UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

"PROGRAMACIÓN AVANZADA Y MÉTODOS NUMÉRICOS"

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

P R E S E N T A N:

DE LUNA BONILLA VIRIDIANA DEL CARMEN

GREEN PEREZ VIRGILIO

DIRECTORA DE TESIS: ING. LAURA SANDOVAL MONTAÑO

CIUDAD UNIVERSITARIA, MÉXICO, D. F. 2007

AGRADECIMIENTOS

A mis padres, por apoyarme siempre en todo momento, por brindarme su cariño incondicional y sobretodo su paciencia, por el gran esfuerzo que hicieron por hacer de mí una persona preparada y de principios. Mamá, gracias por ser además de madre una excelente amiga, por estar siempre conmigo escuchándome y apoyándome cuando mas lo he necesitado. Papá, gracias por ser como eres, por tenerme siempre confianza y hacer hasta lo imposible por concretar esta meta.

A mi familia, por ser el regalo más lindo que me pudo haber brindado la vida. Ivette, gracias por apoyarme siempre y tratar de que nunca me hiciera falta nada, por ser una excelente hermana, amiga y consejera, eres un gran ejemplo para mí. Jorge, gracias por ser un excelente hermano, por cuidarme y defenderme desde que era pequeña, gracias por demostrarme siempre tu cariño y regalarme una sonrisa en todo momento. Oscar, agradezco a Dios el que me haya regalado un hermano menor, no sabes lo importante que eres para mí porque traes a mi vida esa chispita de alegría y felicidad, gracias por confiar siempre en mí. Osvaldo, gracias por el apoyo que siempre he recibido de parte tuya y por estar con nosotros en todo momento. Adrián, lo mejor que me pudo haber pasado es saberme tu tía porque eres toda ternura y llenas mi vida de alegría y felicidad.

A mi tío Manuel, por el apoyo recibido a lo largo de mi vida, pero sobretodo por brindarme todo su cariño y amor incondicional, gracias por estar conmigo siempre que lo he necesitado.

A Toño, por apoyarme siempre incondicionalmente, por haber contribuido de una u otra manera a la realización de este sueño, porque desde que te conocí te convertiste en una parte especial de mi vida. Gracias por todo lo que has hecho por mí, y sobre todo, por estar conmigo hasta el final.

A Virgilio, por ser parte importante en la realización de este trabajo, porque juntos aprendimos a superar los diferentes obstáculos que se nos presentaron, pudiendo hacer de este sueño una realidad. Gracias por ser además de compañero de trabajo un buen amigo.

A mis amigos, para a todos los que me han apoyado siempre, que han creído en mí y me han brindado su amistad incondicional. Adriana, gracias por apoyarme e impulsarme siempre a seguir adelante y por estar conmigo en los momentos más especiales. Eduardo, Jorge, Juan, Edgar, Osvaldo, Elba, Panchito, Eliseo me siento feliz de contar con su amistad. Mary, Alex, Jorge y Rachel gracias por todo el apoyo moral recibido.

A la Ing. Laura Sandoval Montaño, por todo el apoyo brindado y por contribuir en la realización de esta meta. Gracias por ser una excelente profesora, directora de tesis y excelente amiga, pero sobretodo por la confianza que siempre depositó en nosotros.

A la Universidad Nacional Autónoma de México, porque me permitió ser parte de su comunidad estudiantil, lo cual es mi mayor orgullo; porque gracias a ella pertenecí al CCH y posteriormente a la Facultad de Ingeniería.

A la Facultad de Ingeniería, por todo lo que hasta la fecha me ha brindado, por darme una excelente formación y por permitirme conocer a excelentes personas durante mi estancia en ella.

De Luna Bonilla Viridiana del Carmen

AGRADECIMIENTOS

Antes que nada quiero agradecer a mis padres por haberme aguantado toda la carrera, por haberme apoyado a lo largo de todo el tiempo que le dedique a la escuela. Por los consejos y los regaños que me ayudaron a no dejarme vencer y siempre seguir adelante. Y que todos mis logros, se los debo a ustedes por todo lo que me han enseñado de la vida.

A mis hermanas, por el apoyo recibido y por ser mi familia.

A la Facultad de Ingeniería, por toda la formación que recibí, no solo como ingeniero, sino como persona. Que siempre se debe estar comprometido con lo que se hace, y que debe ser en beneficio de la sociedad.

A la Ing. Laura Sandoval por el tiempo invertido y el apoyo brindado en la realización de este trabajo de tesis.

A Alinne por ser mi amiga desde hace ya muchos años y que sin su amistad no se donde estaría en este momento; por todas las aventuras por las que hemos pasado y que me hacen recordar el por qué somos los mejores amigos.

A Lizeth, por ser mí amiga desde el primer año de la preparatoria, y porque hemos tenido una amistad sincera ya por muchos años, y que espero siga así por muchos años más.

A Viridiana por haber hecho este sueño una realidad, y por todos los tropiezos a los que nos enfrentamos y que supimos resolver como equipo.

A toda la banda de la facultad (dígase Adriana, Eduardo, Elba, Eliseo, Erika, Jorge, Juan Pablo, Miriam, Osvaldo, y por último pero no el menos importante Ricardo), por ser tan buenos amigos y por ser tan unidos.

Green Pérez Virgilio

ÍNDICE

PREFACIO

La Facultad de Ingeniería de la Universidad Nacional Autónoma de México es una institución de educación superior la cual imparte varias carreras de ingeniería. Esta institución es generadora de profesionales capacitados tanto para el sector público como el privado. Es una de las mejores escuelas de ingeniería en nuestro país y con la finalidad de que ésta siga manteniendo el nivel que actualmente tiene, se realiza con frecuencia el proceso de revisión y actualización de los planes de estudio de todas las carreras que se imparten en ésta, siendo el último proceso concluido en 2005.

En lo que se refiere a la carrera de Ingeniería en Computación, el comité encargado de realizar dicho proceso buscó que el nuevo plan de estudios ajustara sus contenidos a los avances científicos y tecnológicos, ya que incluye asignaturas que comprenden temas de vanguardia de acuerdo a las necesidades requeridas para el desarrollo sustentable del país y con esto dar bases a los egresados para tener un nivel más alto de competitividad en el mercado laboral.

Además, con la intención de mejorar la estructura curricular, se llevaron a cabo varias modificaciones notables en este nuevo plan, siendo algunas de ellas: la reducción de 10 a 9 semestres, con la consecuente disminución de créditos (de 448 a 408), así como organizar las asignaturas con un mínimo de seriaciones obligatorias, y eliminar los cursos propedéuticos del plan de estudios anterior al incorporar los contenidos relevantes a asignaturas básicas curriculares. También se incluyen módulos terminales, los cuales van encaminados a alguna área del campo de trabajo de la ingeniería en computación, llevándola de manera estructurada y ordenada, con la finalidad de hacer especial énfasis en las áreas de mayor demanda laboral; logrando con ello que el alumno se especialice en el área acorde con sus intereses y/o vocación.

Es así que el nuevo plan de estudios de la carrera de Ingeniería en Computación permite que el egresado tenga conocimientos tanto generales como específicos al concluir sus estudios. Dicho plan fue aprobado por el Consejo Académico del Área de las Ciencias Físico Matemáticas y de las Ingenierías (CAACFMI) el día 11 de agosto de 2005.

El nuevo mapa curricular el cual se muestra en la figura 1, y podemos compararlo con el plan de estudios anterior, mostrado en la figura 2 donde observaremos más claramente las modificaciones realizadas.

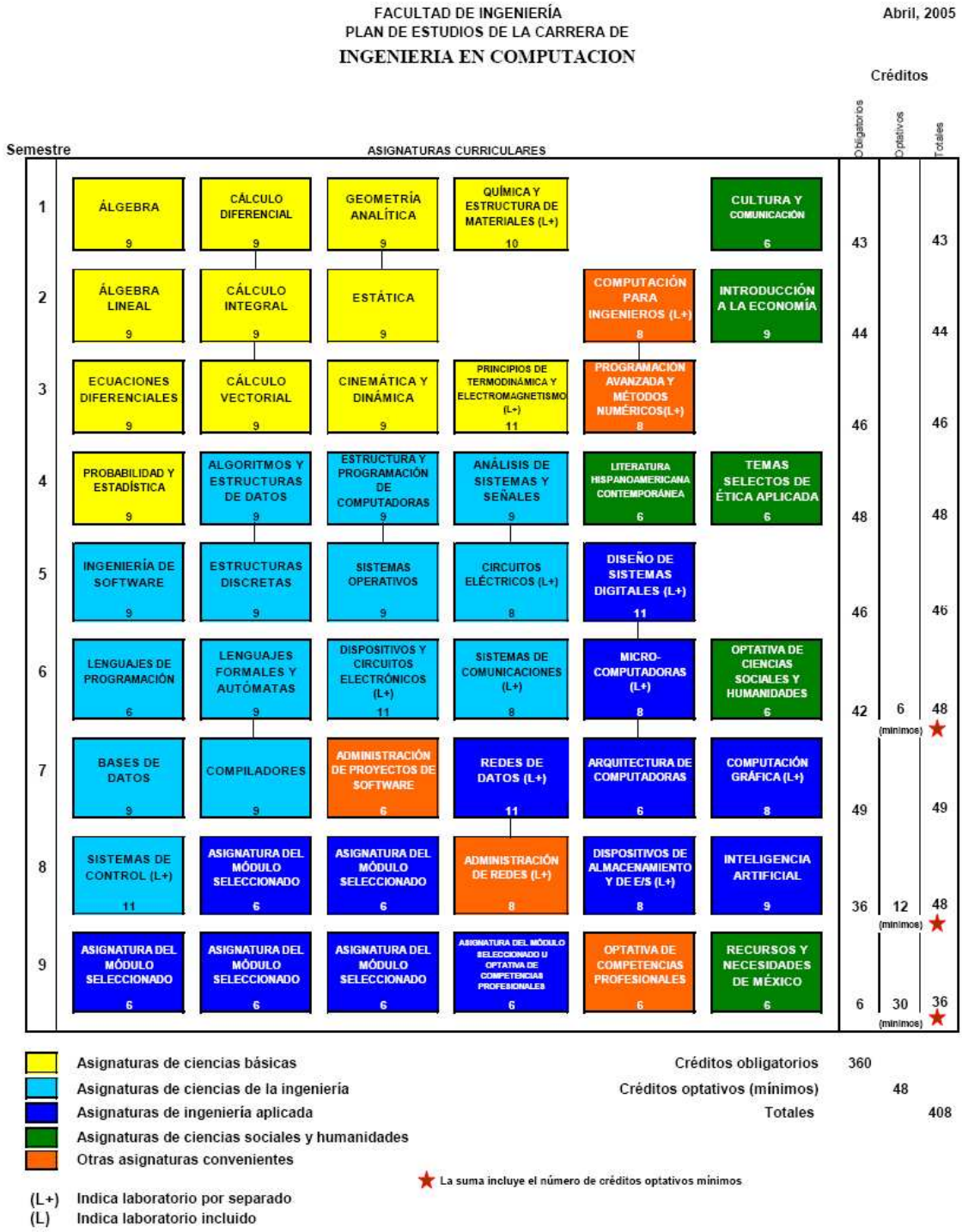


FIGURA 1. Mapa curricular vigente de la carrera Ingeniería en Computación

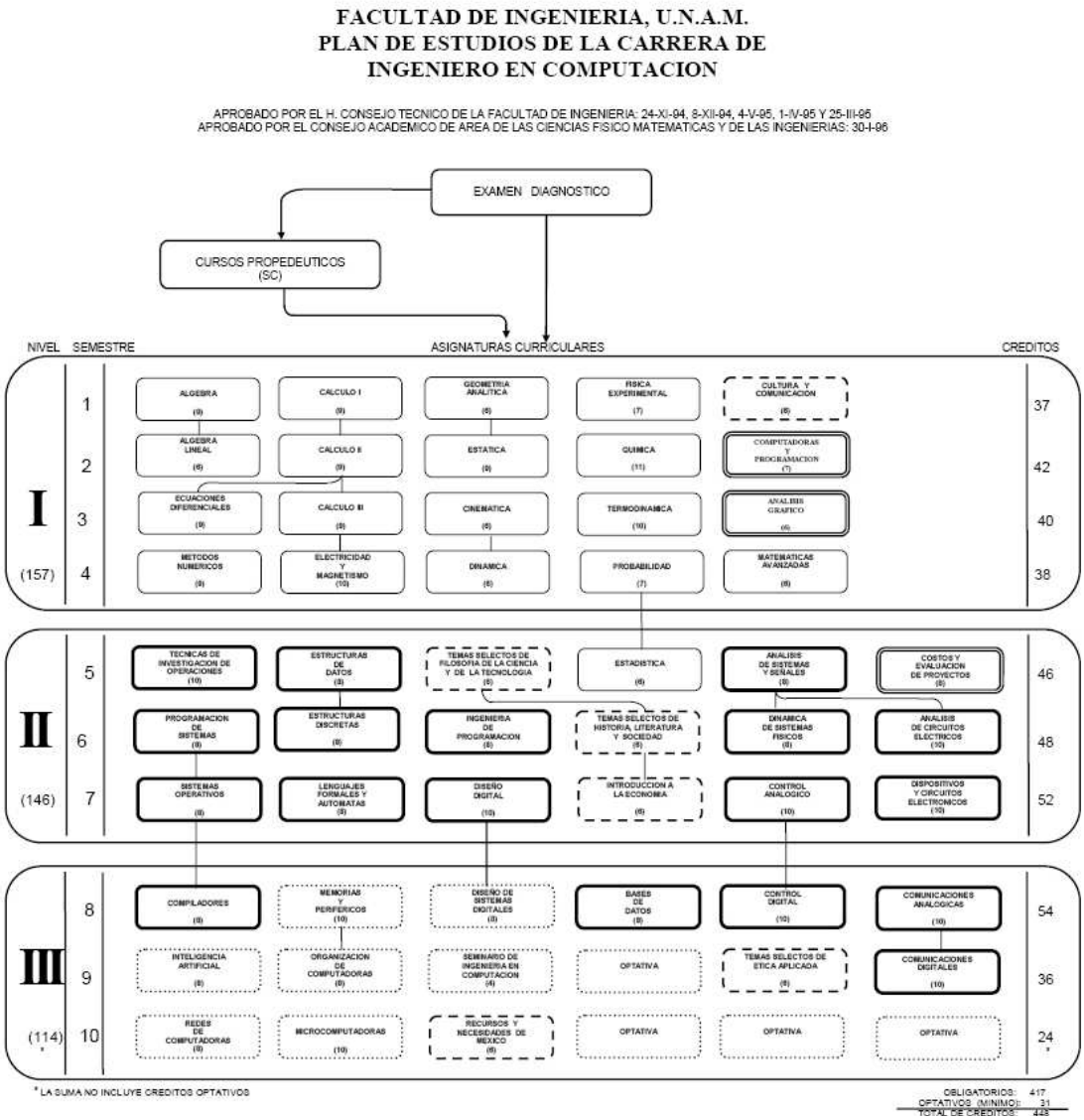


FIGURA 2. Mapa curricular anterior de la carrera Ingeniería en Computación

La computación en las ingenierías.

En la actualidad, muchas actividades se realizan con ayuda de una computadora, y se le da diferentes usos, desde la realización de escritos hasta para resolver problemas de diferentes disciplinas (matemáticas, contables, ingenieriles), sin dejar a un lado que también se utilizan para entretenimiento, como medio de comunicación (e-mail, voz, video), almacenamiento de datos, búsqueda de información, etc.

La utilidad que se le dé a una computadora define el tipo de usuario. Por ejemplo, los usuarios que utilizan la computadora sólo para realizar escritos, presentaciones, escuchar música, acceder a información, etc., son los llamados usuarios cotidianos. Otros que emplean cierto tipo de software que además de permitir el uso de aplicaciones también permite generar un producto utilizando programación básica como hojas de cálculo, manejadores de bases de datos, paquetes matemáticos, paquetes especializados, etc., son usuarios avanzados o programadores. Finalmente, a los usuarios que hacen uso

de lenguajes de programación y software aplicativo para resolver problemas específicos de un área, se les nombran desarrolladores de sistemas.

De a cuerdo con la clasificación de usuarios descrita en el párrafo anterior, se puede decir que los estudiantes de todas las carreras de ingeniería que ofrece la facultad, pertenecen al grupo de usuarios avanzados, los cuales requieren conocimientos mínimos necesarios de programación, entornos de las tecnologías de la información, manejo interno de datos y metodologías básicas de desarrollo de software. Dichos temas se incluyen en el programa de la asignatura Computación para Ingenieros, resultado de la revisión y actualización de la asignatura Computadoras y Programación.

De manera particular los estudiantes de las carreras de Ingeniería en Computación, Ingeniería Eléctrica Electrónica, Ingeniería en Telecomunicaciones e Ingeniería Geomática, se ubican en el tipo de usuarios desarrolladores de sistemas, debido a que requieren conocimientos más específicos de programación y aplicación de ésta en resolución de problemas matemáticos e ingenieriles para su buen desempeño tanto en su trayectoria académica como en su vida profesional.

Por tal motivo, se crea la asignatura Programación Avanzada y Métodos Numéricos, la cual se pensó para reforzar los conocimientos que se han adquirido en la asignatura precedente, es decir, Computación para Ingenieros. Adicionalmente, ésta incrementará los conocimientos de los alumnos en el desarrollo de programas, aplicando para ello el paradigma de programación orientada a objetos, así como podrán implementar diversas técnicas de análisis numérico en dos diferentes paradigmas de programación: el paradigma estructurado y el orientado a objetos.

Durante la fase de definición del contenido de dicha asignatura, por recomendación de la subcomisión del CAACFMI para la revisión de los proyectos de modificación a los planes y programas de estudios, se pidió que en la estructuración de los contenidos de la asignatura ***Programación Avanzada y Métodos Numéricos***, la cual está coordinada por el departamento de Ingeniería en Computación de la División de Ingeniería Eléctrica, se hicieran explícitos los temas de enseñanza de programación y del aprendizaje de los métodos numéricos a utilizar, razón por la cual se hizo necesario estructurarlo en seis temas, donde se abordaran de forma explícita temas de enseñanza de programación y otros referentes a la enseñanza de los métodos numéricos, conservando una relación estrecha entre ellas con el desarrollo de programas que implementen los métodos numéricos analizados ocupando el paradigma y lenguaje de programación estudiados en temas previos.

En el programa de la asignatura los métodos numéricos que se estudian son los suficientes y de una profundidad adecuada para el entendimiento y aplicación de solución numérica con ayuda de la computadora de problemas de las ingenierías correspondientes a las cuatro carreras en las que se imparte.

De acuerdo a la fuente de la Asociación Nacional de Instituciones de Educación en Informática A.C, ANIEI, “Modelos curriculares nivel licenciatura informática computación” los conocimientos mínimos necesarios que debe tener un ingeniero del área de las ingenierías eléctrica y geomática en cuanto a métodos numéricos se refiere son: Aritmética de punto flotante y aproximaciones. Teoría de errores. Resolución de sistemas de ecuaciones lineales. Resolución de ecuaciones algebraicas.

Es importante resaltar que el programa de la asignatura cubre con los temas mínimos indicados por la ANIEI en el rubro de métodos numéricos. Adicionalmente se incluyen métodos numéricos avanzados que propician un mejor dominio del tema.

En el programa de la asignatura, mostrado en la figura 3, se puede observar lo siguiente:

- La asignatura obligatoria antecedente es Computación para Ingenieros.
- El objetivo de la asignatura, el cual surge como resultado del análisis e investigación de los conocimientos mínimos requeridos para las carreras de Ingeniería en Computación, Ingeniería Eléctrica Electrónica, Ingeniería en Telecomunicaciones e Ingeniería Geomática.
- La asignatura se encuentra ubicada en el tercer semestre del mapa curricular de las cuatro carreras en que se imparte.
- Se encuentran de forma explícita los temas de enseñanza de programación y otros referentes a la enseñanza de los métodos numéricos.
- Finalmente, las horas de enseñanza tanto para Métodos Numéricos como para Programación se distribuyen equitativamente.

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE INGENIERÍA

PROGRAMA DE ESTUDIO

PROGRAMACIÓN AVANZADA Y MÉTODOS NUMÉRICOS	1312	3°	08
Asignatura	Clave	Semestre	Créditos

Ingeniería Eléctrica	Ingeniería en Computación	Ingeniería en Computación
División	Departamento	Carrera en que se imparte

Asignatura:		Horas:		Total (horas):	
Obligatoria	X	Teóricas	3.0	Semana	5.0
Optativa		Prácticas	2.0	16 Semanas	80.0

Aprobado:	Fecha:
Consejo Técnico de la Facultad	25 de febrero, 17 de marzo y 16 de junio de 2005
Consejo Académico del Área de las Ciencias Físico Matemáticas y de las Ingenierías	11 de agosto de 2005

Modalidad: Curso, laboratorio

Asignatura obligatoria antecedente: Computación para Ingenieros.

Asignatura obligatoria consecuente: Ninguna.

Objetivo(s) del curso:
El alumno empleará estructuras de almacenamiento de datos complejas para la resolución de problemas numéricos. Seleccionará y aplicará métodos numéricos para obtener soluciones aproximadas de modelos matemáticos que no se pueden resolver por métodos analíticos; y desarrollará programas tanto en lenguaje estructurado como orientado a objetos que implementen dichos métodos numéricos.

Temario

NÚM.	NOMBRE	HORAS
1.	Programación avanzada en lenguaje estructurado	6.0
2.	Aproximación numérica, errores y métodos numéricos iniciales	12.0
3.	Fundamentos de la programación orientada a objetos	6.0
4.	Programación orientada a objetos	6.0
5.	Métodos numéricos para solución de sistemas y ecuaciones avanzadas	12.0
6.	Programación orientada a objetos avanzada	6.0
		48.0
	Prácticas de laboratorio	32.0
	Total	80.0

FIGURA 3. Programa de la asignatura Programación Avanzada y Métodos Numéricos (1/5)

PROGRAMACIÓN AVANZADA Y MÉTODOS NUMÉRICOS (2 / 5)

1 Programación avanzada en lenguaje estructurado

Objetivo: El alumno elaborará programas en lenguaje C empleando estructuras de almacenamiento de datos complejas.

Contenido:

- **1.1** Arreglos de varias dimensiones y arreglos de apuntadores
- **1.2** Estructuras.
- **1.3** Archivos y bancos de datos
- **1.4** Desarrollo de programas simples con estructuras de almacenamiento complejas

2 Aproximación numérica, errores y métodos numéricos iniciales

Objetivo: El alumno aplicará los primeros métodos de solución numéricas considerando y minimizando los errores y la convergencia

Contenido:

- **2.1** Aproximación numérica y errores.
- **2.2** Solución numérica de ecuaciones algebraicas y trascendentes
- **2.3** Solución numérica de sistemas de ecuaciones lineales
- **2.4** Interpolación numérica
- **2.5** Desarrollo de programas en lenguaje estructurado para la implementación de los métodos de este tema

3 Fundamentos de la programación orientada a objetos

Objetivo: El alumno explicará los diferentes paradigmas de programación, así como los conceptos y diseño de la programación orientada a objetos en solución de problemas.

