de una unión anónima es:

```
union {
  int nuevoID;
  int contador;
};
```

Las variables de la unión anónima comparten la misma posición de memoria y espacio de datos.

```
int main()
{
  union {
    int x;
    float y;
    double z;
  };
  x = 25;
  y = 245.245;    //el valor en y sobreescribe el valor de x

  z = 9.41415;    //el valor en z sobreescribe el valor de z
}
```

E.15. CADENAS

Una *cadena* es una serie de caracteres almacenados en bytes consecutivos de memoria. Una cadena se puede almacenar en un array de caracteres (`char`) que termina en un carácter nulo (cero, `'\0'`).

```
char perro[5] = {'m','o','r','g','a','n'}; // no es una cadena
char gato[5] = {'f','e','l','i','s','\o'}; // es una cadena
```

Lectura de una cadena del teclado

```
#include <isostream.h>
main()
{
   char cad [80];
   cout <<"Introduzca una cadena:";    // lectura del teclado
   cin >> cad;
   cout << "Su cadena es:";
   cout << cad;

   return 0;
}
```


Esta lectora del teclado lee una cadena hasta que se encuentra el primer carácter blanco. Así, cuando se lee "`Sierra Magina. Jaen`" la primera cadena, en `cad` sólo se almacena `Sierra`. Para resolver el problema, utilizará la función `gets()` que lee una cadena completa leída del teclado. El programa anterior se lee así:

```
#include <iostream.h>
#include <stdio.h>

main()
{
   char cad[80];

   cout << "Introduzca una cadena:";
   gets(cad);
   cout << "Su cadena es:";
   cout << cad;

   return 0;
}
```

E.16. FUNCIONES

Una *función* es una colección de declaraciones y sentencias que realizan una tarea única. Cada función tiene cuatro componentes: (1) su nombre, (2) el tipo de valor que devuelve cuando termina su tarea, (3) la información que toma al realizar su tarea, y (4) la sentencia o sentencias que realizan su tarea. Cada programa C++ tiene al menos una función: la función `main`.

E.16.1. Declaración de funciones

En C++, se debe declarar una función antes de utilizarla. La declaración de la función indica al compilador el tipo de valor que devuelve la función y el número y tipo de argumentos que toma. La declaración en C++ se denomina *prototipo*:

```
tipo   NombreFuncion (lista argumentos);
```

Ejemplos válidos son:

```
double Media(double x, double y);
void print(char* formato,...);
extern max(const int*, int);
char LeerCaracter();
```

E.16.2. Definición de funciones

La definición de una función es el cuerpo de la función que se ha declarado con anterioridad.

```
double Media(double x, double y)
//Devuelve la media de x e y
{
   return (x + y)/2.0
}
```

```
char LeerCaracter();
//Devuelve un caracter de la entrada estandar
{
   char c;
   cin >> c;
   return c;
}
```

E.16.3. Argumentos por omisión

Los parámetros formales de una función pueden tomar valores por omisión, o argumentos cuyos valores se inicializan en la lista de argumentos formales de la función.

```
int Potencia (int n, int k = 2);
Potencia(256);                    //256 elevado al cuadrado
```

Los parámetros por omisión no necesitan especificarse cuando se llama a la función. Los parámetros con valores por omisión deben estar al final de la lista de parámetros:

```
void funcl (int i =3, int j);              //ilegal
void func2 (int i, int j = o, int k = 0);  //correcto
void func3 (int i = 2, int j, int k = 0);  //ilegal
void ImprimirValores (int cuenta, double cantidad = 0.0);
```

Llamadas a la función `ImprimirValores`:

```
ImprimirValores (n,a);
ImprimirValores (n);    //equivalente a ImprimirValores (n, 0.0)
```

Otras declaraciones y llamadas a funciones son:

```
double f1(int n, int m, int p=0);          //legal
double f2(int n, int m = 1, int p = 0);    //legal
double f3(int n = 2, oint m = 1, int p = 0); //legal
double f4(int n, int m = 1, int p);        //ilegal
double f5(int n = 2, int m, int p = 0);    //ilegal
```