Contenido:

- **3.1** Paradigmas de programación: imperativa, funcional, lógica, declarativa y orientada a objetos
- **3.2** Conceptos manejados en la programación orientada a objetos: objetos, métodos, mensajes, clase, herencia, encapsulamiento, polimorfismo.
- **3.3** Diseño de programación orientada a objetos. Notación UML

4 Programación orientada a objetos

Objetivo: El alumno conocerá y aplicará las técnicas y herramientas de la programación orientada a objetos para la solución de problemas.

Contenido:

- **4.1** Teoría del Diseño de jerarquía de clases
- **4.2** Control de flujo.
  - **4.2.1** Sentencia if-else.
  - **4.2.2** Sentencia switch.
  - **4.2.3** Ciclo for.

FIGURA 3. Programa de la asignatura Programación Avanzada y Métodos Numéricos (2/5)

PROGRAMACIÓN AVANZADA Y MÉTODOS NUMÉRICOS (3 / 5)

- **4.2.4** Ciclo while y do-while.
- **4.3** Tipos de Clase.
  - **4.3.1** abstract, final, public, private.
  - **4.3.2** Métodos constructores.
  - **4.3.3** Interfaces.
- **4.4** Resolución de problemas matemáticos, físicos y químicos sencillos

5 Métodos numéricos para solución de sistemas y ecuaciones avanzadas

Objetivo: El alumno aplicará métodos numéricos para la solución numérica de sistemas y ecuaciones avanzadas.

Contenido:

- **5.1** Derivación e integración numérica
- **5.2** Solución numérica de ecuaciones y sistemas de ecuaciones diferenciales.
- **5.3** Solución de ecuaciones en derivadas parciales
- **5.4** Desarrollo de programas en lenguaje orientado a objetos para la implementación de los métodos de este tema

6 Programación orientada a objetos avanzada

Objetivo: El alumno aplicará los conceptos avanzados de la programación orientada a objetos para la resolución de problemas complejos.

Contenido:

- **6.1** Multihilos.
- **6.2** Flujos de Datos.
- **6.3** Resolución de problemas complejos

Bibliografía básica:	**Temas de la asignatura para los que se recomienda**
BURDEN, L. R. , FAIRES, J.D. *Análisis Numérico* 7ª. Edición México Thomson International, 2003	2 y 5
CHAPRA, Steven C, CANALE, Raymond P. Métodos Numéricos para Ingenieros 3ª. Edición México Mc. Graw-Hill, 1999	2 y 5

FIGURA 3. Programa de la asignatura Programación Avanzada y Métodos Numéricos (3/5)

PROGRAMACIÓN AVANZADA Y MÉTODOS NUMÉRICOS (3 / 5)

4.2.4 Ciclo while y do-while.
4.3 Tipos de Clase.
4.3.1 abstract, final, public, private.
4.3.2 Métodos constructores.
4.3.3 Interfaces.
4.4 Resolución de problemas matemáticos, físicos y químicos sencillos

5 Métodos numéricos para solución de sistemas y ecuaciones avanzadas

Objetivo: El alumno aplicará métodos numéricos para la solución numérica de sistemas y ecuaciones avanzadas.

Contenido:
5.1 Derivación e integración numérica
5.2 Solución numérica de ecuaciones y sistemas de ecuaciones diferenciales.
5.3 Solución de ecuaciones en derivadas parciales
5.4 Desarrollo de programas en lenguaje orientado a objetos para la implementación de los métodos de este tema

6 Programación orientada a objetos avanzada

Objetivo: El alumno aplicará los conceptos avanzados de la programación orientada a objetos para la resolución de problemas complejos.

Contenido:
6.1 Multihilos.
6.2 Flujos de Datos.
6.3 Resolución de problemas complejos

Bibliografía básica:	**Temas de la asignatura para los que se recomienda**
BURDEN, L. R. , FAIRES, J.D. *Análisis Numérico* 7ª. Edición México Thomson International, 2003	2 y 5
CHAPRA, Steven C, CANALE, Raymond P. Métodos Numéricos para Ingenieros 3ª. Edición México Mc. Graw-Hill, 1999	2 y 5

FIGURA 3. Programa de la asignatura Programación Avanzada y Métodos Numéricos (4/5)

PROGRAMACIÓN AVANZADA Y MÉTODOS NUMÉRICOS	(4 / 5)
DEITEL, Harvey M.; DEITEL, Paul J *C++ How to Program* 4th. Edition USA Prentice Hall., 2002	3, 4,5 y 6
DEITEL, HARVEY M.; DEITEL, PAUL J *Java How to Program* 5th. Edition Prentice Hall., 2002	3, 4,5 y 6
FELLEISEN, M.; FINDLET, R.B.; FLATT, M. *How to Design Class Hierarchies: an introducction to object-oriented programming.* Cambridge, USA MIT Press, 2004	3 y 4
GERALD, Curtis F. *Análisis Numérico* 6ª. Edición México Prentice Hall, 2001	2 y 5
KERNIGHAN,B.W.; RITCHIE, D.; RITCHIE, D.M. *C Programming Language* 2th. Edition USA Prentice Hall, 1988	1
PENDER, TOM *UML Bible* Indianapolis, IN. USA Wiley. 2003	3 y 4
Bibliografía complementaria:	
ROQUES, PASCAL *UML in Practice* England, Great Britain Jhon Wiley & Sons. 2004	3 y 4

FIGURA 3. Programa de la asignatura Programación Avanzada y Métodos Numéricos (5/5)

INTRODUCCIÓN

La programación estructurada así como la programación orientada a objetos, son paradigmas totalmente diferentes pero que al final llevan a un mismo resultado.

En el presente libro trabajaremos con estos paradigmas; explicaremos, ejemplificaremos y resolveremos problemas en los cuales se aplicarán los conocimientos adquiridos en cada uno de ellos. Al mismo tiempo se procederá a la enseñanza de métodos numéricos.

En el caso de la programación estructurada manejaremos ANSI C, el cuál es un lenguaje muy utilizado actualmente cuando se programa en C, debido a que su origen provino de ANSI (Instituto Americano Nacional de Estándares), este lenguaje es ejecutable en prácticamente todas las computadoras debido a que funciona como la base de todas las variaciones de C que existen actualmente como Turbo C, Borland C, etc.; mientras que para la programación orientada a objetos, nos enfocaremos al lenguaje JAVA. Este último, debido a que consideramos que es un lenguaje de programación que actualmente se encuentra ubicado a la cabeza de los demás, es decir, lo encontramos en pleno auge.

En el capítulo 1 de este libro abordaremos temas avanzados en cuanto a lenguaje estructurado se trate, veremos estructuras para almacenamiento de datos (arreglos, estructuras y archivos) enfatizando en sus funciones primordiales.

En el capítulo 2, empezamos a involucrarnos con el análisis numérico, aquí resolveremos problemas matemáticos (ecuaciones algebraicas y sistemas de ecuaciones, así como interpolación) de forma numérica, y con lo aprendido en el capítulo 1 los resolveremos utilizando programación estructurada.

En el capítulo 3, conoceremos diferentes paradigmas de programación y los lenguajes de programación que utilizan, además nos empezaremos a familiarizar con conceptos básicos de la programación orientada a objetos (objetos, clase, atributos, etc.) y finalizaremos con la enseñanza de notación UML, enfocada a la programación orientada a objetos.

En el capítulo 4, entraremos por completo al estudio de la programación orientada a objetos, empezaremos desde el diseño de la jerarquía de clases, hasta las características propias del lenguaje orientado a objetos, la estructura básica de un programa, el manejo de excepciones, los tipos de clase, y las interfaces, enfocado al lenguaje JAVA.

En el capítulo 5, ya habiendo comprendido ambos paradigmas (estructurado y orientado a objetos) retomaremos de nuevo métodos numéricos, pero en esta ocasión para ayudar a resolver problemas matemáticos más complejos, dichos problemas también los

resolveremos utilizando el paradigma orientado a objetos. A estas alturas podremos comparar la resolución de los problemas matemáticos con dos diferente enfoques.

Finalizamos el contenido de este libro con el capítulo 6, el cual contiene conceptos avanzados de la programación orientada a objetos (hilos, flujos de datos) para la resolución de problemas complejos.

Este libro se adecua al Programa de Estudio de la asignatura Programación Avanzada y Métodos Numéricos.

Es para nosotros un placer poder presentarles este trabajo ya que nuestro principal objetivo es que este libro sirva de apoyo para el estudio de la asignatura, esperando sea de suficiente ayuda para reforzar sus conocimientos.

CAPÍTULO 1

PROGRAMACIÓN AVANZADA EN LENGUAJE ESTRUCTURADO

Introducción

Dada la importancia del manejo correcto de datos, ya sean numéricos o alfanuméricos, para resolver problemas, es necesario que el programador haga uso adecuado de los diferentes elementos con los que cuenta un lenguaje de programación para su manipulación. Por esto, en este capítulo, se abordarán temas avanzados del lenguaje estructurado ANSI C, donde se describirán las estructuras para almacenamiento de datos como lo son: los arreglos, las estructuras y los archivos. Se explicarán las principales funciones que realizan cada una de estas estructuras, presentando de esta manera ejemplos comprensibles e ilustrativos.

1.1 Arreglos de varias dimensiones y arreglos de apuntadores

Antes de entrar al tema, es conveniente conocer cómo es el proceso de ejecución de un programa y los elementos que están interrelacionados en dicho proceso.

Memoria Principal

Cuando un programa está en ejecución, el sistema operativo, le asigna memoria principal donde almacena tanto el programa como los datos necesarios para su ejecución; de esta forma, la memoria asignada al programa para su ejecución se divide en dos partes: memoria de programa y memoria de dato.

En la memoria de programa se encuentran las instrucciones a ejecutarse.

En la memoria de datos, se almacenan dos conjuntos de datos:

- Los datos que requiere el sistema operativo para administrar la ejecución del programa, y
- Los datos que utiliza el programa, es decir las variables definidas por el usuario dentro del programa.

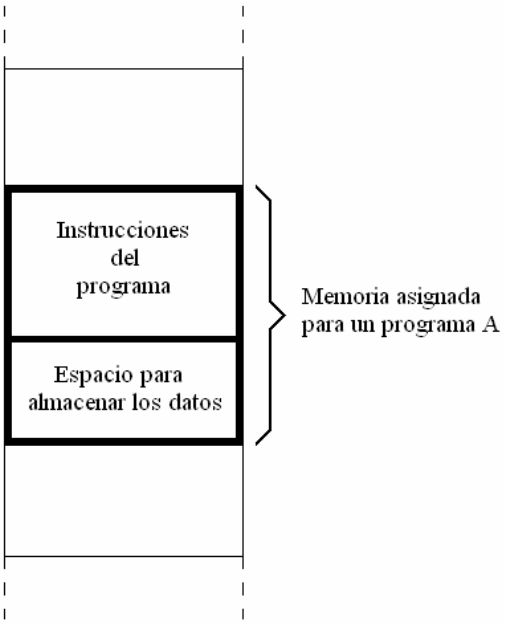


FIGURA 1.1 Esquema de asignación de memoria principal, para un programa A en ejecución.

Localidad de memoria

Comúnmente, cuando se acceden a los datos almacenados en memoria, se hace a través del nombre de una variable; de esta forma no es necesario conocer la ubicación exacta de dicho dato dentro de la memoria.

Sin embargo, para el mejor manejo de los datos en memoria, es indispensable conocer la forma en que ésta está organizada.

La memoria principal puede ser vista como un conjunto de *celdas* donde se almacenan *bytes*. Así, un *byte* es la mínima unidad de información que se almacena en memoria. A estas celdas se les denominan *localidades de memoria*.

Cuando se define una variable, dentro de un programa, al momento de ejecución del programa, al nombre de dicha variable se le asocia una o varias localidades de memoria; esto dependerá del tipo de dato que contenga la variable.

Dirección de Memoria

Las localidades de memoria se enumeran consecutivamente empezando desde cero, a esta numeración se le denomina dirección de memoria. Por lo tanto, cuando se asocia el nombre de una variable a una localidad de memoria, más bien, se asigna un nombre a dicha dirección.

Generalmente, la dirección de memoria se trabaja con números en base 16 (hexadecimal).

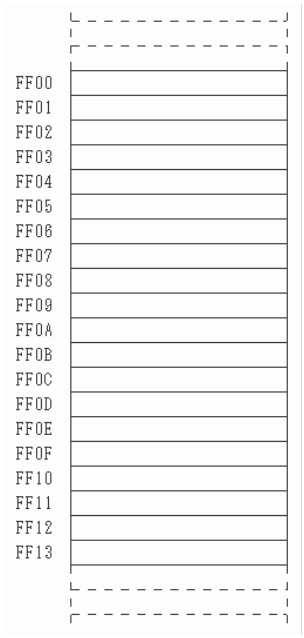


FIGURA 1.2 Mapa de memoria, indicando la dirección de las localidades de memoria; en este caso empieza en FF00.

Concepto de Arreglo

Un arreglo, es un conjunto de datos del mismo tipo, que están organizados secuencialmente en memoria principal. Es decir, que cuando se ejecuta un programa en lenguaje estructurado, por ejemplo C, donde se define un arreglo de *n* elementos, el sistema asigna en memoria principal, tantas localidades como sean necesarias, para almacenar todos los elementos del arreglo. Todas las localidades asignadas para almacenar los elementos de un arreglo son contiguas.

Dimensión de un Arreglo

Arreglos unidimensionales o de una dimensión

Los arreglos unidimensionales, pueden ser vistos como listas. Así, una lista contiene datos homogéneos como lo son las calificaciones de los exámenes de un estudiante. Éstas se pueden almacenar secuencialmente y en el orden en que fueron aplicados. Si este caso se proyecta a un arreglo, y suponiendo que fueron 10 exámenes a lo largo de un periodo, el arreglo se visualizaría como lo muestra la figura 1.3.

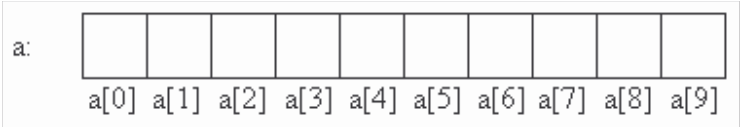


FIGURA 1.3 Vista conceptual de un arreglo unidimensional de 10 elementos.

Se puede observar que el arreglo unidimensional, es únicamente una serie de elementos que guardan una posición entre sí; las posiciones se enumeran partiendo del cero. Por lo tanto, si un estudiante obtiene en sus exámenes 9, 6, 7, 4, 8, 8, 10, 9, 8 y 8, el primer 9 va en la posición 0, el 6 en la posición 1, y así sucesivamente. Generalmente al indicador de la posición se le denomina *subíndice*.

Utilizando un mapa de memoria, que inicia en la dirección FF00, el arreglo unidimensional es almacenado como se muestra en la figura 1.4.

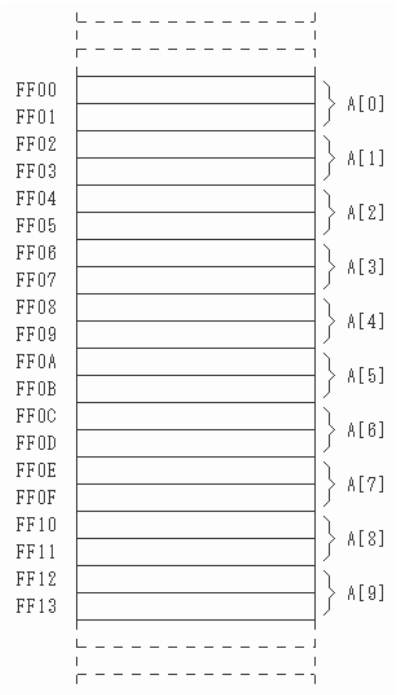


FIGURA 1.4 Mapa de memoria para un arreglo A de 10 elementos enteros. Se supone que el tamaño de un tipo entero es de 2 localidades de memoria.

Se ve claramente que los elementos del arreglo están situados, como lo indica la figura 1.3, en localidades de memoria contiguas.

Arreglos bidimensionales

Un arreglo de dos dimensiones, puede ser visto como una tabla que está compuesta por columnas y renglones. Tomando el ejemplo anterior, de las calificaciones de un estudiante, si se tuvieran 20 estudiantes, es poco práctico utilizar 20 arreglos unidimensionales, en cambio se puede utilizar un arreglo bidimensional, donde en cada renglón se encuentran las calificaciones de un estudiante en particular.

Por ejemplo, un arreglo bidimensional de 3 x 4 elementos, es decir, de 3 renglones y 4 columnas, tiene como vista conceptual la que se muestra en la figura 1.5.

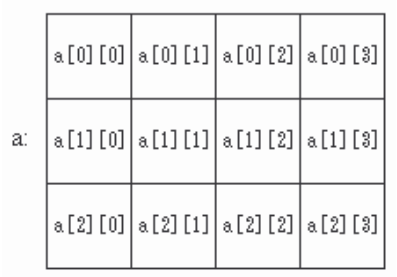


FIGURA 1.5 Vista conceptual de un arreglo bidimensional de 3 x 4 elementos.

Como se puede observar, el arreglo bidimensional es similar a una matriz de n x m, donde se utilizan dos subíndices para hacer referencia a la posición de un elemento del arreglo; el primer subíndice, es el indicador del renglón, y el segundo es para la columna. De igual forma que en los arreglos unidimensionales, los subíndices inician con el valor de 0.

Utilizando un mapa de memoria, que inicia en la dirección FF00, los elementos de este arreglo bidimensional se almacenarían como se muestra en la figura 1.6.

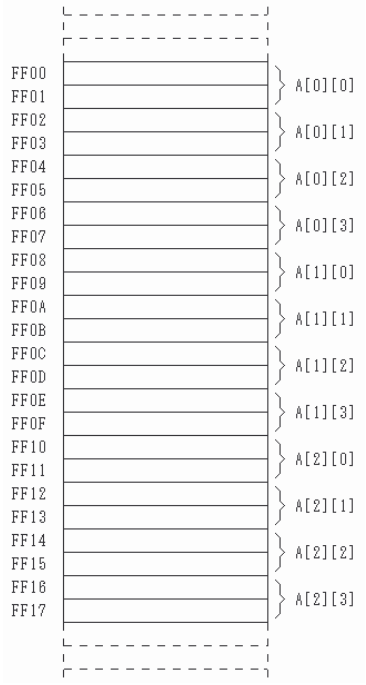


FIGURA 1.6 Mapa de memoria para un arreglo A de dos dimensiones con elementos enteros, de orden 3x4

Cuando se define un arreglo de 2 dimensiones, las localidades de memoria que se usan para almacenar los datos del arreglo, siguen estando contiguas, en este caso, acomodadas por renglones o filas; es decir, primero todos los datos del renglón 0, luego los del renglón 1, y así sucesivamente.

Arreglos de tres dimensiones

Un arreglo de tres dimensiones, puede ser visto como un conjunto de tablas. Si se toma el ejemplo anterior de las calificaciones de varios estudiantes, si un profesor además tiene varios grupos, podría trabajar con una tabla por grupo. Sin embargo, sería mejor agrupar, en un solo arreglo, todas las tablas haciendo uso de una tercera dimensión.

Por ejemplo, si se define un arreglo de 3 dimensiones de 2 x 3 x 4 elementos, es decir, un conjunto de 4 tablas con dos renglones y tres columnas cada una, tiene como vista conceptual la que se muestra en la figura 1.7.

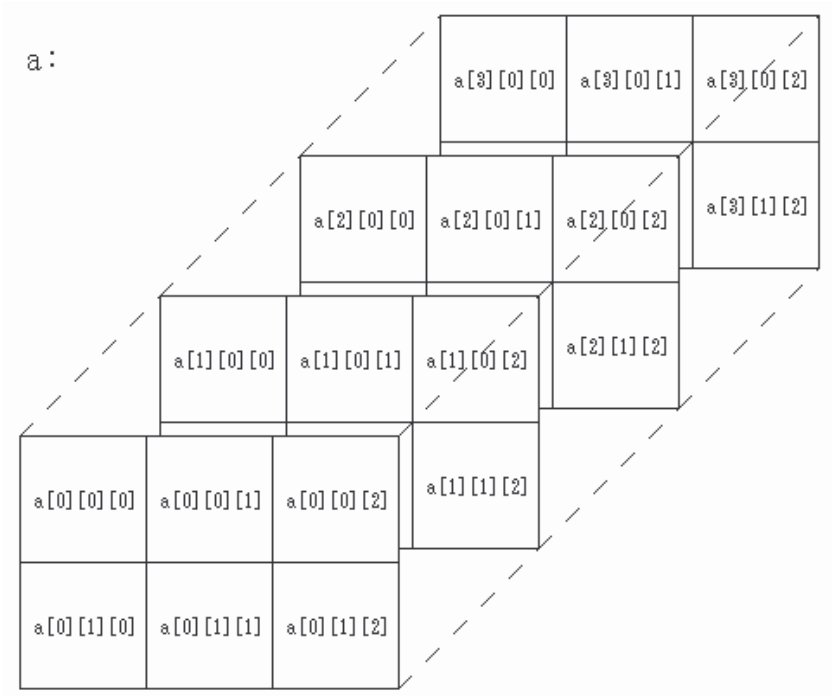


FIGURA 1.7 Vista conceptual de un arreglo tridimensional de 2 x 3 x 4 elementos.

Como se puede observar, un arreglo de 3 dimensiones, ya se puede redefinir como un arreglo de arreglos bidimensionales, debido a que se forman “capas”; en el ejemplo de los estudiantes, cada “capa” sería el grupo, entonces se tendrían los grupos 0, 1, 2 y 3.

En resumen, los arreglos son una colección de variables o datos del mismo tipo a los que se hace referencia utilizando un nombre en común y la posición. Como se mencionó

anteriormente, un arreglo ocupa un conjunto de localidades de memoria contigua, en la que la dirección más baja corresponde al primer elemento y la más alta al último elemento.

Los arreglos en 'C'

En lenguaje C, para declarar un arreglo, se necesita definir el tipo de dato que almacenará, así como un nombre de variable y cuántos elementos como máximo almacenará en cada dimensión. Es decir:

tipo nombre_arr [#elem dimensión1][#elem dimensión2]…[#elem dimensiónN];

Por ejemplo, para definir un arreglo unidimensional de 10 elementos, de tipo *int,* y que se llame *listanum*, se deberá escribir la siguiente sentencia:

```
int listanum[10];
```

Para hacer referencia a un elemento en particular dentro del arreglo, únicamente se debe indicar la posición del elemento menos una posición; esto se debe a que el primer elemento no es `listanum[1]`, sino `listanum[0]`. Por tanto, el ejemplo anterior declara un arreglo de enteros con diez elementos, es decir, desde `listanum[0]` hasta `listanum[9]`.

Por ejemplo, se quiere almacenar dentro del arreglo anterior el valor de 15 como tercer elemento, y a su vez, asignar el contenido del tercer elemento del arreglo, en otra variable de tipo `int` llamada `num`. Se deberán escribir las siguientes 2 líneas:

```
listanum[2]=15; /* Asigna 15 al 3er elemento del arreglo */
num=listanum[2]; /* Asigna el valor del 3er elemento a num */
```

Ahora, para definir un arreglo bidimensional de tipo *int*, de 6 filas y 8 columnas, se deberá escribir lo siguiente:

```
int tabladenums[6][8];
```

Si se desea acceder a los elementos, se procede de forma similar que con un arreglo unidimensional, esto es, se usa el nombre del arreglo y se indica la posición del elemento al que se desea acceder.

```
tabladenums[2][3]=15; /* Asigna 15 al elemento de la 3ª fila
                         y la 4ª columna*/
num=tabladenums[2][3]; /*Se asigna el valor del elemento de
                         la 3ª fila y la 4ª columna a num */
```

Algo que es importante mencionar, es que el lenguaje C no realiza una comprobación de contornos en los arreglos. En el caso de que sobrepase el final durante una operación de asignación, entonces se asignarán valores a otra variable o a un trozo del código. Esto es, si

se dimensiona un arreglo de tamaño *N*, se puede referenciar el arreglo por encima de *N* sin provocar ningún mensaje de error en tiempo de compilación o ejecución, incluso aunque probablemente se provoque el fallo del programa. El programador es el responsable del buen manejo de los arreglos para evitar acceder a localidades de memoria que no le corresponden.

EJEMPLO 1.1 OBTENER EL PROMEDIO GENERAL DE LOS ALUMNOS DE UNA CLASE. DESPLEGAR EN PANTALLA LAS CALIFICACIONES LEÍDAS, Y EL PROMEDIO CALCULADO. UTILIZAR ARREGLOS.

- Análisis y solución del problema

Primero se debe definir un arreglo de tipo *float* unidimensional de 15 elementos, para almacenar calificaciones; una variable temporal que almacene la suma de las 15 calificaciones, y una variable que almacene el promedio.

Se necesita un ciclo de lectura de las calificaciones desde teclado, y otro para hacer la suma de las calificaciones.

Una vez hecha la suma, se divide entre 15 para obtener el promedio.

Y por último, un ciclo para desplegar en pantalla las calificaciones leídas.

Es necesario definir una variable de tipo *int*, que ayude en la ejecución de los ciclos.

-Explicación de las variables

calif: Arreglo de tipo real.
suma: Variable de tipo real. Se utiliza para almacenar la suma de las calificaciones.
promedio: Variable de tipo real. Se utiliza para almacenar el promedio de las calificaciones.
i: Variable de tipo entero. Se utiliza como variable de control de ciclos y como índice del arreglo.

-Pseudocódigo

1. Repetir con i desde 0 hasta 14
   Escribir “Ingrese calificación”, i
   Leer calif[i]
   Hacer suma ← suma + calif [i]
   Hacer i ← i + 1
2. {Fin de ciclo paso 1}
3. Hacer promedio ← suma / 15
4. Repetir con i desde 0 hasta 14
   Escribir "Calificación",i,": ",calif [i]
   Hacer i ← i + 1
5. Escribir "Promedio general: ",promedio

-Codigo

```
#include<stdio.h>

void main(){
   float calif[15];
   float suma;
   float promedio;
   int i;
   suma=0; /*Asignación de valor inicial*/

   /*Ciclo de obtención y suma de calificaciones*/
   for(i=0;i<15;i++){
      printf("Dame la calificacion %d: ",i);
      scanf("%g",&calif[i]);
      suma=suma+calif[i];
   }
   promedio=suma/15; /*obtención de promedio*/
   /*Impresión de calificaciones*/
   for(i=0;i<15;i++){
      printf("Calificación %d: %.2f\n”,i,calif[i]);
   }
   printf("El promedio general del grupo es de
         %.2f\n\n",promedio);
}
```

EJEMPLO 1.2 ALMACENAR EN UNA MATRIZ DE 3 X 3 VALORES LEÍDOS DESDE EL TECLADO. LA MATRIZ DEBERÁ SER DE TIPO *FLOAT*. AL TÉRMINO DE LA LECTURA DE VALORES SE DEBERÁ DESPLEGAR EN PANTALLA LA MATRIZ LEÍDA.

-Análisis y solución del problema

Se requiere declarar un arreglo, que como se menciona en el enunciado, de tipo *float* de dos dimensiones 3 x 3 elementos.

Para simplicidad en el posicionamiento del elemento dentro del arreglo, se deberán declarar dos contadores de tipo *int*, uno para la posición por filas y otro para las columnas.

-Explicación de las variables

A: Arreglo bidimensional de tipo real.
fila: Variable de tipo entero, utilizada tanto para control de ciclos como índice del arreglo.
columna: Variable de tipo entero, utilizada como variable de control de ciclos y como índice del arreglo.

-Pseudocódigo

1. Repetir con fila desde 0 hasta 2
   1.1. Repetir con columna desde 0 hasta 2
        Escribir “Valor de A[][]”,fila, columna
        Leer A[fila][columna]
        Hacer columna ← columna + 1
   1.2. {Fin de ciclo paso 1.1}
   1.3. Hacer fila ← fila + 1
2. {Fin de ciclo paso 1}
3. Repetir con fila desde 0 hasta 2
   3.1 Repetir con columna desde 0 hasta 2
        Escribir A[fila][columna]
        Hacer columna ← columna + 1
   3.2 {Fin de ciclo paso 3.1}
   3.3 Hacer fila ← fila + 1
4. {Fin de ciclo paso 3}

-Código

```
#include <stdio.h>
main(){
 float A[3][3];
 int fila, columna;
 printf("\nEjemplo de arreglo bidimensional\n\n");
 /*ciclos de lectura*/
 for(fila=0; fila < 3; fila++){
   for(columna=0; columna<3; columna++){
    printf("Teclea el valor de A(%d,%d): ",fila,columna);
    scanf("%f",&A[fila][columna]);
   }
 }
  printf("\La matriz A es: \n");
 /*ciclos de impresión*/
 for(fila=0; fila < 3; fila++){
   for(columna=0; columna<3; columna++){
    printf("%.1f\t",A[fila][columna]);
   }
   printf("\n");
 }
}
```

1.1.1 Arreglos y apuntadores

En el lenguaje de programación C existe una estrecha relación entre arreglos y apuntadores. Es por ello que en esta sección se explicará desde el concepto de apuntador hasta su uso en el manejo de arreglos.

Los *apuntadores* son simplemente otra manera de acceder a los datos almacenados en la memoria de la computadora, proporciona además una manera flexible y/o accesible de manejar los datos dentro de un programa. Para comprender el concepto de *apuntador* es necesario haber entendido perfectamente el concepto de memoria que se explicó al inicio de este capítulo.

Es importante tener presente que la memoria principal está formada por *localidades* y que cada una de éstas tienen asociada una *dirección*. Las direcciones de memoria se enumeran secuencialmente desde cero hasta el tamaño de la memoria, generalmente esta numeración está dada en base 16 o hexadecimal.

En sí, un *apuntador* o *puntero* es una variable que contiene la dirección de una variable o dato. Para definir una variable apuntador se le antepone al nombre un * y se le indica el tipo de dato al que "apuntará". De esta forma las siguientes sentencias declarativas definen apuntadores a diferentes tipos de datos:

```
char *pa;     /* pa es un apuntador a un carácter */
int  *pi;     /* pi es un apuntador a un entero */
float *pf;    /* pf es un apuntador a un real */
```

Por ejemplo, si se define una variable de tipo carácter llamada `a` y un apuntador a carácter llamado `p`, como se presenta en la siguiente sentencia declarativa:

```
char  a, *p;
```

el diagrama de memoria podría ser el que se muestra en la figura 1.8.

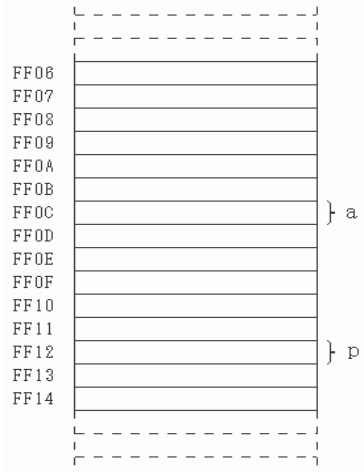


FIGURA 1.8 Mapa de memoria con 2 variables.

Ahora, si se ejecutan las siguientes instrucciones:

```
a = 'c';
p = &a;
```

El mapa de memoria cambia a como se muestra en la figura 1.9.

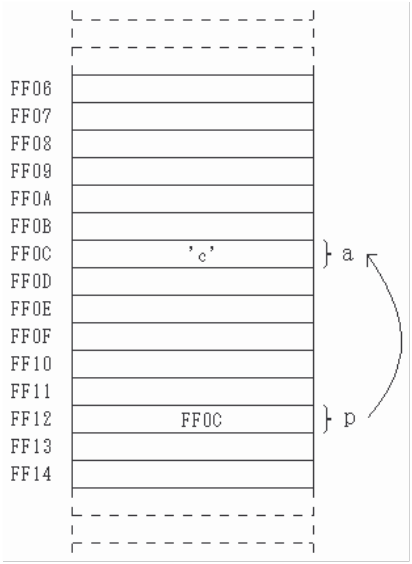


FIGURA 1.9 Mapa de memoria con 2 variables, una vez que se les asignan valores.

Como se puede observar, en la localidad de memoria asignada para `a`, se almacena el carácter `'c'`, mientras que en la localidad de memoria asignada para `p` se almacena la dirección de `a`. Esto se debe a que el '&' es un operador que equivale a "devuelve la dirección de:". En este caso, el operador se aplica a `a`, por lo que el operador & devuelve la dirección de `a`.

Otro operador en el manejo de apuntadores es el '*', el cual hace referencia al valor almacenado en la localidad de memoria contenida por el apuntador. Del ejemplo, si ahora se ejecuta la siguiente asignación:

```
*p = 'd';
```

la localidad de `a` adquiere el valor del carácter `d`

Es equivalente a haber hecho:

```
a = 'd';
```

ya que `p` contiene la dirección de `a`

Cuando se usan apuntadores, es importante que exista igualdad del tipo de variable o dato con el apuntador que les va a hacer referencia. Por ejemplo, si se declara un apuntador a entero, éste sólo podrá contener como valor direcciones de memoria en donde se almacenen

enteros. Esto se debe a que al hacer referencia a una variable por medio de un apuntador, éste hará referencia a tantas localidades de memoria para obtener correctamente el dato ya que cada tipo de dato ocupa un número diferente de bytes o localidades de memoria. Si se tuviera una variable de tipo entera, suponiendo que un entero se almacena en dos localidades, y un apuntador de tipo carácter, suponiendo que un carácter se almacena en una localidad, cuando se haga referencia al valor de la variable por medio del apuntador, sólo se consideraría una de las dos localidades del entero.

El siguiente ejemplo muestra de manera clara el uso de un apuntador:

EJEMPLO 1.3 PROGRAMA QUE DEFINE UN APUNTADOR DE TIPO *CHAR* Y UNA VARIABLE DE TIPO *CHAR*. Y UTILIZANDO DIFERENTES FORMAS DE IMPRESIÓN, SE DESPLIEGA EN PANTALLA EL VALOR DEL APUNTADOR, LA DIRECCIÓN DEL APUNTADOR, LA DIRECCIÓN DE LA VARIABLE, Y EL CONTENIDO DE LA VARIABLE USANDO EL APUNTADOR Y EL NOMBRE DE LA VARIABLE.

-Explicación de variables

valor: Variable de tipo *char*.
p: Variable de tipo apuntador a *char*.

-Código

```
#include <stdio.h>
main(){
     char valor;
     char *p;

     valor = 97; /*Asignación de valor inicial*/

     /*Impresión de la dirección con el formato %X, y el
     contenido de valor*/
     printf("\n %X => | %d | <= dirección y datos de valor.
            \n", &valor, valor);

     p = &valor; /*Asignación de valor inicial*/
     /*Impresión de la dirección con el formato %X, y el
     contenido, con el formato %X, de p*/
     printf(" %X => | %X | <= dirección y datos del
     apuntador. \n", &p,p);
     printf("\n Valor almacenado en apuntador = %X \n", p);
     printf(" Dirección de apuntador : &p = %X\n", &p);
     printf(" Valor referenciado por apuntador: *p = %d\n",
            *p);
}
```

El programa anterior muestra una salida como la siguiente:

```
FFF5 => | 97 | <= dirección y datos de valor.
FFF2 => | FFF5 | <= dirección y datos del apuntador

Valor almacenado en apuntador = FFF5
Dirección de puntero : &p = FFF2
Valor referenciado por apuntador: *p = 97
```

-Análisis del código

El programa inicia con la declaración de dos variables: `valor` de tipo carácter y `p` de tipo apuntador a carácter.

```
char valor;
char *p;
```

Después se le asigna el valor 97 a la variable `valor`.

```
valor=97;
```

Siendo la variable `valor` de tipo carácter, al asignarle el 97 es equivalente a asignarle el carácter `'a'` de acuerdo al código ASCII, debido a que la información que se guarda en memoria es en forma numérica, aún sean caracteres.

Luego de la asignación, se manda desplegar en pantalla tanto la dirección de la localidad de memoria asignada como el contenido de `valor`; con el formato %X utilizado en ANSI C, se puede desplegar la dirección en números base 16.

```
printf("\n %X => | %d | <= dirección y datos de valor. \n",
        &valor, valor);
```

Como se puede observar, cuando se manda a desplegar en pantalla la dirección de una variable se utiliza el operador “&”, el cual se utiliza como “La dirección en memoria de:”; así, cuando se escribe `&valor`, lo que se desea desplegar es *la dirección en memoria de: valor*.

La forma en que se manipulan los apuntadores, por ser también variables, se hace similarmente a las variables comunes. Por ejemplo en la sentencia:

```
p = &valor;
```

se le asigna el valor de la dirección de `valor` a `p`.

Como se mencionó anteriormente, otro operador importante en el manejo de apuntadores, es el asterisco "*", el cual indica "El contenido de la dirección apuntada por `p`". En la siguiente línea:

```
printf(" Valor referenciado por apuntador: *p = %d\n", *p);
```

se manda a desplegar en pantalla el valor de `valor`, utilizando el apuntador `p`, en el cual se ha almacenado la dirección en memoria de `valor`.

EJEMPLO 1.4 MODIFICAR VARIAS VECES EL VALOR DE UNA VARIABLE, MEDIANTE EL USO DE UN APUNTADOR.

-Explicación de variables

var: Variable de tipo *int*.
p: Variable de tipo apuntador a *int*.
i: Variable de tipo entero. Se utiliza como variable de control de ciclos.

-Código

```
#include<stdio.h>

void main(){
   int var,*p,i;
   var=0; /*Asignación de valor inicial*/
   p=&var; /*Asignación de valor inicial*/
   printf("El valor inicial de var es: %d\n",var);
   /*Ciclo de lectura y modificación de var*/
   for(i=0;i<6;i++){
      printf("nuevo valor: ");
      scanf("%d",p); /*Lectura de valor y asignación a var,
           usando el apuntador p*/
      printf("var ha cambiado por: %d\n",var);
   }
}
```

-Análisis del código

Con la sentencia

```
var=0;
```

se asigna inicialmente el valor de cero a `var`, ya que C no da valores iniciales a la variables por omisión.

Se asigna la dirección de la localidad de memoria de `var`, al apuntador `p` con la siguiente instrucción:

```
p=&var;
```

Cuando inicia el ciclo, se hace la lectura del nuevo valor a asignar en `var`, utilizando el apuntador `p`

```
scanf("%d",p);
```

En esta instrucción se puede observar que el segundo argumento de la función `scanf`, es el apuntador `p`, y no se utiliza el operador `&`, debido a que dicha función requiere que se le pase como parámetro una dirección, y el contenido de `p` es una dirección: la dirección de `var`, que fue asignada por la sentencia

```
p=&var;
```

Por último se comprueba que se haya modificado correctamente el valor de `var` con la siguiente instrucción.

```
printf("var ha cambiado por: %d\n",var);
```

1.1.2 Relación entre arreglo y apuntador

Hay una estrecha relación entre los apuntadores y los arreglos, se pueden direccionar arreglos como si fueran apuntadores y apuntadores como si fueran arreglos. En el lenguaje de programación C, el nombre de un arreglo es el índice a la dirección de comienzo del arreglo, más específicamente, el nombre de un arreglo es un apuntador al arreglo. Por ejemplo:

Si se declara un arreglo de la siguiente forma:

int a [5]={2,4,6,8,10};

entonces

a[0]= 2, a[1]=4, a[2]=6, a[3]=8, a[4]=10

El almacenamiento de dichos elementos pudieran visualizarse como lo muestra el mapa de memoria de la figura 1.10, considerando que un entero ocupa 2 bytes.

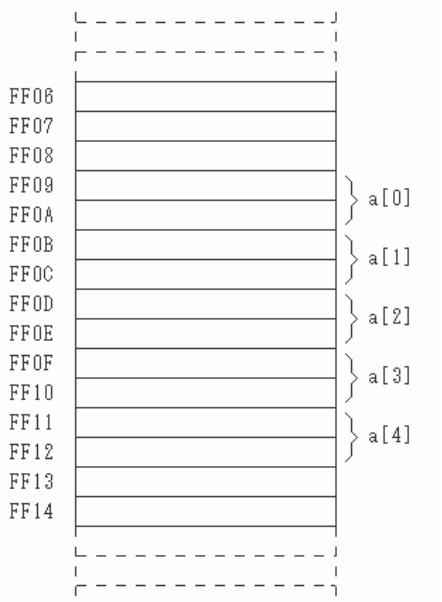


FIGURA 1.10 Mapa de memoria de un arreglo de tipo entero.

Para hacer referencia al primer elemento del arreglo, utilizando subíndices, se realiza con la notación: a [0]. Pero también se puede hacer referencia a este mismo elemento con la notación: *a, ya que, como se mencionó, el nombre de un arreglo es un apuntador.

Ahora, si se desea hacer referencia a otros elementos del arreglo, utilizando únicamente el nombre como apuntador, sólo hay que sumar posiciones, como lo muestra la siguiente tabla:

a + 0	apunta a	a[0]	que corresponde a la dirección FF09
a + 1	apunta a	a[1]	que corresponde a la dirección FF0B
a + 2	apunta a	a[2]	que corresponde a la dirección FF0D
a + 3	apunta a	a[3]	que corresponde a la dirección FF0F
a + 4	apunta a	a[4]	que corresponde a la dirección FF11

Así, si se hace referencia, a los elementos del arreglo a través del nombre del arreglo como apuntador, se debe emplear el operador "*" ya que hace referencia al contenido de la dirección apuntada por la variable o expresión que le sigue. Por lo tanto si se desea asignarle el valor de 7 al segundo elemento del arreglo, la sentencia se escribe de la siguiente forma:

*(a + 1)= 7;

Es indispensable el uso de paréntesis ya que el operador "*" tiene mayor precedencia que la suma, y ésta debe hacerse antes que la referencia al contenido de la localidad.

A pesar de que el nombre de un arreglo funciona como un apuntador, éste no puede ser modificado, debido a que su valor se mantiene constante durante la ejecución del programa, es decir, siempre apuntará al primer elemento del arreglo. Por esta característica se dice que el nombre de un arreglo es un *apuntador constante*.

EJEMPLO 1.5 PROGRAMA QUE MUESTRA EL USO DEL NOMBRE DE UN AREGLO COMO APUNTADOR CONSTANTE.

-Explicación de variables

v: Arreglo de 5 elementos de tipo *float*.
x: Variable de tipo *float*.
i: Variable de tipo *int*. Se utiliza como variable de control de ciclos.

-Código

```
#include <stdio.h>
main()
{
   float v[5];
   float x = 100.5;
   int i;

   for(i=0;i<5;i++){
     *(v+i)=i*5.0;
     printf("valor de v[%d]: %f \n", i,*(v+i));
   }
   v=&x; /*error: intento de modificar un apuntador
         constante*/
}
```

-Análisis del código

En el ciclo

```
   for(i=0;i<5;i++){
     *(v+i)=i*5.0;
     printf("valor de v[%d]: %f \n", i, v[i]);
   }
```

se asignan valores a los elementos del arreglo utilizando su nombre como apuntador y se van imprimiendo estos valores utilizando subíndices.

En la sentencia

```
v=&x; /*error: intento de modificar un apuntador constante*/
```

como el nombre de un arreglo es un apuntador constante, no un apuntador variable, no se puede cambiar el valor de un nombre de arreglo, así como no se pueden cambiar constantes.

Esto explica por qué no se pueden asignar valores nuevos a un arreglo durante una ejecución de un programa. Sólo se pueden cambiar apuntadores para hacerlos apuntar a valores diferentes en memoria. Por ejemplo, el siguiente fragmento de código:

```
int a[10], x;
int *p;

p=&a[0]; /* asignación de la dirección de a[0] */
x=*p;    /* asignación del contenido de la localidad a la que
                  apunta p a x */
*(p+1)=100; /* asignación de 100 al segundo elemento de a*/
```

Mediante este ejemplo se puede observar que **a[i]** es lo mismo que **p + i**, pero se debe tener cuidado, ya que en C no hay una revisión de los límites del arreglo, por lo que se podría fácilmente ir más allá del arreglo en memoria y sobreescribir otros datos. Sin embargo, el lenguaje C puede manejar perfectamente bien la combinación de arreglos y apuntadores. Por ejemplo:

Si generalmente se escribe:	Se puede cambiar por:
p = &a[0];	**p = a;**
a[i],	***(a + i)**
a+i	**&a[i]**

El direccionamiento de apuntadores se puede expresar como:

a[i] que es equivalente a ***(a + i)**

Sin embargo aunque tienen una estrecha relación, los apuntadores y los arreglos se diferencian de la siguiente manera:

- Un apuntador es una variable. Se puede escribir **ap = a** y **ap++**.
- Un arreglo NO ES un apuntador variable. Escribir **a = ap** y **a++** no está permitido.

EJEMPLO 1.6 LEER DESDE EL TECLADO UN ARREGLO BIDIMENSIONAL DE 3X3 ELEMENTOS ENTEROS. UTILIZANDO 2 APUNTADORES, DETERMINAR EL ELEMENTO DE MAYOR VALOR.

-Explicación de variables

A: Arreglo bidimensional de tipo entero

i: Variable de tipo entero. Se utiliza como control de ciclos y como índice del arreglo.
j: Variable de tipo entero. Se utiliza como control de ciclos y como índice del arreglo.
max: Apuntador a entero. Almacena la dirección de la localidad del elemento de mayor valor.
p: Apuntador a entero. Se utiliza para apuntar a todos los elementos del arreglo.

-Código

```
#include<stdio.h>

void main(){
   int A[3][3],i,j;
   int *max, *p;
   /*Ciclo de lectura*/
   for(i=0;i<3;i++){
      for(j=0;j<3;j++){
         printf("Dame el valor de A[%d][%d]: ",i,j);
         scanf("%d",&A[i][j]);
      }
   }
   printf("Matriz leída:\n");
   for(i=0;i<3;i++){
      for(j=0;j<3;j++){
         printf("%d\t",A[i][j]);
      }
      printf("\n");
   }
   max=&A[0][0]; /*Asignación de valor inicial*/
   p=max; /*Asignación de valor inicial*/
   /*Ciclo de búsqueda del elemento de mayor valor*/
   for(i=0;i<9;i++){
      /*Si el elemento al que está apuntando (p+i) es mayor
      que el elemento al que apunta max, se asigna la nueva
      dirección a max*/
      if(*(p+i)>*max){
         printf("%d > %d\n",*(p+i),*max);
         printf("Cambia max: %x",max);
         max = p+i;
         printf(" por: %x\n",max);
      }
      else{
         printf("%d < %d\n",*(p+i),*max);
      }
   }
   printf("El elemento más grande del arreglo es %d\n",*max);
```

```
}
```

-Análisis del código

En este programa se puede observar la forma en que trabajan los apuntadores. Aún cuando se tiene un arreglo bidimensional, si al apuntador se le van sumando posiciones, éste va haciendo referencia a los elementos por renglones, tal como se muestra en la figura 1.6.