E.16.4. Funciones en línea (`inline`)

Si una declaración de función está precedida por la palabra reservada `inline`, el compilador sustituye cada llamada a la función con el código que implementa la función.

```
inline int Max(in a, int b)
{
   if (a > b) return a;
      return b;
}

main()
{
   int x = 5, y = 4;
   int z = Max(x,y);
}
```


Los parámetros con valores por omisión deben entrar al final de la lista de parámetros: Las funciones `f1`, `f2` y `f3` son válidas, mientras que las funciones `f4` y `f5` no son válidas.

Las funciones *en línea* (*inline*) evitan los tiempos suplementarios de las llamadas múltiples a funciones. Las funciones declaradas en línea deben ser simples con sólo unas pocas sentencias de programa; sólo se pueden llamar un número limitado de veces y no son recursivas.

```
inline int abs(int i);
inline int min(int v1, int v2);
int mcd(int v1, int v2);
```

Las funciones en línea deben ser definidas antes de ser llamadas.

```
#include <isostream.h>
int incrementar (int i);

inline incrementar (int i)
{
   i++
   return;
}
main(void)
{
   int i = 0;
   while (i < 5)
   {
       i = incrementar (i);
      cout << "i es: " << i <<endl;
   }
}
```

E.16.5. Sobrecarga de funciones

En C++, dos o más funciones distintas pueden tener el mismo nombre. Esta propiedad se denomina *sobrecarga*. Un ejemplo es el siguiente:

```
int max (inc, int);
double max (double, double);
```

o bien este otro:

```
void sumar (char i);
void sumar (float J);
```

Las funciones sobrecargadas se diferencian en el número y tipo de argumentos, o en el tipo que devuelven las funciones, y sus cuerpos son diferentes en cada una de ellas.

```
#include <iostream.h>

void suma (char);
void suma (float);
main (void)
{
   int i = 65;
   int j = 6.5;
```

```
    suma(i);
    suma(j);
}

void suma(char i)
{
   cout << "Suma interior(char)" <<endl;
}

void suma(float j)
{
   cout << Suma interior (float) " << endl;
}
```

E.16.6. El modificador *const*

El modificador de tipos `const` se utiliza en C++ para proporcionar protección de sólo lectura para variables y parámetros de funciones. Cuando se hace preceder un tipo de argumento con el modificador `const` para indicar que este argumento no se puede cambiar, el argumento al que se aplica no se puede asignar un valor ni cambiar.

```
void copia (const char * fuente, char* dest);
void func_demo (const int i);
```

E.16.7. Paso de parámetros a funciones

En C++ existen tres formas de pasar parámetros a funciones:

1. *Por valor.* La función llamada recibe una copia del parámetro y este parámetro no se puede modificar dentro de la función

```
void intercambio (intx, int y)
{
   int aux = y;
       y = x;
       x = aux;
}
//...
intercambio (i, j); //las variables i, j, no se intercambian
```

2. *Por dirección.* Se pasa un puntero al parámetro. Este método permite simular en C/C++ la llamada por referencia, utilizando tipos punteros en los parámetros formales en la declaración de prototipos. Este método permite modificar los argumentos de una función.

```
void intercambio (int*x, int*y)
{
   int aux = *y;
       *y = *x;
       *x = aux;
}
//...
intercambio (&i, &j); //i,   se intercambiamus valores
```


3. *Por referencia.* Se pueden pasar tipos referencia como argumentos de funciones, lo que permite modificar los argumentos de una función.

```
void intercambio (int&x, int&y);
{
  int aux = y;
      y = x;
      x = aux;
}
//...
intercambio (i, j);    //i, j intercambian sus valores
```

> Si se necesita modificar un argumento de una función en su interior, el argumento debe ser un tipo de referencia en la declaración de la función.

E.16.8. Paso de arrays

Los arrays se pasan por referencia. La dirección del primer elemento del array se pasa a la función; los elementos individuales se pasan por valor. Los arrays se pueden pasar indirectamente por su valor si el array se define como un miembro de una estructura.

```
//Paso del array completo. Ejemplo 1
#include <isostream.h>
void func1 (int x []);     //prototipo de funcion

void main() {
   int a[3] = {1,2,3};
   func1(a);              //sentencias
   func1(&a[0];           //equivalentes
}

void func(int x[]) {
   int i;
   for (i = 0; i < 3; i + 1)
     cout << i << x[i] << '\n';
}
```

El siguiente ejemplo pasa un elemento de un array:

```
#include <isostream.h>

{
const int N = 3;
void func2(int x);
void main() {
   int a[N] = {1,2,3};
   func2(a[2]);
}

void func2(int x) {
   cout << x << '\n';
```

ANEXO H

CÓDIGO PROGRAMA ***CALMI A.CPP***

PROGRAMA CALMI_A.CPP

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <complex.h>

/* Definicion de constantes */

   const int MAX = 23 ;

/* Definicion de variables  */

   /* Archivo de texto */

      FILE     *archivo   ;      //Manipula archivos de entrada y salida
      char     a_ent[30] ;       //Nombre de archivo de entrada
      char     a_sal[30] ;       //Nombre de archivo de salida

   /* Datos de linea */

      int      i  ,              //Barra de envio
               j    ;           //Barra de recepcion
      double   r  ,              //Resistencia de linea
               x  ,              //Reactancia de linea
               ys   ;           //Admitancia shunt de linea (Yc/2)

   /* Matriz de admitancias */

      complex  Y[MAX][MAX] ;     //Matriz de admitancias
      int      n ;               //Dimension de Y (Tamaño maximo = MAX)


/* Definicion de procedimientos y funciones */

   void Inicializa_matriz() ;
   void Inversion_Shipley(int) ;
   void Genera_reporte(int,char *) ;

void main()
 {

   /* Identificacion del archivos                        */

      clrscr() ;
      printf ("*** Calculo de la Matriz de Impedancias ***\n\n") ;
      printf ("* Archivo de entrada : ") ;
      scanf ("%s", &a_ent) ;
      printf ("* Archivo de salida : ") ;
      scanf ("%s", &a_sal) ;
      printf ("\n\n") ;

   /* Procesamiento del archivo de entrada               */

      if ((archivo=fopen(a_ent,"rt"))==NULL)
       printf ("Archivo no encontrado ... \n\n") ;
      else
       {
       printf ("Procesando archivo %s ...\n\n",a_ent) ;

       Inicializa_matriz() ;

       fscanf(archivo,"%d",&n) ;
       while(!feof(archivo))
        {
          fscanf(archivo,"%d%d%lf%lf%lf",&i,&j,&r,&x,&ys) ;
```

```
            if (i==j)
             Y[i-1][i-1] = Y[i-1][i-1] + 1/complex(r,x) + complex(0,ys)
            else
             {
               Y[i-1][i-1] = Y[i-1][i-1] + 1/complex(r,x) + complex(0,ys)
               Y[j-1][j-1] = Y[j-1][j-1] + 1/complex(r,x) + complex(0,ys)
               Y[i-1][j-1] = Y[j-1][i-1] = - 1/complex(r,x) ;
             }
          }

         Inversion_Shipley(n) ;

         Genera_reporte(n,a_sal) ;
        }

       fclose(archivo) ;

   /* Salida del programa                                    */

       printf ("\n\n") ;
       printf ("Presione cualquier tecla para salir ...") ;
       getch() ;

 }

/**/void Inicializa_matriz()
     {
       int  k, m ;

       for (k=0;k<MAX;k++)
       for  (m=0;m<MAX;m++)  Y[k][m] = complex(0,0) ;
     }

/**/void Inversion_Shipley(int n)
     {
       int  i, j, k ;