Por tanto, la primera parte del código utiliza el arreglo y 2 índices para ubicarse dentro de éste, pero después de desplegar en pantalla el arreglo leído, se empieza la búsqueda del elemento mayor del arreglo utilizando apuntadores.

La forma en que se busca el elemento es aumentando la posición en uno al apuntador. Visto como una posición del arreglo se tiene que:

*(p+0)	apunta a:	A[0][0]
*(p+1)	apunta a:	A[0][1]
*(p+2)	apunta a:	A[0][2]
*(p+3)	apunta a:	A[1][0]
*(p+4)	apunta a:	A[1][1]
*(p+5)	apunta a:	A[1][2]
*(p+6)	apunta a:	A[2][0]
*(p+7)	apunta a:	A[2][1]
*(p+8)	apunta a:	A[2][2]

Existe una relación entre los subíndices del elemento del arreglo y la posición que se suma al apuntador. El esquema de la figura 1.11, indica esta relación para un arreglo bidimensional de 3x5.

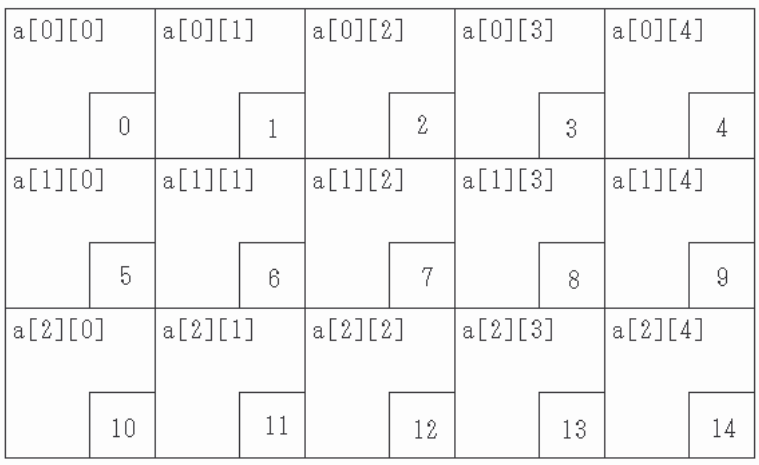


FIGURA 1.11 Arreglo bidimensional de 3x5 indicando la posición del elemento.

La ecuación general para determinar la posición a partir de los subíndices es:

$$pos_{i,j} = ((J+1)*i)+j$$

Donde:

i: es el índice que indica el renglón
j: es el índice que indica la columna
J: es el número de columnas del arreglo menos 1, o sea el valor máximo de j.

Ahora, aplicando la ecuación para determinar algunas posiciones de elementos del arreglo de la figura 1.11:

$$pos_{0,2} = ((4+1)*0)+2 = (5*0)+2 = 2$$

$$pos_{1,3} = ((4+1)*1)+3 = (5*1)+3 = 5+3 = 8$$

$$pos_{2,1} = ((4+1)*2)+1 = (5*2)+1 = 10+1 = 11$$

1.1.3 Aritmética de direcciones

Como se describió en el apartado anterior, cuando se hace uso de apuntadores, éstos se comportan como variables, debido a que pueden modificar su valor durante la ejecución de un programa.

De esta forma, cuando se utiliza un apuntador a un arreglo, al sumar 1, 2, 3, etc., el valor del apuntador cambia a la dirección de la localidad del siguiente elemento, que dependiendo del tipo de dato es el número de localidades que se suman a la dirección inicial.

Por ejemplo si se tuviera un apuntador *p*, a un arreglo de reales *R*, con la dirección inicial AB10, y suponiendo que un número real ocupa 4 localidades de memoria, cuando se suma uno al apuntador, es decir *p+1*, lo que se hace es sumar AB10 + 4, es decir que el resultado es AB14, y no AB11. Y si se sumara *p+3*, lo que se suma es AB10 + C, es decir que apunta a la localidad AB1C. Esto se puede observar en la gráfica de la figura 1.12.

Además del operador suma (+) en los apuntadores, también existe el operador resta (-), el cual realiza decrementos en direcciones para acceder a un dato que se encuentre en localidades más bajas de memoria.

Otros operadores para manejo de apuntadores son ++ y -- , donde el primero incrementa el valor del apuntador a la siguiente localidad de memoria de acuerdo al tipo dato; mientras el operador -- hace el decremento.

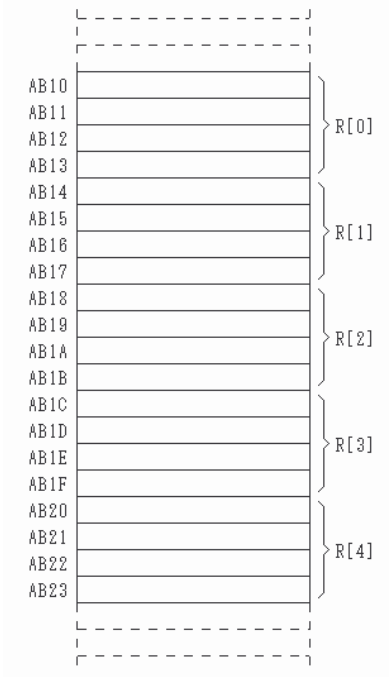


FIGURA 1.12 Mapa de memoria de un arreglo de tipo entero.

Asignación dinámica de memoria para almacenar datos

Hay ocasiones en que un programa requiere utilizar un segmento de memoria para almacenar datos, pero no se sabe cuántos datos serán hasta que se ejecute el programa. El utilizar arreglos declarados con una capacidad máxima, aunque después no se utilice en su totalidad, propicia un uso ineficiente de memoria. Lo ideal es siempre definir un arreglo con el número de elementos que realmente contendrá.

Una solución para resolver este mal uso de memoria es emplear el concepto de *asignación dinámica de memoria*; esto es, definir un espacio de memoria al momento de ejecución del programa con la cantidad de localidades que realmente se necesitarán. En el lenguaje C existe una función que realiza esta asignación dinámica de memoria: la función *malloc()*, incluida en la biblioteca estándar *stdlib.h.*

Si se requiriera un arreglo, para almacenar, por ejemplo, las coordenadas de varios puntos, pero todavía no se sabe con exactitud cuántos puntos son, porque los define el usuario al momento de ejecutar el programa, la función *malloc()*, busca un espacio libre, en el cual se puedan almacenar justamente la cantidad de datos a utilizar.

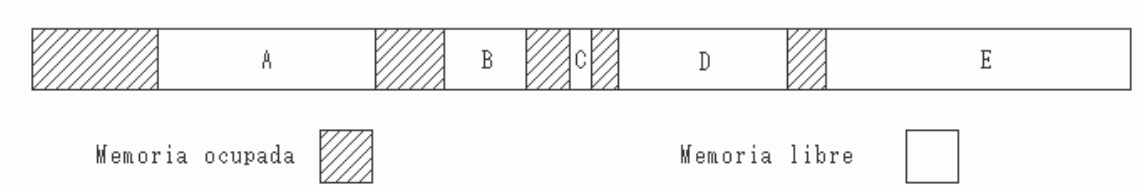


FIGURA 1.13 Diagrama de una memoria de dato.

La figura 1.13 muestra cómo está distribuida una memoria de dato en un momento determinado. Por ejemplo, si se requiriera un espacio para almacenar un conjunto de 12 datos, lo que hace el sistema operativo es buscar, dentro de los espacios libres de la memoria, el mejor lugar para asignar. En este caso, si el espacio que se debe asignar para que se almacenen los 12 datos fuera aproximadamente del tamaño de D, primeramente el sistema busca en memoria los segmentos libres, luego, de entre ellos, aquél que tenga el tamaño suficiente para almacenarlos. Observando el diagrama, el segmento puede ser el A, el D o el E; de acuerdo a la técnica de asignación de memoria que emplee el sistema operativo, seleccionará el segmento y si éste es mayor al que se requiere, deja libre el sobrante.

Función *malloc( )*

Con el uso de la función *malloc( )*, se pueden obtener bloques de memoria dinámicamente.

La función *malloc( )* devuelve la dirección inicial de un espacio de memoria, del tamaño especificado en el argumento de la función, de la siguiente manera:

```
p = (tipo_dato *) malloc (tamaño_requerido);
```

En la asignación anterior, el apuntador `p` contendrá la dirección de inicio del segmento asignado. El cambio de tipo `(tipo_dato *)`, se utiliza debido a que la función está definida como `void`, por lo que hay que hacer una conversión de formato, o sea un *cast*. Es importante notar que `tipo_dato` debe ser igual que el tipo de dato que apunta `p`.

Por último, `tamaño_requerido` debe especificar el tamaño exacto en *bytes*. Para este cálculo, es muy conveniente utilizar la función *sizeof ()*, que devuelve el tamaño en *bytes* de un tipo de dato. Por ejemplo, si se desea asignar un espacio en memoria que almacene los datos de un arreglo de 5 elementos de tipo *float*, se deben escribir las siguientes instrucciones:

```
float *fp;

fp = (float *) malloc( 5 * sizeof(float));
```

Con esto se asigna un espacio en memoria suficiente para almacenar 5 elementos de tipo *float*.

Función *calloc( )*

Otra variación de la función *malloc( )* es la *calloc( )*, que en esencia realiza la misma función: obtiene un espacio en memoria. Esta función recibe 2 parámetros: el número de elementos que se van a almacenar y el tamaño de dichos elementos. Se emplea de la siguiente forma:

```
p = (tipo_dato *) calloc(número_objetos,tamaño_objetos);
```

Y tomando el mismo ejemplo que en la función malloc(), se tiene:

```
float *fp;

fp = (float *) calloc( 5 , sizeof(float));
```

Función *free( )*

Esta función es igual de importante que las anteriores, porque con esto se libera el espacio asignado en memoria, y de esta forma, si se está trabajando en un programa que sólo requiere asignar segmentos de memoria para almacenar resultados parciales, se pueden ir asignando segmentos y luego irlos liberando para evitar el problema de agotar la memoria. Esta función sólo recibe como parámetro el apuntador en el cual se almacenó la dirección inicial del segmento, de la siguiente forma:

```
free(fp);
```

EJEMPLO 1.7 UTILIZANDO LA FUNCION *malloc()* Y/O *calloc()*, ASIGNAR 2 ESPACIOS EN MEMORIA PARA DOS ARREGLOS, UNO DE TIPO ENTERO Y OTRO DE TIPO REAL, Y DESPLEGAR EN PANTALLA LAS DIRECCIONES INICIALES, ASÍ COMO USANDO LA ARITMÉTICA DE APUNTADORES, LAS DEMÁS DIRECCIONES.

-Explicación de variables

ip: Apuntador a entero. Almacena la dirección devuelta por la función *malloc()*.
fp: Apuntador a real. Almacena la dirección devuelta por la función *calloc()*.
ielementos: Variable entera que almacena el número de elementos enteros que defina el usuario.
felementos: Variable entera que almacena el número de elementos reales que defina el usuario.
i: Variable de tipo entero. Se utiliza como control de ciclos.

-Código

```
#include<stdio.h>
#include<stdlib.h>

void main(){
   int *ip;
   float *fp;
   int ielementos=0,felementos=0,i;

   printf("\n\nTamaño en bytes de un int: %d\n",sizeof(int));
   printf("Tamaño en bytes de unfloat: %d\n",sizeof(float));

   while(ielementos<=0){
      printf("¿Cuántos elementos enteros desea almacenar?");
      scanf("%d",&ielementos);
   }
   while(felementos<=0){
      printf("¿Cuántos elementos reales desea almacenar?");
      scanf("%d",&felementos);
   }

   ip=(int *)malloc(ielementos*sizeof(int));
   fp=(float *)calloc(felementos,sizeof(float));

   for(i=0;i<ielementos;i++){
      printf("Dirección de ip + %d: %X\n",i,ip+i);
   }
   for(i=0;i<felementos;i++){
      printf("Dirección de fp + %d: %X\n",i,fp+i);
   }
   free(ip);
   free(fp);
}
```

-Análisis del código

Inicialmente se definen los apuntadores que nos ayudarán a ubicar el espacio asignado en memoria, para los arreglos.

```
int *ip;
float *fp;
```

Luego, utilizando dos variables auxiliares, se pregunta al usuario el número de elementos de cada arreglo, para después obtener el espacio en memoria adecuado; en los apuntadores correspondientes se reciben las direcciones iniciales de cada espacio de memoria.

```
ip=(int *)malloc(ielementos*sizeof(int));
fp=(float *)calloc(felementos,sizeof(float));
```

Para verificar la correcta asignación de memoria para cada arreglo, se despliega en pantalla la dirección (utilizando el formato %X) donde se almacenará cada uno de los elementos.

```
for(i=0;i<ielementos;i++){
   printf("Dirección de ip + %d: %X\n",i,ip+i);
}
for(i=0;i<felementos;i++){
   printf("Dirección de fp + %d: %X\n",i,fp+i);
}
```

Por último, se utiliza la función *free()* para liberar la memoria.

```
free(ip);
free(fp);
```

1.1.4 Arreglos de apuntadores

Cuando se requiere utilizar un cierto número de apuntadores, de manera simultánea y que cada uno apunte a un dato diferente, pero del mismo tipo, se hace uso de los arreglos de apuntadores. Así, un arreglo de apuntadores es un arreglo tal que contiene como elementos a apuntadores, cada uno de ellos apuntando a un dato específico del mismo tipo.

Por ejemplo:

```
int *ap[10];
```

reserva un arreglo de 10 apuntadores a enteros, es un arreglo unidimensional.

A los elementos de `ap` sólo se les pueden asignar direcciones, de la misma manera que variables apuntador simples. Por ejemplo:

```
ap[5] = &edad; /* ap[5] apunta a la dirección de edad */
ap[4] = NULL;  /* ap[4] no contiene dirección alguna*/
```

Otro caso de arreglo de apuntadores es con caracteres:

```
char *puntos[25];   /*arreglo de 25 apuntadores a carácter*/
```

Se pueden definir arreglos tan complejos como el siguiente:

```
int *(*ap)[10];
```

el cual define un apuntador a un arreglo de apuntadores, que éstos a su vez, apuntan a enteros. La disección de la sentencia se presenta a continuación:

`(*ap)`	es un apuntador, `ap` es un nombre de variable
`(*ap)[10];`	es un apuntador a un arreglo
`*(*ap)[10];`	es un apuntador a un arreglo de punteros
`int *(*ap)[10];`	es un apuntador a un arreglo de punteros a enteros

El siguiente código es una muestra del uso de diferentes tipos de arreglos y apuntadores a arreglos:

```
#include <stdio.h>

int main() {
    int num;         /* variable entera, usada para control
                        de ciclos */
    int arr[5];      /* arreglo de 5 enteros */
    int *arrap[5];   /* arreglo de 5 apuntadores a enteros */
    int (*aparr)[5]; /* apuntador a un arreglo de 5 enteros*/
    int *(*ap)[5];   /* apuntador a un arreglo de 5 punteros
                        a enteros*/

   /* ciclo que da valores iniciales a los arreglos arr y  arrap*/
    for (num=0;num<5;num++){
        arr[num]= num;
        arrap[num]=&arr[num];
        printf("\narr[%d] %d &arr[%d] X",num,arr[num],num,
                &arr[num]);
        printf(" arrap[%d] %X", num, arrap[num]);
    }
    /* aparr apunta al arreglo de enteros arr */
    aparr=&arr;
    /*se imprime el arreglo de enteros usando aparr*/
    for (num=0;num<5;num++){
        printf("\n %d ", (*aparr)[num]);
    }
    /* ap apunta al arreglo de apuntadores a enteros (arrap) */
    ap=&arrap;
    /* se imprime el arreglo de enteros usando ap*/
        for (num=0;num<5;num++){
        printf("\n %X %d ", (*ap)[num],*(*ap)[num]);
    }
    getchar();
}
```

Otro uso común de los arreglos de apuntadores, es el arreglo de apuntadores a cadenas. Un ejemplo clásico de la definición e inicialización de este tipo de arreglos se muestra a continuación:

```
char *dias_semana[7]={“lunes”, “martes”, “miercoles”,
                “jueves”, “viernes”, “sabado”, “domingo” };
```

EJEMPLO 1.8 UTILIZANDO UN ARREGLO DE APUNTADORES, ORDENAR UN ARREGLO DE 10 ELEMENTOS ENTEROS EN FORMA ASCENDENTE, POR EL MÉTODO DE LA BURBUJA.

-Explicación de variables

a: Arreglo de tipo entero de 10 elementos.
p: Arreglo de 10 apuntadores a entero.
i: Variable de tipo entero. Utilizada como control de ciclos y como índice del arreglo.
j: Variable de tipo entero. Utilizada como control de ciclos y como índice del arreglo de apuntadores.
temp: Apuntador a entero. Utilizada como variable temporal.
c: Variable de tipo carácter. Utilizada para el espacio entre ciclos del método.

-Código

```
#include<stdio.h>

void main(){
   int a[10],*p[10],i,j,*temp;
   char c;
   /*Lectura de los elementos enteros*/
   for(i=0;i<10;i++){
      p[i]=&a[i];
      printf("Dame el elemento %d: ",i);
      scanf("%d",p[i]);
   }

   /*Método de burbuja*/
   for(i=0;i<9;i++){
   /*En cada ciclo se reduce el límite debido a que el último
   elemento ya es el mayor del arreglo*/
      for(j=0;j<(9-i);j++){
        /*Comparación de elementos para determinar el mayor*/
         if(*p[j]>*p[j+1]){
            printf("%d > %d\n",*p[j],*p[j+1]);
            temp=p[j];
            p[j]=p[j+1];
            p[j+1]=temp;
```

```
            }
            else{
               printf("%d < %d\n",*p[j],*p[j+1]);
            }
         }
         /*Se imprime el arreglo en cada ciclo del método*/
         for(j=0;j<10;j++){
            printf("%d\n",*p[j]);
         }
         /*Función que simula el getch()*/
         c=getc(stdin);
      }
/*Impresión tanto del arreglo original ubicado en a[],como el
ordenado ubicado en el arreglo de apuntadores *p[10] */
      printf("\tarr\t*p\n\n");
      for(i=0;i<10;i++){
         printf("%d:\t%d\t%d\n",i,a[i],*p[i]);
      }

}
```

1.2 Estructuras

En la sección anterior se revisó una estructura de almacenamiento de datos homogéneos denominada *arreglo*. Sin embargo no siempre se requiere emplear un conjunto de datos del mismo tipo, por lo que en lenguaje C, se tiene una estructura que permite manejar un conjunto de datos heterogéneos. Dicha estructura de almacenamiento se llama *estructura* o *struct*. Una estructura trabaja de la misma forma que un arreglo; visto en un mapa de memoria, los elementos de una estructura se almacenan secuencialmente en memoria.

El formato general para declarar una estructura en C, es:

struct{
  tipo_dato1 nom_var11,nom_var12,…
  tipo_dato2 nom_var21,nom_var22,…
  …
  tipo_daton nom_varn1, nom_varn2,…
}nom_estructura;

Por ejemplo, si se requiere almacenar los datos de un empleado, como lo son: *nombre*, *clave del empleado* y *sueldo*; se observa que son datos de diferente tipo ya que *nombre* es una cadena, *clave del empleado* es un entero y *sueldo* es un número real. Por tanto la forma

de almacenar estos datos es a través de una estructura que podría llamarse ***empleado***. La declaración de esta estructura se muestra a continuación:

```
struct{
  char nombre[50];
  int clave;
  float sueldo;
}empleado;
```

En la figura 1.14, se presenta un mapa de memoria para esta estructura donde se supone que el tamaño de un *int* es de 2 *bytes*, el de un *float* es de 4 *bytes*, y el de un *char* es de 1 *byte*. Se puede observar que estos elementos, están organizados secuencialmente en memoria.

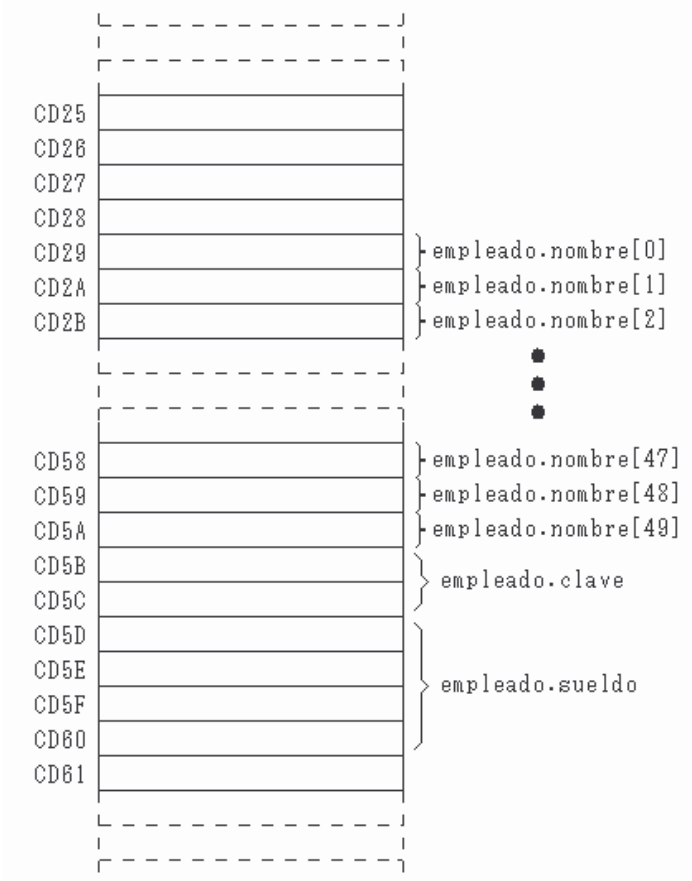


FIGURA 1.14 Mapa de memoria para una estructura *struct*, con tres elementos: un arreglo de tipo *char*, un int *clave* y un float *sueldo*.

Para hacer referencia a cada elemento de la estructura, basta con utilizar el nombre de la estructura, seguida del operador punto, y el nombre de la variable, por ejemplo:

```
empleado.clave=58736;
```

Con la cual, se almacena el valor de 58736 en la variable *clave*, definida en la estructura empleado.

EJEMPLO1.9 CALCULAR EL MÓDULO DE UN NÚMERO COMPLEJO, PIDIENDO COMO DATOS LA PARTE REAL Y LA PARTE IMAGINARIA. UTILIZAR LA BIBILIOTECA *math.h*, QUE CONTIENE LA FUNCION *sqrt( )*.

-Explicación de variables

complejo: estructura con 3 componentes
preal: variable de tipo real. Almacena la parte real del número complejo.
pimaginaria: variable de tipo real. Almacena la parte imaginaria del número complejo.
modulo: variable de tipo real. Almacena la magnitud del módulo del número complejo.