       /* Inversion Shipley-Coleman */

         for (k=0;k<n;k++)
          {

            /* Elemento Pivote */

              Y[k][k] = -1/Y[k][k] ;

            /* Columna Pivote */

              for (i=0;i<n;i++)
               {
                 if (i!=k)  Y[i][k] *= Y[k][k] ;
               }

            /* Elementos que no estan en una fila o columna pivote */

              for (i=0;i<n;i++)
               {
                 if (i!=k)
                  {
                    for (j=0;j<n;j++)
                     {
                      if (j!=k)
                        {
                          Y[i][j] += Y[i][k]*Y[k][j] ;
                        }
                     }
                  }
```

```
            }

          /* Elementos en una fila pivote */

            for (i=0;i<n;i++)
             {
               if (i!=k)
                {
                  Y[k][i] *= Y[k][k] ;
                }
             }
        }

      /* Cambio de Signo */

        for (i=0;i<n;i++)
         {
           for (j=0;j<n;j++)
            {
             Y[i][j] *= -1 ;
            }
         }
    }

/**/void Genera_reporte(int n, char *nombre)
    {
      FILE  *arch ;
      int   k, m  ;

      arch = fopen(nombre,"wt") ;
      fprintf(arch,"*** Matriz de Impedancias ***\n\n") ;
      for (k=0;k<n;k++)
      {
        for (m=0;m<n;m++)
         fprintf (arch,"%7.3lf+j%7.3lf  ",real(Y[k][m]),imag(Y[k][m]))
        fprintf (arch,"\n") ;
      }
      fclose(arch) ;
    }
```

ANEXO I
CÓDIGO PROGRAMA *CALMF A.CPP*

PROGRAMA CALMF A.CPP

```
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <complex.h>
/* Definicion de constantes */

   const int MAX = 23 ;

/* Definicion de variables  */

   /* Archivo de texto */

      FILE     *archivo          //Manipula archivos de entrada y salida
      char     a_ent[30]   ;     //Nombre de archivo de entrada
      char     a_sal[30]  ;      //Nombre de archivo de salida
                          ;
   /* Datos de linea */

      int      i                 //Barra de envio
               j    ,            //Barra de recepcion
      double   r     ;           //Resistencia de linea
               x    ,            //Reactancia de linea
               ys   ,            //Admitancia shunt de linea (Yc/2)
                    ;
   /* Matriz de admitancias */

      complex  Y[MAX][MAX]       //Matriz de admitancias
      int      n ;          ;    //Dimension de Y (Tama¤o maximo = MAX)


/* Definicion de procedimientos y funciones */

   void Inicializa_matriz() ;
   void Factorizacion_LUDM(int)
   void Genera_reporte(int,char *)
                                  ;

void main()
 {

   /* Identificacion del archivos                        */

      clrscr() ;
      printf ("*** Calculo de la Matriz de Impedancias ***\n\n")
      printf ("* Archivo de entrada : ")                          ;
      scanf ("%s", &a_ent) ;              ;
      printf ("* Archivo de salida : ") ;
      scanf ("%s", &a_sal)
      printf ("\n\n")       ;
                          ;
   /* Procesamiento del archivo de entrada               */

      if ((archivo=fopen(a_ent,"rt"))==NULL)
       printf ("Archivo no encontrado ... \n\n")
      else                                              ;
       {
        printf ("Procesando archivo %s ...\n\n",a_ent)
                                                        ;
        Inicializa_matriz()
                          ;
        fscanf(archivo,"%d",&n)
        while(!feof(archivo))    ;
         {
           fscanf(archivo,"%d%d%lf%lf%lf",&i,&j,&r,&x,&ys)
                                                              ;
```

```
          if (i==j)
           Y[i-1][i-1] = Y[i-1][i-1] + 1/complex(r,x) + complex(0,ys) ;
          else
           {
             Y[i-1][i-1] = Y[i-1][i-1] + 1/complex(r,x) + complex(0,ys) ;
             Y[j-1][j-1] = Y[j-1][j-1] + 1/complex(r,x) + complex(0,ys) ;
             Y[i-1][j-1] = Y[j-1][i-1] = - 1/complex(r,x) ;
           }
        }

       Factorizacion_LUDM(n) ;

       Genera_reporte(n,a_sal) ;
      }

     fclose(archivo) ;

  /* Salida del programa                                    */

     printf ("\n\n") ;
     printf ("Presione cualquier tecla para salir ...") ;

     getch() ;

}

/**/void Inicializa_matriz()
    {
      int  k, m ;

      for (k=0;k<MAX;k++)
      for  (m=0;m<MAX;m++)  Y[k][m] = complex(0,0) ;
    }

/**/void Factorizacion_LUDM(int n)
    {
      int  i, j, k ;

      /* Factorizacion LUDM */

        for (k=0;k<n-1;k++)
         {
           for (i=k+1;i<n;i++)
            {
             Y[i][i] -= Y[k][i]*Y[k][i]/Y[k][k] ;

             for (j=i+1;j<n;j++)
              {
                Y[i][j] -= Y[k][i]*Y[k][j]/Y[k][k] ;
              }
            }
         }
    }
/**/void Genera_reporte(int n, char *nombre)
    {
      FILE  *arch ;
      int   k, m  ;
      arch = fopen(nombre,"wt") ;
      fprintf(arch,"*** Matriz de Impedancias Factorizada ***\n\n") ;
      for (k=0;k<n;k++)
      {
        for (m=0;m<n;m++)
         fprintf (arch,"%7.3lf+j%7.3lf  ",real(Y[k][m]),imag(Y[k][m])) ;
        fprintf (arch,"\n") ;
      }
       fclose(arch) ;
```

BIBLIOGRAFÍA

[01] Alvaro Acosta Montoya, "Análisis de Sistemas de Potencia", Universidad Tecnológica de Pereira, Colombia, 2001.

[02] Alvaro Acosta Montoya, "Formación de la Matriz de Admitancia de Nodos", Universidad Tecnológica de Pereira, Colombia, 2000.

[03] Allen J. Wood / Bruce F. Wollenberg, "Power Generation, Operation and Control", Second Edition, 1996.

[04] John J. Grainger / William D. Stevenson Jr., "Análisis de Sistemas de Potencia", Editorial McGraw Hill, 1994.

[05] Marciano Morozowski Filho, "Matrizes Esparsas em Redes de Potencia", Editora FundaÇao Ensino da ngenharia em Santa Catarina, 1981.

[06] Enriquez Harper, "Introducción Al Análisis de los Sistemas Eléctricos de Potencia", Editorial Limusa Wiley, 1971.

[07] William F. Tinney, "Direct Solutions of Sparse Network Equations by Optimally Ordered", Proceedings IEEE, November 1967.

[08] Harvey M. Deitel / Paul J. Deitel, "Cómo Programar en C++", Cuarta Edición, Editorial Pearson Education, 2003.

[09] Alberto Jaime Sisa, "Estructura de Datos y Algoritmos", Editorial Prentice Hall, 2002.

[10] Derek O´Connor, "Algorithms and Data Structures", University College Dublin, Department of Management Information Systems, 2000.

[11] Francisco Javier Zevallos, "Enciclopedia del Lenguaje C", Editorial Rama, 1997.

[12] Luis Joyanes Aguilar, "Algoritmos y Estructura de Datos", Segunda Edición, Editorial McGraw Hill, 1996.

[13] Aaron M. Tenenbaum / Yediyah Langsam / Moshe A. Augenstein, "Estructura de Datos en C", Editorial Prentice Hall, 1990.

[14] Herbert Schildt, "Turbo C/C++ Manual de Referencia", Editorial Osborne / McGraw Hill, 1990.

[15] Herbert Schildt, "Lenguaje C: Programación Avanzada", Editorial Osborne / McGraw Hill, 1986.