-Código

```
#include <stdio.h>
#include <math.h>
int main(){
   /*Definición de la estructura complejo*/
   struct {
      float preal;
      float pimaginaria;
      float modulo;
   }complejo;

   printf("\nParte real: ");
   /*Lectura de parte real*/
   scanf("%f",&complejo.preal);
   printf("\nParte imaginaria: ");
   /*Lectura de parte imaginaria*/
   scanf("%f",&complejo.pimaginaria);

   /*Obtención del módulo*/
   complejo.modulo = sqrt(complejo.preal*complejo.preal+
              complejo.pimaginaria*complejo.pimaginaria);
   /*Despliegue del resultado*/
   printf("Modulo %f ", complejo.modulo);
}
```

EJEMPLO1.10 UTILIZANDO ESTRUCTURAS, CALCULAR LA CANTIDAD A PAGAR DE UN PRODUCTO. LOS DATOS DE ENTRADA SON: EL NOMBRE DE LA MERCANCÍA, LA CANTIDAD DE PIEZAS A COMPRAR Y EL PRECIO UNITARIO.

-Explicación de variables

negocio: Estructura con 3 componentes

mercancia: Arreglo de tipo carácter. Almacena el nombre del producto a comprar.

cantidad: Variable de tipo entero. Almacena la cantidad de piezas a comprar.

precio_unitario: Variable de tipo real. Almacena el precio por unidad.

valor_total: Variable de tipo real. Almacena el valor total de la compra.

-Código

```
#include <stdio.h>
main()
{
     struct /*Definición de la estructura*/
     {    char mercancia[20];
          int cantidad;
          float precio_unitario;
     }negocio;
     float valor_total;
     printf("\n\t\tNEGOCIO");
     printf("\n Solicitud de pedido\n");
     /*Nombre de mercancia*/
     printf("\nNombre del mercancia: ");
     gets(negocio.mercancia);
     /*Número de piezas*/
     printf("\nCantidad: ");
     scanf("%d",&negocio.cantidad);
     /*precio unitario*/
     printf("\nPrecio de cada pieza:  ");
     scanf("%f", & negocio.precio_unitario);

     /*Calculo del total de la compra*/
     valor_total=negocio.cantidad*negocio.precio_unitario;
     /*imprime datos recibidos y el cálculo*/
     printf("\n\n");
     printf("\t\t NOTA DE COMPRA\n\n");
     printf("mercancia:      %s\n",negocio.mercancia);
     printf("Precio unitario:
     $%0.3f\n",negocio.precio_unitario);
     printf("Cantidad:       %d\n",negocio.cantidad);
     printf("Valor total:         $%0.3f\n",valor_total);
}
```

Una estructura también podría ser vista como un tipo de dato; esto puede ser útil si se requieren varias estructuras iguales, que manejen el mismo número y tipo de datos, pero cada una con valores propios. Por lo que cada estructura podrá ser manejada con un nombre de variable propio. Para generar una estructura como tipo de dato, la declaración debe tener la siguiente forma:

struct nom_tipo{
tipo_dato1 nom_var11,nom_var12,…
tipo_dato2 nom_var21,nom_var22,…
…
tipo_daton nom_varn1, nom_varn2,…
};

Con la definición de una estructura como tipo, ahora se pueden declarar variables de dicho tipo anteponiéndoles la palabra *struct* y el nombre de la estructura. Por ejemplo, utilizando la estructura de empleado:

```
struct datos{
   char nombre[50];
   int clave;
   float sueldo;
};
struct datos empleado, jefe, secretaria;
```

se declaran tres variables de tipo `struct datos`: `empleado`, `jefe` y `secretaria`; donde cada variable es una estructura con las características de la estructura definida como ***`datos`***.

Arreglos de Estructuras y Estructuras compuestas por Estructuras

Un arreglo puede contener como elementos a estructuras del mismo tipo. Con el formato visto anteriormente de definición de un tipo de estructura, la declaración de un arreglo de estructuras, tiene la siguiente forma:

struct nom_estructura {
tipo_dato1 nom_var11,nom_var12,…
tipo_dato2 nom_var21,nom_var22,…
…
tipo_daton nom_varn1, nom_varn2,…
} ;
struct nom_ estructura nom_arreglo[tamaño];

Por ejemplo, si se necesitaran los datos de 6 empleados, se debe generar un arreglo a partir de la definición vista previamente, quedando de la siguiente forma:

```
struct datos {
  char nombre[50];
  int clave;
  float sueldo;
};
struct datos empleado[6];
```

Debido a que generando estructuras, se generan nuevos tipos de datos, también se pueden anidar estructuras, como se observa en el siguiente ejemplo.

EJEMPLO 1.11 UTILICE 2 ESTRUCTURAS PARA DEFINIR UN PUNTO EN UN ESPACIO DE TRES DIMENSIONES, ES DECIR, CON COORDENADAS *X*, *Y* Y *Z*.

-Explicación de variables

punto2D: estructura con 2 componentes
        x: variable de tipo real
        y: variable de tipo real
punto3D: estructura con 2 componentes
        una estructura punto2D
        z: variable de tipo real

-Código

```
#include<stdio.h>

/*Definición de la estructura de un punto en 2D*/
struct punto2D{
   float x;
   float y;
};

/*Definición de un punto en 3D, utilizando la estructura de
2D, como otra componente*/
struct punto3D{
   struct punto2D xy;
   float z;
};

void main(){
/*Declaración de una estructura en 3D*/
   struct punto3D xyz;
```

```
   /*Asignación de valores iniciales*/
   xyz.xy.x=2;
   xyz.xy.y=4;
   xyz.z=6;

   printf("De nuestro punto en 3D\n");
   printf("El valor de x es %.2f\n",xyz.xy.x);
   printf("El valor de y es %.2f\n",xyz.xy.y);
   printf("El valor de z es %.2f\n",xyz.z);
}
```

-Análisis del código

Inicialmente se definen las estructuras que se utilizarán, en este caso, la estructura para un punto en 2D, y otra para un punto en 3D

```
struct punto2D{
   float x;
   float y;
};

struct punto3D{
   struct punto2D xy;
   float z;
};
```

Analizando el código de la estructura para un punto en 3D, en lugar de repetir las coordenadas `'x'` y `'y'`, se utiliza otra estructura, justamente la estructura de un punto en 2D, definida anteriormente.

Por esta razón, cuando se hace referencia a la coordenada `z`,' se utiliza el nombre de la estructura, `xyz`, seguida del operador `.`'y el nombre de la variable de la estructura.

```
xyz.z=6;
```

Pero cuando se quiere hacer referencia a las coordenadas `x`'y `y`,' primero se hace referencia a la estructura `xyz`, seguida del operador `.`,' para luego hacer referencia a la estructura `xy`, y después de otro operador `.`,' poder utilizar las coordenadas `x`'y `y`.'

```
xyz.xy.x=2;
xyz.xy.y=4;
```

1.3 Archivos y bancos de datos

Un archivo es un conjunto de datos que se almacenan en memoria secundaria, dichos datos cuentan con un nombre. A diferencia de la memoria principal, la memoria secundaria es toda aquella que no es usada directamente por la CPU, son todos los dispositivos tales como CD's, discos de 3 ½ , etc.

Ahora, la interrogante es ¿cómo se puede acceder a esa información desde un lenguaje de programación? La respuesta es simple, así como se observó en el tema de arreglos, los datos almacenados dentro de un archivo, se pueden visualizar como si estuvieran puestos secuencialmente dentro de otra memoria, pero para tener acceso tanto para lectura como para escritura, se debe utilizar un apuntador que nos permita obtener esos datos.

El tipo de apuntador que se necesita en el acceso a los datos de un archivo, en esencia es igual al visto anteriormente, pero se declara de distinta forma, esto debido a que no son localidades de memoria principal, sino de memoria secundaria, por lo que el tipo de apuntador que se tiene que declarar es de tipo FILE, y su formato es:

*FILE *apuntador_archivo;*

La definición de esta estructura depende del compilador, pero en general mantienen un campo con la posición actual de lectura/escritura, un buffer para mejorar las prestaciones de acceso al archivo y algunos campos para uso interno.

Una vez definido dicho apuntador, hay que "abrir" el archivo para acceder a los datos que contenga o que se grabarán en él; esto se hace con la función *fopen()*, la cual recibe como parámetros el nombre del archivo y el modo en que va a ser empleado.

Función *fopen*( )

La cabecera de esta función es la siguiente:

*FILE *fopen(char *nombre, char *modo);*

Teniendo definido el apuntador al archivo, la primera operación a realizar es la de "abrir" el archivo, esto es para poder manipularlo o utilizarlo; esta acción se realiza mediante la función *fopen()*, la cual recibe como parámetros el nombre del archivo y el modo en que va a ser empleado. El nombre del archivo es una cadena que contiene un nombre de archivo válido, y dependiendo del sistema operativo donde se esté trabajando, el nombre puede incluir el camino completo. El *modo* especifica el tipo de acceso al archivo, que puede ser para lectura, para escritura o para agregar datos a éste si ya existe; también se indica si los datos están en binario o en texto. La especificación del modo, se da a través de una secuencia de siglas como se indica en la siguiente tabla:

Modo	Acción
“r”	Abre un archivo para lectura. El archivo debe existir.
“w”	Crea un archivo para escritura; si existe, borra y escribe nueva información en el archivo.
“a”	Abre el archivo para escritura; se sitúa al final del mismo y añade; o si no existe lo crea.
“rb”	Abre un archivo en modo binario para lectura.
“wb”	Crea un archivo binario para escritura; si existe, borra y escribe nueva información en el archivo.
“ab”	Abre o crea un archivo binario y añade al final del mismo.
“r+”	Realiza lectura y escritura. El archivo debe existir
“w+”	Realiza lectura y escritura, se crea un archivo nuevo o se sobreescribe si ya existe.
“a+”	Añade, lee y escribe; el apuntador se sitúa al final del archivo. Si no existe, se crea.
“t”	tipo texto, si no se especifica "t" ni "b", se asume por defecto que es "t".
“b”	Indica tipo binario.

Función *fclose( )*

La cabecera de esta función es:

*int fclose(FILE *archivo);*

Es importante ‘cerrar’ los archivos cuando se han terminado de utilizar; esto con la finalidad de hacer un buen manejo de memoria. La función ***fclose( )*** realiza esta operación que incluye una serie de actividades: almacena los datos que aún están en el buffer de memoria, actualiza algunos datos de la cabecera del archivo que mantiene el sistema operativo y libera el archivo para que otros programas puedan abrirlo para su uso.

Un valor de retorno cero indica que el archivo ha sido correctamente cerrado, si ha ocurrido algún error, el valor de retorno es la constante EOF. El parámetro es un apuntador a la estructura FILE del archivo que se desea cerrar.

Función *fgetc()*

La cabecera de esta función es:

*int fgetc(FILE *archivo);*

La función *fgetc()*, obtiene un carácter del archivo al que se está apuntando, posteriormente actualiza el valor del apuntador para que apunte al siguiente carácter. Cuando el apuntador llega al final del archivo, éste apunta al carácter EOF (End Of File).

Función *fputc( )*

La cabecera de esta función es la siguiente:

*int fputc(int caracter, FILE *archivo);*

Esta función escribe un carácter a un archivo. El valor de retorno es el carácter escrito si la operación fue completada con éxito, en caso contrario será EOF. Los parámetros de entrada son el carácter a escribir, convertido a **int** y un apuntador a una estructura FILE del archivo en el que se hará la escritura.

Función *feof( )*

La cabecera de esta función es:

*int feof(FILE *archivo);*

Esta función sirve para comprobar si se ha alcanzado el final del archivo. Muy frecuentemente se debe trabajar con todos los valores almacenados en un archivo de forma secuencial, la forma que suelen tener los ciclos para leer todos los datos de un archivo es permanecer leyendo mientras no se detecte el fin del archivo. Esta función suele usarse como prueba para verificar si se ha alcanzado o no ese punto.

El valor de retorno es distinto de cero sólo si no se ha alcanzado el fin del archivo. El parámetro es un apuntador a la estructura FILE del archivo que queremos verificar.

Por ejemplo:

```
while(!feof(f)){
} /* Verifica que no haya llegado al final del archivo*/
```

Función *fgets( )*

Cabecera de esta función:

*char *fgets(char *cadena, int n, FILE *archivo);*

Esta función lee cadenas de caracteres. Leerá hasta n-1 caracteres o hasta que lea un retorno de línea. En este último caso, el carácter de retorno de línea también es leído. El parámetro *n* permite limitar la lectura para evitar desbordar el espacio disponible en la *cadena*.

El valor de retorno es un apuntador a la cadena leída si se leyó con éxito, y es NULL si se detecta el final del archivo o si hay un error. Los parámetros son: la cadena a leer, el número de caracteres máximo a leer y un apuntador a una estructura FILE del archivo del que se leerá.

Por ejemplo:

```
fgets(variable_donde_guardar,numero_MAX,
apuntor_a_archivo);
fgets(cadena,20,f);
```

Función *fputs( )*

Cabecera de la función:

*int fputs(const char *cadena, FILE *archivo);*

La función *fputs( )* escribe una cadena en un archivo. No añade el carácter de retorno de línea ni el carácter nulo final. El valor de retorno es un número no negativo o EOF en caso de error. Los parámetros de entrada son la cadena a escribir y un apuntador a la estructura FILE del archivo donde se realizará la escritura.

Función *fread( )*

Cabecera:

*size_t fread(void *apuntador, size_t tamaño, size_t nregistros, FILE *archivo);*

Esta función está pensada para trabajar con registros de longitud constante. Es capaz de leer desde un archivo uno o varios registros de la misma longitud y a partir de una dirección de memoria determinada. El usuario es responsable de asegurarse de que hay espacio suficiente para contener la información leída.

El valor de retorno es el *número de registros leídos*, no el número de bytes. Los parámetros son: un apuntador a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un apuntador a la estructura FILE del archivo del que se hará la lectura.

Función *fwrite*( )

Cabecera:

*size_t fwrite(void *apuntador, size_t tamaño, size_t nregistros, FILE *archivo);*

Esta función también está pensada para trabajar con registros de longitud constante y forma pareja con *fread( )*. Es capaz de escribir hacia un archivo uno o varios registros de la misma longitud almacenados a partir de una dirección de memoria determinada.

El valor de retorno es el *número de registros escritos*, no el número de bytes. Los parámetros son: un apuntador a la zona de memoria donde se almacenarán los datos leídos, el tamaño de cada registro, el número de registros a leer y un apuntador a la estructura FILE del archivo del que se hará la lectura.

Función *fprintf( )*

Cabecera:

*int fprintf(FILE *archivo, const char *formato, ...);*

La función *fprintf( )* funciona igual que *printf( )* en cuanto a parámetros, pero la salida se dirige a un archivo en lugar de a la pantalla.

Por ejemplo:

```
fprintf(f,"El número escogido es: %d",num);
```

Función *fscanf( )*

Cabecera:

*int fscanf(FILE *fichero, const char *formato, ...);*

La función *fscanf( )* funciona igual que *scanf( )* en cuanto a parámetros, pero la entrada se toma de un archivo en lugar del teclado.

Por ejemplo:

```
fscanf(f,"%d %d",&var1,&var2);
```

EJEMPLO 1.12 LEER UN ARCHIVO DE TEXTO SIMPLE, INDICANDO DENTRO DE LA EJECUCIÓN DEL PROGRAMA EL NOMBRE DEL ARCHIVO A LEER.

-Explicación de variables

f: Apuntador a FILE.
arch: Arreglo de tipo carácter. Almacena el nombre del archivo a ser leído.
c: Variable de tipo carácter. Utilizada para ir leyendo carácter por carácter el archivo.

-Código

```
#include<stdio.h>
```

```
void main(){
   FILE *f;      /*Declaración del apuntador a archivo*/
   char arch[25]=”archivo.txt”; /*Declaración del arreglo con
               el nombre del archivo */
   char c;      /*variable auxiliar*/

   f=fopen(arch,"r");/* Apertura del archivo para lectura*/
   c=fgetc(f); /*Obtención del primer carácter del archivo*/
  /*Mientras no encuentre final de archivo, hace la lectura*/
   while(c!=EOF){
      printf(“%c”,c);
      c=fgetc(f);
   }
   fclose(f); /* Se cierra el archivo*/
}
```

-Análisis del código

Con la declaración `FILE *f;` se define la variable `f` de tipo apuntador a archivo. Mediante este apuntador se manipulará al archivo que el programa leerá.

Luego se abre el archivo con `fopen(arch,"r")` asignando el valor que devuelve la función al apuntador `f`, o sea, la dirección de la primera localidad de memoria del archivo.

Con la función `fgetc(f)`, se obtiene el carácter que está apuntando `f;` al término de esta operación, `f` actualiza su valor para apuntar al siguiente carácter. Con la sentencia

```
c=fgetc(f);
```

la variable tipo carácter `c` recibe el carácter apuntado por `f`. Esta sentencia se encuentra en un ciclo `while` para leer carácter por carácter del archivo e irlo imprimiendo, hasta encontrar la constante simbólica `EOF` (End Of File).

Una vez que se ha terminado de utilizar el archivo se “cierra” con la sentencia

```
fclose(f);
```

EJEMPLO 1.13 ESCRITURA A UN ARCHIVO CON DATOS INTRODUCIDOS DESDE EL TECLADO. LA INDICACIÓN DE FIN DE INTRODUCCIÓN DE DATOS ES CON 2 SALTOS DE LÍNEA JUNTOS.

-Explicación de variables

f: Apuntador a FILE.
arch: Arreglo de tipo carácter. Almacena el nombre del archivo sobre el que se escribirá.
c: Variable de tipo carácter. Utilizada para ir leyendo carácter por carácter el archivo.

salir: Variable de tipo entero. Auxiliar en la terminación del programa.

-Código

```
#include<stdio.h>

void main(){
   FILE *f;
   char arch[25],c;
   int salir = 0;
   printf("Dame un nombre para el archivo:");
   gets(arch);
   f = fopen(arch,"w");
   c = getchar();
   while(salir<2){
      if(c=='\n'){
         salir++;
      }
      else{
         salir=0;
      }
      fputc(c,f);
      c=getchar();
   }
   fclose(f);
}
```

-Análisis del código

Al igual que en el programa anterior, se debe declarar un apuntador de tipo FILE.

```
FILE *f;
```

Se declara un arreglo de caracteres `char arch[25]` para almacenar el nombre del archivo y con él se abre en modo escritura

```
f = fopen(arch,"w");
```

Con la siguiente sentencia se obtiene el carácter tecleado y se asigna a la variable *c*

```
c = getchar();
```

El ciclo `while` se emplea para ir leyendo carácter por carácter desde el teclado e irlo almacenándolo en el archivo utilizando la sentencia:

```
fputc(c,f);
```

El ciclo while se detiene cuando el programa lea dos saltos de línea juntos.

Por último, se cierra el archivo cuando se termina de escribir datos, para asegurar el correcto almacenamiento de ellos en el archivo, con la sentencia

```
fclose(f);
```

Manejo de archivos de datos numéricos.

Es común, para ciertas disciplinas, almacenar datos numéricos en archivos. En lenguaje C, la manipulación de archivos que contengan datos numéricos no difiere mucho de uno de tipo texto.

Existen varias formas de almacenar números en un archivo. Por ejemplo, si se desea almacenar los elementos de un arreglo de 5 enteros en un archivo, el siguiente código podría ser parte de un programa que realizara esta operación.

```
FILE *apArch;
int arr[]={10,20,14,9,5};
/*Apertura del archivo para escritura */
apArch=fopen(“numeros.txt”,”w”);
for (int i=0;i<5;i++) {
     fprintf(apArch,“%d ”,arr[i]);
}
fclose(apArch);
```

La sentencia

```
fprintf(apArch,“%d ”,arr[i]);
```

escribe en el archivo apuntado por `apArch` el valor del *i*-ésimo elemento del arreglo `arr` y deja un espacio. Como esta sentencia está en un ciclo `for`, los cinco números escritos en el archivo estarán situados en un mismo renglón separados por un espacio.

Para leer 5 números enteros de un archivo que están separados por un espacio, y colocarlos en un arreglo, puede emplearse el siguiente código como parte de un programa:

```
FILE *apArch;
int arr[5];
/*Apertura del archivo para lectura */
apArch=fopen(“numeros.txt”,”r”);
for (int i=0;i<5;i++) {
     fscanf(apArch,“%d ”,&arr[i]);
}
fclose(apArch);
```

En caso de que los números a almacenar o a leer de un archivo sean reales o flotantes, sólo se cambia el formato por el adecuado (%f o %g).

En ocasiones los datos numéricos están almacenados en el archivo en modo texto, por lo que hay que hacer la conversión de cadena a número; para ello se sugiere revisar las siguientes funciones de C.

Función *atoi()*

Cabecera:

*int atoi(const char *s);*

Esta función convierte la cadena *s* a *int*. Se encuentra en la biblioteca <stdlib.h>

Función *atof()*

Cabecera:

*int atof(const char *s);*

Esta función convierte la cadena *s* a *double*. Se encuentra en la biblioteca <stdlib.h>

Función *atol()*

Cabecera:

*int atoi(const char *s);*

Esta función convierte la cadena *s* a *long*. Se encuentra en la biblioteca <stdlib.h>

Argumentos en la línea de comandos

Cuando se ejecuta un programa escrito en Lenguaje C, la primera función que se ejecuta es la función *main( )*; si no existe esta función el programa no se puede ejecutar. Esto permite que cuando se dé la orden de ejecución del programa, al mismo tiempo, se le pueda dar datos de entrada.

Debido a que *main( )* es una función, ésta puede utilizar dos argumentos, que por convención se llaman *argc* (por argument count) y *argv* (por argument vector). Si en la cabecera de la función *main( )* se incluyen estos argumentos, entonces se pueden enviar datos al momento de ordenar la ejecución del programa en la línea de comandos. La siguiente línea muestra el aspecto de la cabecera de la función *main( )* cuando se usan argumentos:

*main(int argc, char *argv[ ])*

Como se observa, el primer argumento, *argc*, es de tipo entero e indica el número de argumentos (cadenas) en la línea de comandos con los que se invocó el programa; el segundo, *argv*, es un arreglo de apuntadores a cadenas, que contiene los argumentos, uno por cadena.

Un ejemplo muy ilustrativo, es el programa *eco*, el cual despliega los argumentos que se le den en la línea de comandos, excepto el nombre del programa (*eco*). Esto es, si se escribe la siguiente línea de comandos

```
eco hola, mundo
```

donde `eco` es el nombre del programa ejecutable y las cadenas `hola,` y `mundo` son argumentos, entonces imprime

```
hola, mundo
```

El código del programa *eco.c* se presenta a continuación:

```
#include <stdio.h>

/* eco de los argumentos de la línea de comandos */
main(int argc, char*argv[ ]){
  int i; /*variable de control de ciclos*/

  /*ciclo de impresión de los argumentos*/
  for (i=1; i < argc; i++)
     printf("%s ", argv[i]);
  printf("\n");
  getchar();
}
```

-Ejemplo de ejecución

Si al momento de ejecutar el programa en la línea de comandos se escribe:

```
eco La vida es bella
```

el valor del primer argumento de la función *main( )*, *argc*, es 5 ya que son 5 cadenas las que se escribieron en la línea de comandos; el arreglo de apuntadores a cadenas, *argv[ ]*, tiene los siguientes valores:

```
argv[0] = "eco"
argv[1] = "La"
argv[2] = "vida"
argv[3] = "es"
argv[4] = "bella"
```

Es importante notar que `argv[0]` contiene el nombre del programa que se invocó; y propiamente los argumentos o datos de entrada están a partir de `argv[1]`. Es por esto que en el ciclo `for` del programa, el valor inicial del índice `i` del vector `argv[]` es 1, para que sólo escriba la frase sin incluir el nombre del programa. Por lo que salida es:

```
La vida es bella
```

Un uso común de argumentos en la función *main( )* es para introducir el nombre de un archivo que contiene datos que requiere el programa. Del ejemplo 1.12, si en lugar de solicitar el nombre del archivo se escribe al momento de ejecutar el programa, el código se transcribe de la siguiente forma:

```
#include<stdio.h>

int main(int argc, char *argv[]){
   FILE *f;
   char c;

   /* se comprueba si se escribió el nombre del archivo en la
      línea de comandos*/
   if (argc == 1){
       printf("No se escribió el nombre para el archivo");
       return(1);
   }
   else{
      f = fopen(argv[1],"r");
      c=fgetc(f); /*obtiene el primer carácter del archivo*/
      /*Mientras no halle fin de archivo, hace la lectura*/
      while(c!=EOF){
         printf(“%c”,c);
         c=fgetc(f);
      }
   }
   fclose(f);
}
```

Argumentos numéricos en *main()*

En aplicaciones numéricas, es necesario escribir argumentos numéricos en la línea de comandos. Sin embargo, la función *main( )* los va a recibir como cadenas; esto es porque el vector *argv[]* está definido como un arreglo de apuntadores a cadenas.

La solución a este problema es convertir la cadena a número ya sea entero o a real según sea el caso, utilizando las funciones propias descritas anteriormente.

EJEMPLO 1.14 PROGRAMA QUE SUMA UNA SERIE DE NÚMEROS REALES DADOS EN LA LÍNEA DE COMANDOS. ESTO ES, SI SE ESCRIBE EL COMANDO

```
suma  2.3  4.5  6.8
```

DONDE `suma` ES EL NOMBRE DEL PROGRAMA EJECUTABLE, IMPRIME

```
13.6
```

-Explicación de variables

i: Variable de control de ciclo e índice del arreglo de apuntadores a cadenas
suma: Variable que va obteniendo la suma
argc: Primer argumento de la función *main( )* que indica cuántos argumentos se leyeron de la línea de comandos
argv: Arreglo de apuntadores a cadenas, con los argumentos leídos

-Código

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
   int i;
   float suma=0;

   /* se comprueba si se escribieron números a sumar en
      la línea de comandos*/
   if (argc == 1){
       printf("No hay números que sumar");
       return(1);
   }
   else{
     for (int i=1;i<argc;i++) {
        suma=suma+atof(argv[i]);
     }
   }
   printf("El resultado de la suma es: %f", suma);
   getchar();
}
```

-Análisis del código

Se observa, primeramente, que la función `main()` cuenta con argumentos, donde en el primero (`argc`) indicará cuántas cadenas se leyeron en la línea de comandos, y en `argv[]`contendrá las cadenas leídas de la línea de comandos.

En la sentencia condicional `if (argc ==1)` se revisa si hay números a sumar; en caso de que sólo se haya escrito el nombre del programa , el valor de `argc` es 1, por lo que si no hay números a sumar, se termina el programa con la sentencia `return(1).` En caso contrario se efectúa la suma empleando el siguiente ciclo `for`:

```
for (int i=1;i<argc;i++) {
   suma=suma+atof(argv[i]);
}
```

en donde el índice del vector se inicia en 1 ya que `argv[0]` contiene el nombre del programa que se invocó, y a partir de `argv[1]` están las cadenas con los dígitos y el punto, en su caso, que forman los números.

Como el vector `argv[]` contienen cadenas, antes de efectuar la suma, es necesario convertir la cadena a una constante numéricas; por lo tanto se emplea la función `atof()` de la siguiente forma: `atof(argv[i]).`

Por último, se imprime la suma de los números escritos en la línea de comandos.

Ejercicios Propuestos

1) Elaborar un programa que realice la multiplicación de $\begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$

2) Elaborar un programa que multiplique $\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \begin{bmatrix} b_1 & b_2 \end{bmatrix}$

3) Elaborar un programa que multiplique $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$

4) Elaborar un programa que lea un matriz de m×n y que determine cuál es el valor máximo que se encuentra dentro de la matriz.

5) Elaborar un programa que realice la suma de dos matrices.

6) Elaborar un programa que obtenga la multiplicación de 2 matrices.

7) Elaborar un programa que obtenga la matriz inversa de una matriz cuadrada.

8) Elaborar un programa, donde usando un apuntador, se inviertan los elementos de una matriz, y los almacene en otra. Es decir: $\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \Rightarrow \begin{bmatrix} a_{3,3} & a_{3,2} & a_{3,1} \\ a_{2,3} & a_{2,2} & a_{2,1} \\ a_{1,3} & a_{1,2} & a_{1,1} \end{bmatrix}$

9) Elaborar un programa en el que utilizando un apuntador, se despliegue la matriz transpuesta de una matriz dada.

10) Utilizando el nombre del arreglo como apuntador, elaborar un programa que realice el producto cruz de 2 vectores con 3 componentes.

11) Utilizando la función *malloc()*, elaborar un programa que obtenga 2 polinomios de cualquier grado máximo, y obtenga la multiplicación.

12) Utilizando la definición de estructura de punto 3D, elaborar un programa que obtenga el módulo de un segmento de recta definido por dos puntos.

13) Elaborar un programa en el que lea desde un archivo una matriz, indicando al inicio del archivo la dimensión de la matriz, y utilice la función *malloc()* para asignar el espacio correspondiente en memoria. Al término de la lectura mandar desplegar en pantalla.

14) Elaborar un programa lea los coeficientes de 2 polinomios desde un archivo, realice la multiplicación e imprima el resultado al final del mismo archivo.

15) Elaborar un programa que lea desde un archivo los datos de varios empleados, compare sus sueldos y devuelva los nombres de aquellos que perciba un sueldo mayor a 10000.

16) Elaborar un programa que lea un archivo de texto simple, y lo transcriba en otro archivo, cambiando las mayúsculas por minúsculas y viceversa.

17) Elaborar un programa con argumentos en la función *main*, que realice la multiplicación de 2 números; los números se darán en la línea de comandos al momento de ejecutar el programa. Por ejemplo, si se ejecuta `mult 2 5`, el programa deberá devolver `10` como resultado.

18) Elaborar un programa con argumentos en la función *main*, que realice el promedio de varios números; los números se darán en la línea de comandos al momento de ejecutar el programa. Por ejemplo, si se desea el promedio de 6 números y se ejecuta lo siguiente: `promedio 10 9 10 8 8 9`, el programa deberá devolver `9` como resultado.

19) Elaborar un programa con argumentos en la función *main*, que realice operaciones consecutivas como si fuera una calculadora. Por ejemplo, si se ejecuta `calc 1 + 2 * 3 – 4 / 5`, el programa realiza las siguientes operaciones: 1 + 2 = 3, 3 * 3 = 9, 9 – 4 = 5, 5 / 5 = 1, por lo que el programa deberá devolver `1` como resultado.

CAPÍTULO 2

APROXIMACIÓN NUMÉRICA, ERRORES Y MÉTODOS NUMÉRICOS INICIALES

Introducción

El análisis numérico es la rama de la matemática que se encarga del desarrollo y estudio de procedimientos para la solución de problemas matemáticos de forma numérica. Esto es, cuando se realiza el análisis numérico de un problema siempre se obtiene una respuesta numérica, aun cuando no tenga solución "analítica". A los procedimientos sistemáticos o algoritmos que han sido desarrollados y probados para la resolución de este tipo de problemas, se les denominan métodos numéricos. Como en la mayoría de los métodos numéricos las operaciones que ocupan son las aritméticas: suma, resta, multiplicación y división, además de que en muchos de ellos se someten a un proceso repetitivo, resulta muy práctico el uso de la computadora.

2.1 Aproximación numérica y errores

Cuando se utilizan los métodos numéricos para la solución de un problema matemático, únicamente se aproxima al valor verdadero, esto es, porque el método al convertir el problema matemático en aritmético, se vuelve una solución inexacta.

Como se mencionó anteriormente, la forma en que funcionan los métodos, es por medio de algoritmos, los cuales llevan a la solución aproximada mediante un número de pasos finitos que se ejecutan de manera lógica.

Debido a la naturaleza de la solución, y que sólo es una aproximación a la solución verdadera, existen diversos tipos de errores que se llegan a cometer en todo el proceso de solución. Estos errores pueden ser de varios tipos, como:

- **Errores en los datos de entrada o inherentes:** Este tipo de error viene causado por los errores al realizar mediciones de magnitudes físicas, ya que se presenta la imposibilidad de realizar mediciones exactas.

- **Errores de redondeo o truncamiento:** En este caso, el error aparece al operar con representaciones numéricas finitas. Se puede solucionar utilizando más decimales, pero esto conlleva utilizar más memoria (recursos).

Exactitud y Precisión

La exactitud se refiere a qué tan cercano está el valor calculado o medido del valor verdadero. La precisión se refiere a qué tan cercano está un valor individual medido o calculado respecto a los otros.

La inexactitud se define como un alejamiento sistemático de la verdad.

Los métodos numéricos deben ser lo suficientemente exactos para que cumplan los requisitos de un problema particular de ingeniería.

Errores

Cuando se hacen cálculos para obtener la solución de un problema matemático, muchas veces se comenten errores debido a las diferentes limitaciones que se pueden presentar. Por ejemplo cuando se resuelve un problema en papel, los números que se van utilizando de una ecuación a otra, no son los mismos debido a un error en el redondeo o truncamiento de los decimales.

En general, la relación entre el número exacto y el obtenido por aproximación se define como *error* y está dado por:

$$\text{Error} = \text{Valor real} - \text{valor estimado}$$

Error por truncamiento

Este tipo de errores, se comenten generalmente cuando se tiene un número con una gran cantidad de decimales; por ejemplo, si en el transcurso de la solución de un problema se debe realizar la siguiente división $\frac{24}{7}$, el resultado es 3.428571… y suponiendo que se están usando 2 decimales en todos los pasos intermedios, el resultado del quebrado anterior se manejaría como 3.42, con lo cual ya se está cometiendo un error, y si este valor se transforma a una expresión en quebrado, se tiene $\frac{171}{50}$, con lo cual se puede observar a simple vista que los quebrados no son iguales y esto tiene un error por truncamiento.

Error por redondeo

Adicionalmente al error de truncamiento, hay ocasiones en las que para tener un valor más “preciso”, se redondea un decimal, sabiendo que si el siguiente número al decimal en el que se va a redondear es mayor o igual a 5, por ejemplo el valor de *"e"* se conoce como

2.7182818281828…, si este número se trunca a 8 cifras significativas, es decir 2.71828182, se obtiene un error de

E = 2.718281828... -2.71828182 = 0.000000008…
Sin embargo, si se considera que el número que sigue al corte es mayor que 5, y se realiza el redondeo a 2.71828183, el error será sólo de

E = 2.118281828... -2.11828183 = -0.000000002…

Tomando el ejemplo anterior de $\frac{24}{7}$, si se redondea a dos decimales, es decir 3.43, su expresión como quebrado queda como $\frac{343}{100}$, lo cual nos genera un error, al igual que en el caso anterior.

Propagación de errores

Mientras más cálculos se tengan que realizar para obtener un resultado, el error por redondeo se irá incrementando. Pero por otro lado, el error por truncamiento se puede minimizar al incluir más términos en la ecuación, disminuir el paso o proseguir la iteración (o sea mayor número de cálculos y seguramente mayor error de redondeo).

Cálculo del error

Cuando se hace una aproximación, se ha visto que existe un cierto error entre ambos valores, el real y el aproximado. Para conocer cuál es el error existente entre estos dos valores, existen dos tipos de cálculo de error: error absoluto y error relativo.

-Error absoluto

El error absoluto es la diferencia aritmética entre el valor real (V_R) y el valor aproximado (V_A). Es decir $\varepsilon = |V_R - V_A|$

Por ejemplo:

Si $V_R = \frac{24}{7}$, su aproximación por truncamiento a dos decimales es $\frac{171}{50}$, y su aproximación por redondeo es $\frac{343}{100}$; calcular los errores absolutos.

$$\varepsilon_T = |V_R - V_A| = \left|\frac{24}{7} - \frac{171}{50}\right| = \frac{3}{350} = 0.00857...$$

$$\varepsilon_R = |V_R - V_A| = \left|\frac{24}{7} - \frac{343}{100}\right| = \frac{1}{700} = 0.00142...$$

-Error relativo

El error relativo es el cociente entre el error absoluto y el valor real, es decir:

$$\eta = \frac{\varepsilon}{V_R} = \frac{|V_R - V_A|}{V_R}$$

Si se desea este error en forma porcentual, se calcula de la siguiente forma:

$$\eta = \frac{|V_R - V_A|}{V_R} \times 100 = \%Error$$

Por ejemplo, si $V_R = \frac{24}{7}$, su aproximación por truncamiento a dos decimales es $\frac{171}{50}$, y su aproximación por redondeo es $\frac{343}{100}$; calcular los errores absolutos de forma porcentual.

$$\eta_T = \frac{|V_R - V_A|}{V_R} \times 100 = \frac{\left|\frac{24}{7} - \frac{171}{50}\right|}{\frac{24}{7}} \times 100 = \frac{1}{4} = 0.25\%$$

$$\eta_R = \frac{|V_R - V_A|}{V_R} \times 100 = \frac{\left|\frac{24}{7} - \frac{343}{100}\right|}{\frac{24}{7}} \times 100 = \frac{1}{24} = 0.0417\%$$

2.2 Solución numérica de ecuaciones algebraicas y trascendentes

Cuando se habla de obtener raíces de una ecuación, lo primero que se pudiera pensar es en ecuaciones polinomiales, en donde a través de factorizar en binomios se puedan obtener las raíces solución de dicho polinomio. Pero también se puede dar el caso, en que se necesite la raíz o solución a una ecuación en la que estén involucrados términos que no se puedan despejar, como por ejemplo:

$$f(x) = 2x + e^x$$

Por más que se iguale la función a cero y se intente despejar la variable *x* para obtener su solución, no se podría; éste es el caso donde los métodos numéricos ayudan a encontrar dichos valores.

2.2.1 Método de bisección o búsqueda binaria

El método de bisección es un método numérico sencillo, muy útil para encontrar una solución en un intervalo en el que existe una raíz.

Suponiendo que hay una raíz de f(x) = 0 en un intervalo denotado por [a, b] o, de forma equivalente, por a ≤ x ≤ b; el método de bisección se basa en el hecho que dentro del intervalo dado existe una raíz, es decir que f(a) · f(b) < 0.

El primer paso de este método consiste en dividir el intervalo [a, b] en dos subintervalos: [a, c] y [c, b], donde:

$$c = \frac{a+b}{2}$$

Luego, se verifican los signos de f(a) · f(c) y f(c) · f(b), de esta forma se puede conocer en qué subintervalo se encuentra la raíz. Dicho subintervalo, pasa a ser el nuevo intervalo [a,b].

Por ejemplo, si en el subintervalo [c,b] se encontrara la raíz, entonces el valor de c, pasa a ser el nuevo valor de a, y se vuelve a repetir el procedimiento desde la división del intervalo en dos mitades.

Al repetir este procedimiento, el tamaño del intervalo se hará cada vez más pequeño. En cada iteración se supone el punto medio del intervalo como la aproximación más actualizada a la raíz.

La iteración se detiene cuando el valor de la $f(c)$ es cero, es decir, se ha encontrado la raíz, o bien cuando el tamaño del intervalo sea menor a una tolerancia dada.

EJEMPLO 2.1 RESOLVER LA SIGUIENTE ECUACIÓN UTILIZANDO EL MÉTODO DE BISECCIÓN. CONSIDERAR UN INTERVALO DE $[1,2]$ Y UNA TOLERANCIA DE 0.001

$$f(x) = x^4 + 5.8x^3 - 22.4x^2 - 31.2x + 57.6$$

Solución

Iteración 1

a = 1	f(a) = 10.8
c = 1.5	f(c) = -14.9625
b = 2	f(b) = -32

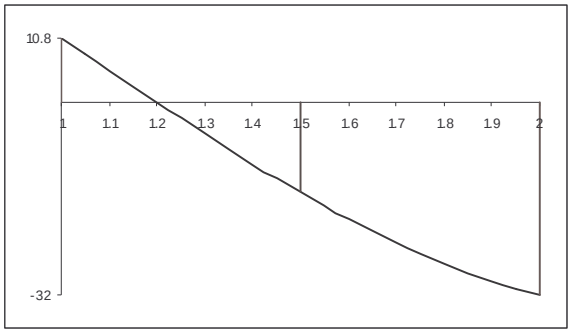


FIGURA 2.1 Gráfica de la iteración 1 del ejemplo 2.1.

El cambio de signo, es decir la raíz, se encuentra entre *a* y *c*, por lo que el nuevo intervalo es [1 , 1.5]. El tamaño del intervalo no es menor a 0.001, por lo que se continúa el método.

Iteración 2
a = 1 f(a) = 10.8
c = 1.25 f(c) = -2.63047
b = 1.5 f(b) = -14.9625

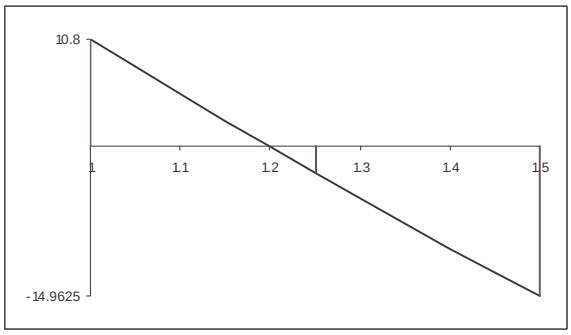


FIGURA 2.2 Gráfica de la iteración 2 del ejemplo 2.1.

El cambio de signo, es decir la raíz, se encuentra entre *a* y *c*, por lo que el nuevo intervalo es [1 , 1.25]. El tamaño del intervalo no es menor a 0.001, por lo que se continúa el método.

Iteración 3
a = 1 f(a) = 10.8
c = 1.125 f(c) = 4.01
b = 1.25 f(b) = -2.63047

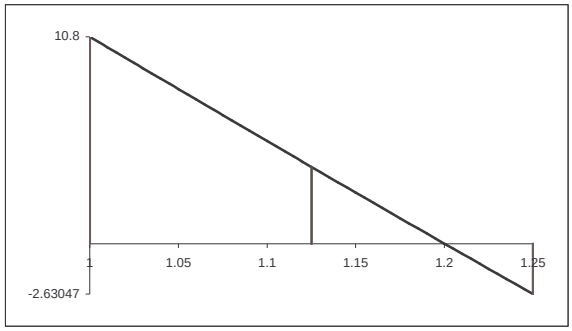


FIGURA 2.3 Gráfica de la iteración 3 del ejemplo 2.1.

Ahora, cambio de signo, se encuentra entre *b* y *c*, por lo que el nuevo intervalo es [1.125 , 1.25]. El tamaño del intervalo no es menor a 0.001, por lo que se continúa el método.

Iteración 4
a = 1.125 f(a) = 4.01
c = 1.1875 f(c) = 0.6635
b = 1.25 f(b) = -2.6305

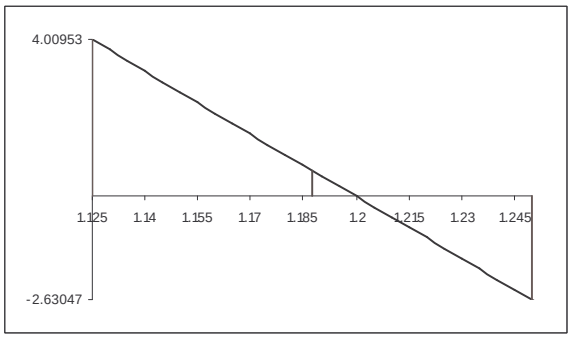


FIGURA 2.4 Gráfica de la iteración 4 del ejemplo 2.1.

En esta iteración, el cambio de signo, se encuentra entre *b* y *c*, por lo que el nuevo intervalo es [1.1875 , 1.25]. El tamaño del intervalo no es menor a 0.001, por lo que se continúa el método.

Iteración 5
a = 1.1875 f(a) = 0.6595
c = 1.21875 f(c) = -0.991
b = 1.25 f(b) = -2.6305

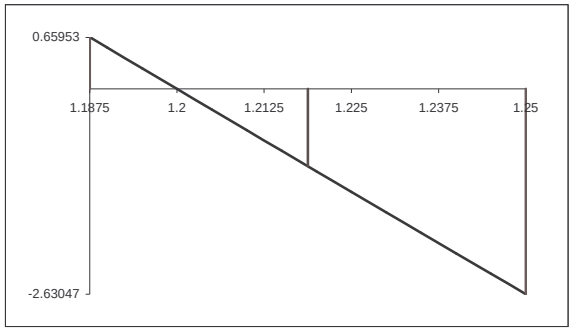


FIGURA 2.5 Gráfica de la iteración 5 del ejemplo 2.1.

El cambio de signo, se encuentra entre *a* y *c*, por lo que el nuevo intervalo es [1.1875 , 1.21875]. El tamaño del intervalo no es menor a 0.001, por lo que se continúa el método.

Después de 6 iteraciones más:

Iteración 11
a = 1.19922 f(a) = 0.0414
c = 1.1997 f(c) = 0.0155
b = 1.2002 f(b) = -0.01035

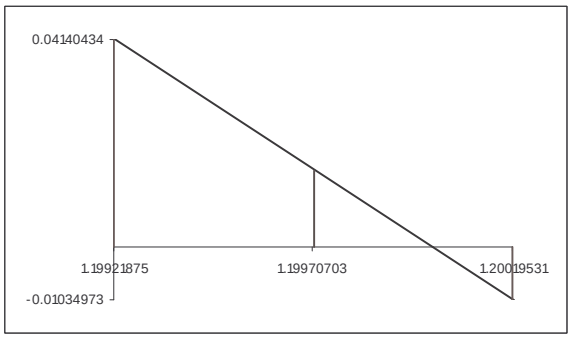


FIGURA 2.6 Gráfica de la iteración 11 del ejemplo 2.1.

El cambio de signo, se encuentra entre *a* y *c*, por lo que el nuevo intervalo es [1.1997 , 1.2002]. El tamaño del intervalo es menor a 0.001, por lo que se termina el método.

El valor exacto de la raíz es 1.2, y se puede observar que el valor de *c* en esta iteración ya es muy cercano.

EJEMPLO 2.2 RESOLVER LA SIGUIENTE ECUACIÓN UTILIZANDO EL MÉTODO DE BISECCIÓN. CONSIDERAR UN INTERVALO DE [-3,-2] Y UNA TOLERANCIA DE 0.002

$$f(x) = e^{-(x-1)} sen(x) + 1$$

NOTA: PARA EL ARGUMENTO DE LA FUNCIÓN SENOIDAL, CONVERTIR EL VALOR DE *X* A RADIANES. ES DECIR.

$$f(x) = e^{-(x-1)} sen\left(\frac{x\pi}{180}\right) + 1$$

Solución

Iteración 1		Iteración 2	
a = -3	f(a) =-1.8574	a = -2.5	f(a) = -0.4445
c = -2.5	f(c) = -0.4445	c = -2.25	f(c) = -0.0125
b = -2	f(b) = 0.2990	b = -2	f(b) = 0.2990
Iteración 3		Iteración 4	
a = -2.25	f(a) = -0.0125	a = -2.25	f(a) = -0.0125
c = -2.125	f(c) = 0.1561	c = -2.1875	f(c) = 0.0752
b = -2	f(b) = 0.2990	b = -2.125	f(b) = 0.1561
Iteración 5		Iteración 6	
a = -2.25	f(a) = -0.0125	a = -2.25	f(a) = -0.0125
c = -2.21875	f(c) = 0.0323	c = -2.2344	f(c) = 0.0101
b = -2.1875	f(b) = 0.0625	b = -2.21875	f(b) = 0.0323
Iteración 7		Iteración 8	
a = -2.25	f(a) = -0.0125	a = -2.2422	f(a) = -0.0012
c = -2.2422	f(c) = -0.0012	c = -2.2383	f(c) = 0.0045
b = -2.2344	f(b) = 0.0101	b = -2.2344	f(b) = 0.0101
Iteración 9		Iteración 10	
a = -2.2422	f(a) = -0.0012	a = -2.2422	f(a) = -0.0012
c = -2.2402	f(c) = 0.0017	c = -2.2412	f(c) = 0.0003
b = -2.2383	f(b) = 0.0045	b = -2.2402	f(b) = 0.0017

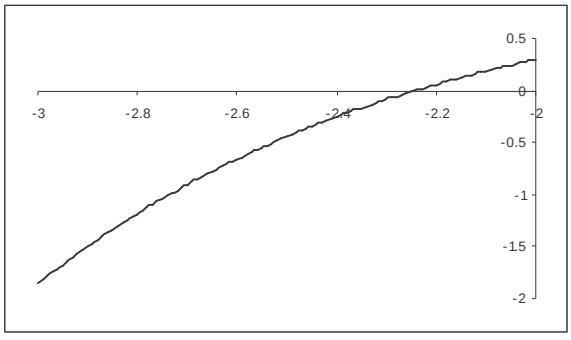


FIGURA 2.7 Gráfica de la función del ejemplo 2.2 en el intervalo [-3,-2].

Como se puede observar en la última iteración, el valor de *c*, ya es muy cercano a la raíz, ya que el valor de la función en ese punto es prácticamente cero.

EJEMPLO 2.3 ELABORAR UN PROGRAMA QUE RESUELA LA ECUACIÓN DEL EJEMPLO 2.1, UTILIZANDO EL MÉTODO DE BISECCIÓN.

-Algoritmo general del método de bisección

a = inicio_intervalo
b = fin_intervalo
REPETIR
    c = (a+b)/2
    SI f(c)*f(a)<0
        b = c
    CASO CONTRARIO
        a = c
HASTA |a – b|<tolerancia o f(c) = 0

-Explicación de variables

tolerancia: Variable de tipo real, utilizada para almacenar la tolerancia permitida para finalizar el método.
a: Variable de tipo real, utilizada para almacenar el valor inferior del intervalo.
b: Variable de tipo real, utilizada para almacenar el valor superior del intervalo.
c: Variable de tipo real, utilizada para almacenar el valor del punto medio del intervalo.
i: Variable de tipo entero, utilizada para contar el número de iteraciones realizadas.

-Código

```
#include<stdio.h>
#include<math.h>
float funcion(float x);

void main(){
   float tolerancia,a,b,c;
   int i=0;
   printf("Dame el valor de a: ");
   scanf("%g",&a);
   printf("Dame el valor de b: ");
   scanf("%g",&b);
   printf("Dame el valor de tolerancia: ");
   scanf("%g",&tolerancia);

   while(fabs(b-a)>tolerancia){
      c=(b+a)/2;
      if((funcion(a)*funcion(c))<0){
         b=c;
      }
      else{
         a=c;
```

```
      }
      i++;
   }
   printf("El valor mas cercano a la raiz es: %.2f en la it:
%d",c,i);
}

float funcion(float x){
   float res;
   res=pow(x,4)+(5.8*pow(x,3))-(22.4*pow(x,2))-(31.2*x)+57.6;
   return res;
}
```

-Análisis de código

El código inicia solicitando los valores del intervalo sobre el cuál se aplicará el método, así como el valor de la tolerancia permitida.

```
printf("Dame el valor de a: ");
scanf("%g",&a);
printf("Dame el valor de b: ");
scanf("%g",&b);
printf("Dame el valor de tolerancia: ");
scanf("%g",&tolerancia);
```

A continuación se ejecuta el algoritmo del método, donde se calcula el punto medio y se evalúa la condición de f(a)*f(c) < 0 para determinar en qué subintervalo se encuentra la solución de la ecuación; lo anterior se realiza iterativamente hasta que la distancia entre el valor superior y el inferior del intervalo sea menor a la tolerancia.

```
while(fabs(b-a)>tolerancia){
   c=(b+a)/2;
   if((funcion(a)*funcion(c))<0){
      b=c;
   }
   else{
      a=c;
   }
   i++;
}
```

Además del algoritmo de bisección, se implementa la función de la que se desea obtener su solución, por lo que en este programa se codifica la función del ejemplo 2.1:

```
float funcion(float x){
   float res;
   res=pow(x,4)+(5.8*pow(x,3))-(22.4*pow(x,2))-(31.2*x)+57.6;
   return res;
}
```

2.2.2 Método de Newton-Raphson

Otro método utilizado en la solución de ecuaciones, es el de Newton-Raphson. Este método es el más usado porque se realizan menos iteraciones, esto debido a que su funcionamiento se basa en rectas tangentes a la curva, y que intersecan el eje de las X para obtener el siguiente valor a utilizar. Es decir:

La forma en que trabaja este método es utilizando el concepto de derivada, que por definición es la pendiente de una recta tangente a la curva en un punto dado, este valor de pendiente se utiliza para encontrar un punto sobre el eje X, y que esté contenido en la recta con pendiente $f'(x)$, y que pasa por el punto $(x_n, f(x_n))$.

La fórmula que se utiliza para las iteraciones se obtiene a partir de dos ecuaciones de recta
- La ecuación de la recta tangente que pasa por el punto $(x_n, f(x_n))$: $f(x_n) = f'(x_n)x_n + C$
- Y otra ecuación que pasa por el punto $(x_{n+1}, 0)$, y tiene la misma pendiente y constante que la ecuación anterior: $0 = f'(x_n)x_{n+1} + C$

Se despejan las constantes de cada ecuación y se obtiene:
$C = f(x_n) - f'(x_n)x_n$
$C = -f'(x_n)x_{n+1}$

Igualando ambas ecuaciones:
$-f'(x_n)x_{n+1} = f(x_n) - f'(x_n)x_n$

Luego se despeja el valor de x_{n+1}:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

La ecuación resultante, es la formula de Newton-Raphson. Este método termina cuando el valor de $|x_{n+1} - x_n|$ es cero o menor a una tolerancia dada.

EJEMPLO 2.4 RESOLVER LA SIGUIENTE ECUACIÓN, UTILIZANDO EL MÉTODO DE NEWTON-RAPHSON. CONSIDERAR UN VALOR INICIAL DE 0.6 Y UNA TOLERANCIA DE 0.001.

$$f(x) = x^4 + 5.8x^3 - 22.4x^2 - 31.2x + 57.6$$

Solución

Se obtiene la primera derivada de la función:
$f'(x_n) = 4x^3 + 17.4x^2 - 44.8x - 31.2$

Iteración 1

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = 0.6 - \frac{32.1984}{-50.952} = 1.2319$$

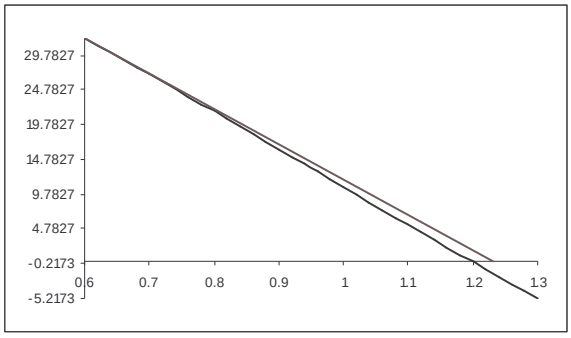


FIGURA 2.8 Gráfica de la iteración 1 del ejemplo 2.4.

Como no se cumple $|1.2319 - 0.6| < 0.001$, se hace una iteración más

Iteración 2

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = 1.2319 - \frac{-1.6847}{-52.5047} = 1.1998$$

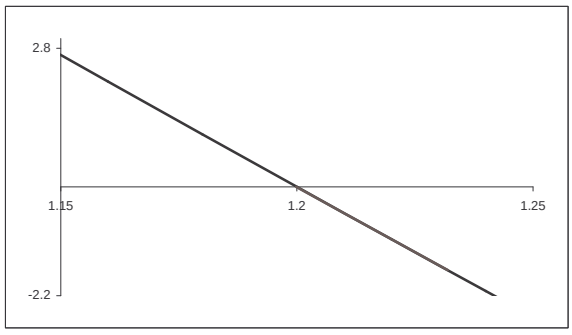


FIGURA 2.9 Gráfica de la iteración 2 del ejemplo 2.4.

Haciendo un acercamiento a alrededor de 1.2, es decir [1.199,1.201]

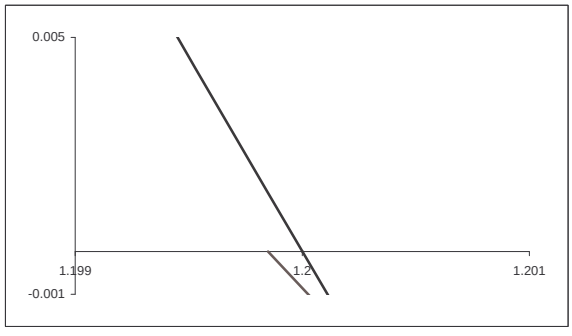


FIGURA 2.10 Gráfica de la iteración 2 del ejemplo 2.4, con un intervalo más pequeño.

Como no se cumple $|1.1998-1.2319|<0.001$, se hace una iteración más

Iteración 3

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}=1.1998-\frac{0.008}{-52.9942}=1.2$$

Este valor ya cumple con la condición de $|1.2-1.1998|<0.001$, por lo que el método se detiene.

EJEMPLO 2.5 RESOLVER LA SIGUIENTE ECUACIÓN, UTILIZANDO EL MÉTODO DE NEWTON-RAPHSON. CONSIDERAR UN VALOR INICIAL DE -2 Y UNA TOLERANCIA DE 0.002.

$$f(x)=e^{-(x-1)}sen(x)+1$$

NOTA: PARA EL ARGUMENTO DE LA FUNCIÓN SENOIDAL, CONVERTIR EL VALOR DE *X* A RADIANES. ES DECIR.

$$f(x)=e^{-(x-1)}sen\left(\frac{x\pi}{180}\right)+1$$

Solución

Se obtiene la primera derivada de la función

$$f'(x)=e^{-(x-1)}(\cos(x)-sen(x))$$

Debido a que se está convirtiendo el argumento a radianes, la primera derivada que se debe utilizar es:

$$f'(x)=e^{-(x-1)}\left(\frac{\pi}{180}\cos\left(\frac{x\pi}{180}\right)-sen\left(\frac{x\pi}{180}\right)\right)$$

Iteración 1

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}=-3-\frac{-1.8574}{3.8091}=-2.5124$$

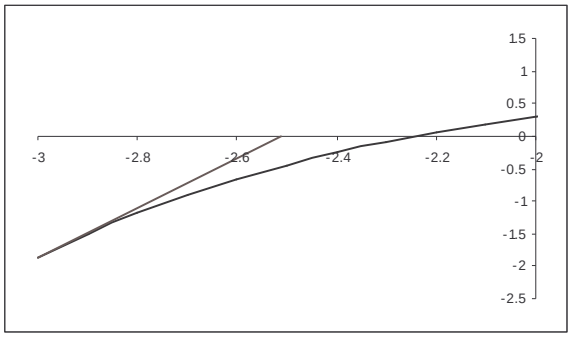


FIGURA 2.11 Gráfica de la iteración 1 del ejemplo 2.5.

Iteración 2

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = -2.5124 - \frac{-0.4697}{2.0543} = -2.2837$$

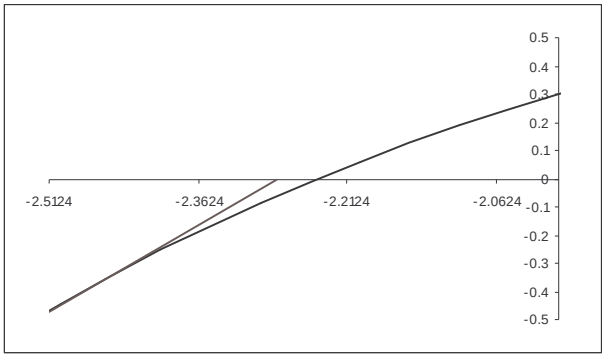


FIGURA 2.12 Gráfica de la iteración 2 del ejemplo 2.5.

Iteración 3

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = -2.2837 - \frac{-0.06295}{1.5281} = -2.2425$$

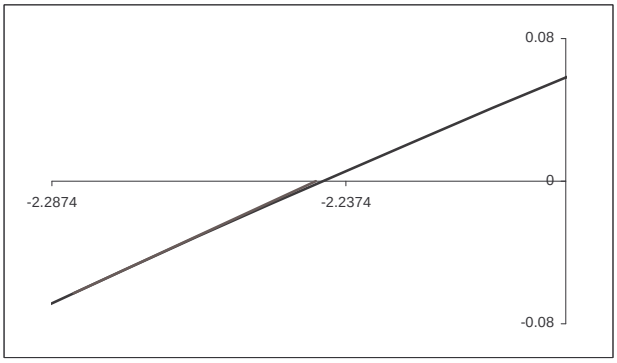


FIGURA 2.13 Gráfica de la iteración 3 del ejemplo 2.5.

Iteración 4

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = -2.2425 - \frac{-0.0017}{1.4481} = -2.2414$$

Como se cumple la condición, se detiene el método.

Se puede observar que el método de Newton-Raphson, a comparación con el método de bisección, hace menos iteraciones.

EJEMPLO 2.6 ELABORAR UN PROGRAMA QUE RESUELVA LA ECUACIÓN DEL EJEMPLO 2.4, UTILIZANDO EL MÉTODO DE NEWTON-RAPHSON.

-Algoritmo general del método de Newton-Raphson

x_0 = valor_inicial
REPETIR

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

HASTA $|x_{n+1} - x_n| <$ tolerancia o $f(x_{n+1}) = 0$

-Explicación de variables

tolerancia: Variable de tipo real, utilizada para almacenar la tolerancia permitida para finalizar el método.
x: Arreglo unidimensional de dos elementos de tipo real, utilizado para almacenar el valor de la iteración anterior y la actual.
i: Variable de tipo entero, utilizada para contar el número de iteraciones realizadas.

-Código

```
#include<stdio.h>
#include<math.h>

float funcion(float x);
float funcionD(float x);

void main(){
   float tolerancia,x[2];
   int i=0;
   printf("Dame el valor inicial: ");
   scanf("%g",&x[1]);
   printf("Dame el valor de tolerancia: ");
   scanf("%g",&tolerancia);

   do{
      x[0]=x[1];
      x[1]=x[0]-(funcion(x[0])/funcionD(x[0]));
      i++;
   }while(fabs(x[1]-x[0])>tolerancia);
   printf("El valor mas cercano a la raiz es: %.2f en la
          iteracion: %d",x[1],i);
}

float funcion(float x){
```

```
   float res;
   res=pow(x,4)+(5.8*pow(x,3))-(22.4*pow(x,2))-(31.2*x)+57.6;
   return res;
}

float funcionD(float x){
   float res;
   res=4*pow(x,3)+17.4*pow(x,2)-44.8*x-31.2;
   return res;
}
```

-Análisis del código

El programa inicia solicitando el valor inicial de *x*, así como la tolerancia permitida para finalizar el método.

```
printf("Dame el valor inicial: ");
scanf("%g",&x[1]);
printf("Dame el valor de tolerancia: ");
scanf("%g",&tolerancia);
```

Se ejecuta el algoritmo del método.

```
do{
   x[0]=x[1];
   x[1]=x[0]-(funcion(x[0])/funcionD(x[0]));
   i++;
}while(fabs(x[1]-x[0])>tolerancia);
```

El programa cuenta con dos funciones, una que devuelve el valor de la función en un punto dado, nombrada `funcion( )`, y la otra que devuelve el valor de la derivada de la función en un punto dado, nombrada `funcionD( )`.

```
float funcion(float x){
   float res;
   res=pow(x,4)+(5.8*pow(x,3))-(22.4*pow(x,2))-(31.2*x)+57.6;
   return res;
}

float funcionD(float x){
   float res;
   res=4*pow(x,3)+17.4*pow(x,2)-44.8*x-31.2;
   return res;
}
```

2.3 Solución numérica de sistemas de ecuaciones lineales

Generalmente, cuando se resuelven problemas de ingeniería, se obtienen varias ecuaciones que conforman un sistema, mismas que se deben resolver para obtener la solución adecuada al problema. Por esta razón, existen diversos métodos numéricos que resuelven de forma práctica y sencilla estas ecuaciones.

2.3.1 Método de eliminación Gaussiana

Un sistema de ecuaciones que contiene múltiples variables (x_1, x_2, …,x_n), se representa de la siguiente forma:

$$\begin{array}{ccccccccccccc}
a_{1,1.}x_1 & + & a_{1,2}x_2 & + & a_{1,3}x_3 & + & . & . & . & + & a_{1,n}x_n & = & b_1 \\
a_{2,1}x_1 & + & a_{2,2}x_2 & + & a_{2,3}x_3 & + & . & . & . & + & a_{2,n}x_n & = & b_2 \\
a_{3,1}x_1 & + & a_{3,2}x_2 & + & a_{3,3}x_3 & + & . & . & . & + & a_{3,n}x_n & = & b_3 \\
. & & . & & . & & & & & & . & & . \\
. & & . & & . & & & & & & . & & . \\
. & & . & & . & & & & & & . & & . \\
a_{n,1}x_1 & + & a_{n,2}x_2 & + & a_{n,3}x_3 & + & . & . & . & + & a_{n,n}x_n & = & b_n
\end{array}$$

Este sistema de ecuaciones, en forma matricial, se representa de la siguiente forma:

$$\begin{bmatrix}
a_{1,1} & a_{1,2} & a_{1,3} & . & . & . & a_{1,n} \\
a_{2,1} & a_{2,2} & a_{2,3} & . & . & . & a_{2,n} \\
a_{3,1} & a_{3,2} & a_{3,3} & . & . & . & a_{3,n} \\
. & . & . & & & & . \\
. & . & . & & & & . \\
. & . & . & & & & . \\
a_{n,1} & a_{n,2} & a_{n,3} & . & . & . & a_{n,n}
\end{bmatrix}
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix}
=
\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ . \\ . \\ . \\ b_n \end{bmatrix}$$

Empleando notación vectorial, resulta: $A\bar{x} = \bar{b}$

El objetivo del método de eliminación gaussiana, es triangular la matriz *A*, modificando con ello, los valores del vector *b*, de la siguiente forma:

$$\begin{bmatrix} a'_{1,1} & a'_{1,2} & a'_{1,3} & . & . & . & a'_{1,n} \\ 0 & a'_{2,2} & a'_{2,3} & . & . & . & a'_{2,n} \\ 0 & 0 & a'_{3,3} & . & . & . & a'_{3,n} \\ . & . & . & & & & . \\ . & . & . & & & & . \\ . & . & . & & & & . \\ 0 & 0 & 0 & . & . & . & a'_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ . \\ . \\ . \\ b'_n \end{bmatrix}$$

Cuando se triangula la matriz *A*, se utiliza álgebra matricial; las acciones que se pueden realizar sobre la matriz son:

Intercambiar renglones

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \xrightarrow{R_1 \leftrightarrow R_3} \begin{bmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix} \xrightarrow{R_2 \leftrightarrow R_3} \begin{bmatrix} 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Multiplicar un renglón por un escalar

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \xrightarrow{2R_1 \rightarrow R_1} \begin{bmatrix} 2 & 4 & 6 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \xrightarrow{6R_3 \rightarrow R_3} \begin{bmatrix} 2 & 4 & 6 \\ 4 & 5 & 6 \\ 42 & 48 & 54 \end{bmatrix}$$

Sustituir un renglón, por la suma de dos renglones

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \xrightarrow{(R_2+R_1) \rightarrow R_1} \begin{bmatrix} 5 & 7 & 9 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \xrightarrow{(2R_1-R_3) \rightarrow R_3} \begin{bmatrix} 5 & 7 & 9 \\ 4 & 5 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

Estas operaciones permiten triangular la matriz, utilizando todo un renglón como pivote para eliminar constantes. Antes de aplicar el álgebra matricial se debe crear lo que se le llama matriz extendida, es decir juntar la matriz A y el vector b.

Al convertir en una matriz triangular superior a la matriz A, queda un nuevo sistema de ecuaciones, en el que sólo se tiene que hacer una sustitución inversa, porque a partir de este punto se tiene el valor de x_n, y éste se despeja en el renglón superior, para obtener x_{n-1}, y así sucesivamente. Es decir:

$$x_n = \frac{b'_n}{a'_{n,n}}$$

$$x_{n-1} = \frac{b'_{n-1} - a'_{n-1,n}\, x_n}{a'_{n-1,n-1}}$$

.

.

.

$$x_1 = \frac{b'_1 - a'_{1,n}\, x_n - \ldots - a'_{1,2}\, x_2}{a'_{1,1}}$$

EJEMPLO 2.7 RESOLVER EL SIGUIENTE SISTEMA DE ECUACIONES, UTILIZANDO EL MÉTODO DE ELIMINACIÓN GAUSSIANA.

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 8 & 3 \\ 1 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 23.3 \\ 7.3 \\ 45 \end{bmatrix}$$

Solución

Se crea la matriz extendida

$$\begin{bmatrix} 1 & 2 & 3 & 23.3 \\ 2 & 8 & 3 & 7.3 \\ 1 & 1 & 5 & 45 \end{bmatrix}$$

Se aplica el álgebra matricial, para triangular la matriz *A*.

$$\begin{bmatrix} 1 & 2 & 3 & 23.3 \\ 2 & 8 & 3 & 7.3 \\ 1 & 1 & 5 & 45 \end{bmatrix} \xrightarrow[(R_3\text{-}R_1)\rightarrow R_3]{(R_2\text{-}2R_1)\rightarrow R_2} \begin{bmatrix} 1 & 2 & 3 & 23.3 \\ 0 & 4 & -3 & -39.3 \\ 0 & -1 & 2 & 21.7 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 & 23.3 \\ 0 & 4 & -3 & -39.3 \\ 0 & -1 & 2 & 21.7 \end{bmatrix} \xrightarrow{(R_3\text{-}\frac{1}{4}R_2)\rightarrow R_3} \begin{bmatrix} 1 & 2 & 3 & 23.3 \\ 0 & 4 & -3 & -39.3 \\ 0 & 0 & 1.25 & 11.88 \end{bmatrix}$$

De este nuevo sistema, ahora se obtienen los valores de las incógnitas, haciendo una sustitución inversa, es decir x_3 se obtiene del último renglón de la matriz extendida, x_2 del segundo renglón utilizando el valor obtenido de x_3, y así consecutivamente; obteniendo:

$$x_3 = \frac{11.88}{1.25} = 9.5$$
$$x_2 = \frac{-39.3 + 3x_3}{4} = \frac{-39.3 + 3(9.5)}{4} = -2.7$$
$$x_1 = \frac{23.3 - 2x_2 - 3x_3}{1} = \frac{23.3 - 2(-2.7) - 3(9.5)}{1} = 0.2$$

EJEMPLO 2.8 RESOLVER EL SIGUIENTE SISTEMA DE ECUACIONES, UTILIZANDO EL MÉTODO DE ELIMINACIÓN GAUSSIANA.

$$\begin{bmatrix} -1 & -0.5 & 1 & 1 \\ 2 & 1 & 5 & 4 \\ 8 & 3 & 1 & 3 \\ 7 & 1 & 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -1.52 \\ 0.25 \\ -12.39 \\ 0.65 \end{bmatrix}$$

Solución

Se obtiene la matriz extendida

$$\begin{bmatrix} -1 & -0.5 & 1 & 1 & -1.52 \\ 2 & 1 & 5 & 4 & 0.25 \\ 8 & 3 & 1 & 3 & -12.39 \\ 7 & 1 & 3 & 2 & 0.65 \end{bmatrix}$$

Se empieza a aplicar el método

$$\begin{bmatrix} -1 & -0.5 & 1 & 1 & -1.52 \\ 2 & 1 & 5 & 4 & 0.25 \\ 8 & 3 & 1 & 3 & -12.39 \\ 7 & 1 & 3 & 2 & 0.65 \end{bmatrix} \xrightarrow[(R_4+7R_1)\rightarrow R_4]{\begin{array}{l}(R_2+2R_1)\rightarrow R_2 \\ (R_3+8R_1)\rightarrow R_3\end{array}} \begin{bmatrix} -1 & -0.5 & 1 & 1 & -1.52 \\ 0 & 0 & 7 & 6 & -2.79 \\ 0 & -1 & 9 & 11 & -24.55 \\ 0 & -2.5 & 10 & 9 & -9.99 \end{bmatrix}$$

Como se puede observar, el siguiente coeficiente $a_{2,2}$ es cero, por lo que no se puede utilizar este renglón como pivote, lo que se hará en este caso para solucionar el problema es intercambiar el segundo con el tercer renglón y continuar con el método

$$\begin{bmatrix} -1 & -0.5 & 1 & 1 & -1.52 \\ 0 & -1 & 9 & 11 & -24.55 \\ 0 & 0 & 7 & 6 & -2.79 \\ 0 & -2.5 & 10 & 9 & -9.99 \end{bmatrix} \xrightarrow{(R_4\text{-}2.5R_2)\rightarrow R_4} \begin{bmatrix} -1 & -0.5 & 1 & 1 & -1.52 \\ 0 & -1 & 9 & 11 & -24.55 \\ 0 & 0 & 7 & 6 & -2.79 \\ 0 & 0 & -12.5 & -18.5 & 51.385 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -0.5 & 1 & 1 & -1.52 \\ 0 & -1 & 9 & 11 & -24.55 \\ 0 & 0 & 7 & 6 & -2.79 \\ 0 & 0 & -12.5 & -18.5 & 51.385 \end{bmatrix} \xrightarrow{(R_4\text{-}\frac{12.5}{7}R_3)\rightarrow R_4} \begin{bmatrix} -1 & -0.5 & 1 & 1 & -1.52 \\ 0 & -1 & 9 & 11 & -24.55 \\ 0 & 0 & 7 & 6 & -2.79 \\ 0 & 0 & 0 & -7.7857 & 46.4029 \end{bmatrix}$$

Ahora se obtiene la solución con sustitución inversa:

$$x_4 = \frac{46.4029}{-7.7857} = -5.96$$

$$x_3 = \frac{-2.79 - 6x_4}{7} = \frac{-2.79 - 6(-5.96)}{7} = 4.71$$

$$x_2 = \frac{-24.55 - 11x_4 - 9x_3}{-1} = \frac{-24.55 - 11(-5.96) - 9(4.71)}{-1} = 1.38$$

$$x_1 = \frac{-1.52 - x_4 - x_3 + 0.5x_2}{-1} = \frac{-1.52 + 5.96 - 4.71 + 0.5(1.38)}{-1} = -0.42$$

Cabe hacer notar, que el uso de decimales en las operaciones del método, propicia errores que pueden ser por redondeo o truncamiento.

EJEMPLO 2.9 ELABORAR UN PROGRAMA QUE RESUELVA EL SISTEMA DE ECUACIONES DEL EJEMPLO 2.7, UTILIZANDO EL MÉTODO DE ELIMINACIÓN GAUSSIANA.

-Algoritmo para la triangulación de un sistema de *n* ecuaciones, con *n* incógnitas

```
DESDE   i =  1 HASTA  n
  DESDE   j = i+1 HASTA  n
    cte = A(j,i)/A(i,i)
    DESDE  k = i  HASTA  n
      A(j,k) = A(j,k)-cte*A(i,k)
    FIN
    B(j) = B(j)-cte*B(i)
  FIN
FIN
```

-Explicación de variables

A: Arreglo bidimensional de tipo real, utilizado para almacenar los coeficientes de las variables del sistema.
b: Arreglo unidimensional de tipo real, utilizado para almacenar los términos independientes del sistema.
cte: Variable de tipo real, utilizada para multiplicar el vector pivote y restarlo al renglón que se desea eliminar.
x: Arreglo de tipo real, utilizado para almacenar la solución del sistema.
i: Variable de tipo entero, utilizada como control en ciclos y como índice del arreglo.
j: Variable de tipo entero, utilizada como control en ciclos y como índice del arreglo.
k: Variable de tipo entero, utilizada como control en ciclos y como índice del arreglo.
MAX: Variable de tipo entero, utilizada como control en ciclos.

-Código

```
#include <stdio.h>
void main(){
   float A[3][3],b[3],cte,x[3];
   int i,j,k,MAX=3;

   for(i=0;i<MAX;i++){
      for(j=0;j<MAX;j++){
         printf("Dame el valor de A(%d,%d): ",i,j);
         scanf("%g",&A[i][j]);
      }
      printf("Dame el valor de b(%d): ",i);
      scanf("%g",&b[i]);
   }

   for(i=0;i<MAX;i++){
      for(j=i+1;j<MAX;j++){
         cte=A[j][i]/A[i][i];
         for(k=i;k<MAX;k++){
            A[j][k]=A[j][k]-(cte*A[i][k]);
         }
         b[j]=b[j]-(cte*b[i]);
      }
   }

   printf("\n");
   for(i=0;i<MAX;i++){
      for(j=0;j<MAX;j++){
         printf("%.2f\t",A[i][j]);
      }
      printf("%.2f\n",b[i]);
   }
```

```
   for(i=0;i<MAX;i++){
      x[i]=0;
   }

   for(i=(MAX-1);i>-1;i--){
      for(j=(MAX-1);j>i;j--){
         x[i]=x[i]+(x[j]*A[i][j]);
      }
      x[i]=(b[i]-x[i])/A[i][i];
   }
   printf("\n\n");
   for(i=0;i<MAX;i++){
      printf("x(%d) = %.2f\n",i,x[i]);
   }
}
```

-Análisis del código

El programa inicia con un ciclo de lectura de la matriz *A* y los términos independientes.

```
for(i=0;i<MAX;i++){
   for(j=0;j<MAX;j++){
      printf("Dame el valor de A(%d,%d): ",i,j);
      scanf("%g",&A[i][j]);
   }
   printf("Dame el valor de b(%d): ",i);
   scanf("%g",&b[i]);
}
```

Se tiene un ciclo de triangulación, que consiste en ir seleccionando un renglón pivote, indicado por el índice *i*. Con el índice *j*, se referencian el resto de los renglones; a los elementos de cada renglón, indicados por *A[j][k]*, se les aplican las operaciones algebraicas para triangular la matriz *A*. Por último se modifica el vector *b* utilizando el índice *j*.

```
for(i=0;i<MAX;i++){
   for(j=i+1;j<MAX;j++){
      cte=A[j][i]/A[i][i];
      for(k=i;k<MAX;k++){
         A[j][k]=A[j][k]-(cte*A[i][k]);
      }
      b[j]=b[j]-(cte*b[i]);
   }
}
```

Una vez triangulada la matriz *A*, se tiene un ciclo que permite encontrar la solución del sistema haciendo la sustitución inversa; el índice *i* selecciona el valor de *x* a obtener, y con el índice *j* se va sumando en x_i la multiplicación de los elementos de *x* obtenidos, por los valores de la matriz triangulada. Al final la suma se resta a b_i y se divide entre $a_{i,i}$.

```
for(i=(MAX-1);i>-1;i--){
   for(j=(MAX-1);j>i;j--){
      x[i]=x[i]+(x[j]*A[i][j]);
   }
   x[i]=(b[i]-x[i])/A[i][i];
}
```

2.3.2 Método de Gauss-Jordan

El método de Gauss-Jordan, es muy similar en su funcionamiento comparado con el método de eliminación gaussiana, porque de igual forma, se manipula la matriz *A* para poder obtener la solución del sistema, la única diferencia radica en que ahora no se busca triangular a la matriz *A*, ya que se pretende dejarla como si fuera una matriz identidad, es decir, con coeficientes de 1 en su diagonal principal. Esto se logra normalizando el renglón pivote.

Entonces, visto de esta forma, lo que se pretende encontrar es:

$$\begin{bmatrix} 1 & 0 & 0 & . & . & . & 0 \\ 0 & 1 & 0 & . & . & . & 0 \\ 0 & 0 & 1 & . & . & . & 0 \\ . & . & . & & & & . \\ . & . & . & & & & . \\ . & . & . & & & & . \\ 0 & 0 & 0 & . & . & . & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ . \\ . \\ . \\ b'_n \end{bmatrix}$$

EJEMPLO 2.10 RESOLVER EL SIGUIENTE SISTEMA DE ECUACIONES, UTILIZANDO EL MÉTODO DE GAUSS-JORDAN.

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 8 & 3 \\ 1 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 23.3 \\ 7.3 \\ 45 \end{bmatrix}$$

Solución

Se obtiene la matriz extendida

$$\begin{bmatrix} 1 & 2 & 3 & 23.3 \\ 2 & 8 & 3 & 7.3 \\ 1 & 1 & 5 & 45 \end{bmatrix}$$

Como el primer renglón se encuentra normalizado; es decir, el coeficiente $a_{1,1}$ es uno, se procede a eliminar los demás coeficientes de la columna

$$\begin{bmatrix} 1 & 2 & 3 & 23.3 \\ 2 & 8 & 3 & 7.3 \\ 1 & 1 & 5 & 45 \end{bmatrix} \xrightarrow[(R_3\text{-}R_1)\rightarrow R_3]{(R_2\text{-}2R_1)\rightarrow R_2} \begin{bmatrix} 1 & 2 & 3 & 23.3 \\ 0 & 4 & -3 & -39.3 \\ 0 & -1 & 2 & 21.7 \end{bmatrix}$$

Como el siguiente pivote, es decir el coeficiente $a_{2,2}$ no es 1, se normaliza el segundo renglón

$$\begin{bmatrix} 1 & 2 & 3 & 23.3 \\ 0 & 1 & -0.75 & -9.825 \\ 0 & -1 & 2 & 21.7 \end{bmatrix}$$

Ahora, utilizando el segundo renglón como pivote, eliminamos los elementos de la segunda columna

$$\begin{bmatrix} 1 & 2 & 3 & 23.3 \\ 0 & 1 & -0.75 & -9.825 \\ 0 & -1 & 2 & 21.7 \end{bmatrix} \xrightarrow[(R_3+R_2)\rightarrow]{(R_1\text{-}2R_2)\rightarrow R_1} \begin{bmatrix} 1 & 0 & 4.5 & 42.95 \\ 0 & 1 & -0.75 & -9.825 \\ 0 & 0 & 1.25 & 11.875 \end{bmatrix}$$

A continuación se normaliza el tercer renglón

$$\begin{bmatrix} 1 & 0 & 4.5 & 42.95 \\ 0 & 1 & -0.75 & -9.825 \\ 0 & 0 & 1 & 9.5 \end{bmatrix}$$

Y utilizando este tercer renglón, se termina de diagonalizar la matriz

$$\begin{bmatrix} 1 & 0 & 4.5 & 42.95 \\ 0 & 1 & -0.75 & -9.825 \\ 0 & 0 & 1 & 9.5 \end{bmatrix} \xrightarrow[(R_2+0.75R_3)\rightarrow R_2]{(R_1\text{-}4.5R_3)\rightarrow R_1} \begin{bmatrix} 1 & 0 & 0 & 0.2 \\ 0 & 1 & 0 & -2.7 \\ 0 & 0 & 1 & 9.5 \end{bmatrix}$$

De esta matriz resultante se obtiene directamente la solución del sistema de ecuaciones

$x_1 = 0.2$

$x_2 = -2.7$

$x_3 = 9.5$

EJEMPLO 2.11 RESOLVER EL SIGUIENTE SISTEMA DE ECUACIONES, UTILIZANDO EL MÉTODO DE GAUSS-JORDAN.

$$\begin{bmatrix} -1 & -0.5 & 1 & 1 \\ 2 & 1 & 5 & 4 \\ 8 & 3 & 1 & 3 \\ 7 & 1 & 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -1.52 \\ 0.25 \\ -12.39 \\ 0.65 \end{bmatrix}$$

Solución

Se obtiene la matriz extendida

$$\begin{bmatrix} -1 & -0.5 & 1 & 1 & -1.52 \\ 2 & 1 & 5 & 4 & 0.25 \\ 8 & 3 & 1 & 3 & -12.39 \\ 7 & 1 & 3 & 2 & 0.65 \end{bmatrix}$$

Se normaliza el primer renglón y se comienza el método:

$$\begin{bmatrix} 1 & 0.5 & -1 & -1 & 1.52 \\ 2 & 1 & 5 & 4 & 0.25 \\ 8 & 3 & 1 & 3 & -12.39 \\ 7 & 1 & 3 & 2 & 0.65 \end{bmatrix} \xrightarrow[(R_4\text{-}7R_1)\rightarrow R_4]{\substack{(R_2\text{-}2R_1)\rightarrow R_2 \\ (R_3\text{-}8R_1)\rightarrow R_3}} \begin{bmatrix} 1 & 0.5 & -1 & -1 & 1.52 \\ 0 & 0 & 7 & 6 & -2.79 \\ 0 & -1 & 9 & 11 & -24.55 \\ 0 & -2.5 & 10 & 9 & -9.99 \end{bmatrix}$$

Como el siguiente renglón pivote, en este caso el segundo, tiene como coeficiente cero, empleando el álgebra matricial, se intercambian el segundo renglón con el tercero para continuar el método. Por lo tanto, se normaliza el nuevo renglón y se continúa el método:

$$\begin{bmatrix} 1 & 0.5 & -1 & -1 & 1.52 \\ 0 & 1 & -9 & -11 & 24.55 \\ 0 & 0 & 7 & 6 & -2.79 \\ 0 & -2.5 & 10 & 9 & -9.99 \end{bmatrix} \xrightarrow[(R_4+2.5R_2)\rightarrow R_4]{(R_1-0.5R_2)\rightarrow R_1,\ (R_3-0R_2)\rightarrow R_3} \begin{bmatrix} 1 & 0 & 3.5 & 4.5 & -10.755 \\ 0 & 1 & -9 & -11 & 24.55 \\ 0 & 0 & 7 & 6 & -2.79 \\ 0 & 0 & -12.5 & -18.5 & 51.385 \end{bmatrix}$$

Ahora se normaliza el siguiente renglón pivote y se continúa el método:

$$\begin{bmatrix} 1 & 0 & 3.5 & 4.5 & -10.755 \\ 0 & 1 & -9 & -11 & 24.55 \\ 0 & 0 & 1 & 0.8571 & -0.3986 \\ 0 & 0 & -12.5 & -18.5 & 51.385 \end{bmatrix} \xrightarrow[(R_4+12.5R_3)\rightarrow R_4]{(R_1-3.5R_3)\rightarrow R_1,\ (R_2+9R_3)\rightarrow R_3} \begin{bmatrix} 1 & 0 & 0 & 1.5 & -9.36 \\ 0 & 1 & 0 & -3.2857 & 20.9626 \\ 0 & 0 & 1 & 0.8571 & -0.3986 \\ 0 & 0 & 0 & -7.7857 & 46.40286 \end{bmatrix}$$

Se continúa con el cuarto renglón pivote:

$$\begin{bmatrix} 1 & 0 & 0 & 1.5 & -9.36 \\ 0 & 1 & 0 & -3.2857 & 20.9626 \\ 0 & 0 & 1 & 0.8571 & -0.3986 \\ 0 & 0 & 0 & 1 & -5.96 \end{bmatrix} \xrightarrow[(R_3+-0.8571R_4)\rightarrow R_3]{(R_1-1.5R_4)\rightarrow R_1,\ (R_2+3.2857R_4)\rightarrow R_3} \begin{bmatrix} 1 & 0 & 0 & 0 & -0.42 \\ 0 & 1 & 0 & 0 & 1.38 \\ 0 & 0 & 1 & 0 & 4.71 \\ 0 & 0 & 0 & 1 & -5.96 \end{bmatrix}$$

Donde la solución de este sistema de ecuaciones es:

$x_1 = -0.42$
$x_2 = 1.38$
$x_3 = 4.71$
$x_4 = -5.96$

EJEMPLO 2.12 ELABORAR UN PROGRAMAQUE RESUELVA EL SISTEMA DE ECUACIONES DEL EJEMPLO 2.10, UTILIZANDO EL MÉTODO DE GAUSS-JORDAN.

- Algoritmo general del método de Gauss-Jordan

```
DESDE  i =  1 HASTA  n
  Se normaliza el renglón pivote
  cte = A(i,i)
  DESDE  j = 1 HASTA  n
    A(i,j) = A(i,j)/cte
  FIN
  B(i)=B(i)/cte

  Sigue el algoritmo de eliminación gaussiana
  DESDE  j = i+1 HASTA  n
    cte = A(j,i)
```

```
    DESDE  k = 0 HASTA  n
      A(j,k) = A(j,k)-cte*A(i,k)
    FIN
    B(j) = B(j)-cte*B(i)
  FIN
FIN

Se realiza la eliminación inversa
DESDE  i = n  hasta 1
  DESDE  j = (i-1) hasta 1
    cte = A(j,i)
    A(j,i) = A(j,i)-cte*A(i,i)
    B(j) = B(j)-cte*B(i)
  FIN
FIN
```

-Explicación de variables

A: Arreglo bidimensional de tipo real, utilizado para almacenar los coeficientes de las variables del sistema.
b: Arreglo unidimensional de tipo real, utilizado para almacenar los términos independientes del sistema.
cte: Variable de tipo real, utilizada para multiplicar el vector pivote y restarlo al renglón que se desea eliminar.
x: Arreglo de tipo real, utilizado para almacenar la solución del sistema.
i: Variable de tipo entero, utilizada como control en ciclos y como índice del arreglo.
j: Variable de tipo entero, utilizada como control en ciclos y como índice del arreglo.
k: Variable de tipo entero, utilizada como control en ciclos y como índice del arreglo.
MAX: Variable de tipo entero, utilizada como control en ciclos.

-Código

```
#include <stdio.h>

void main(){
   float A[3][3],b[3],cte,x[3];
   int i,j,k,MAX=3;

   /*Lectura de la matriz A y del vector b */
   for(i=0;i<MAX;i++){
      for(j=0;j<MAX;j++){
         printf("Dame el valor de A(%d,%d): ",i,j);
         scanf("%g",&A[i][j]);
      }
      printf("Dame el valor de b(%d): ",i);
      scanf("%g",&b[i]);
   }
```

```
   for(i=0;i<MAX;i++){
      /*Se normaliza renglon pivote*/
      cte=A[i][i];
      for(j=0;j<MAX;j++){
         A[i][j]=A[i][j]/cte;
      }
      b[i]=b[i]/cte;

      /*Algoritmo Eliminacion Gaussiana*/
      for(j=(i+1);j<MAX;j++){
         cte=A[j][i];
         for(k=0;k<MAX;k++){
            A[j][k]=A[j][k]-(cte*A[i][k]);
         }
         b[j]=b[j]-(cte*b[i]);
      }
   }

   /*Eliminacion inversa para completar el metodo*/
   for(i=(MAX-1);i>0;i--){
      for(j=(i-1);j>=0;j--){
         cte=A[j][i];
         A[j][i]=A[j][i]-(cte*A[i][i]);
         b[j]=b[j]-(cte*b[i]);
      }
   }

   /*Se imprime la matriz resultante con la solucion del
     sistema */
   for(i=0;i<MAX;i++){
      for(j=0;j<MAX;j++){
         printf("%.2f\t",A[i][j]);
      }
      printf("%.2f\n",b[i]);
   }
}
```

-Análisis del código

El programa inicia con un ciclo de obtención de los valores de la matriz *A*, así como de los valores del vector *b*.

```
for(i=0;i<MAX;i++){
   for(j=0;j<MAX;j++){
      printf("Dame el valor de A(%d,%d): ",i,j);
      scanf("%g",&A[i][j]);
```

```
   }
   printf("Dame el valor de b(%d): ",i);
   scanf("%g",&b[i]);
}
```

Se inicia el método, la forma en que se propone la solución del método con este ejemplo se divide en dos partes, en la primera parte se triangula utilizando el método de Eliminación Gaussiana, a continuación se vuelve a utilizar el algoritmo pero ahora para "triangular" la parte superior. Al mismo tiempo se van modificando los valores del vector b, que contendrá la solución del sistema al final del algoritmo.

```
for(i=0;i<MAX;i++){
   /*Se normaliza renglon pivote*/
   cte=A[i][i];
   for(j=0;j<MAX;j++){
      A[i][j]=A[i][j]/cte;
   }
   b[i]=b[i]/cte;

   /*Algoritmo Eliminacion Gaussiana*/
   for(j=(i+1);j<MAX;j++){
      cte=A[j][i];
      for(k=0;k<MAX;k++){
         A[j][k]=A[j][k]-(cte*A[i][k]);
      }
b[j]=b[j]-(cte*b[i]);
   }
}

/*Eliminacion inversa para completar el metodo*/
for(i=(MAX-1);i>0;i--){
   for(j=(i-1);j>=0;j--){
      cte=A[j][i];
      A[j][i]=A[j][i]-(cte*A[i][i]);
      b[j]=b[j]-(cte*b[i]);
   }
}
```

Al final del algoritmo, se tiene un ciclo de impresión de la matriz resultante; y el vector solución.

```
for(i=0;i<MAX;i++){
   for(j=0;j<MAX;j++){
      printf("%.2f\t",A[i][j]);
   }
   printf("%.2f\n",b[i]);
}
```

2.3.3 Método de LU

Hasta el momento se han visto métodos en los que se involucra una sola matriz y se realizan operaciones básicas sobre ella.

En el método de LU, lo que se pretende es generar dos matrices (L y U) a partir de la matriz A, es decir $A = LU$. La característica principal de estas matrices es que son triangulares, la matriz L es triangular inferior, y la matriz U es triangular superior. Para facilidad de cálculos, la matriz L o la matriz U, puede contener solo unos en su diagonal principal. Por ejemplo:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n} \end{bmatrix} = \begin{bmatrix} l_{1,1} & 0 & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & 0 & \cdots & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ l_{n,1} & l_{n,2} & l_{n,n} & \cdots & l_{n,n} \end{bmatrix} \begin{bmatrix} 1 & u_{1,2} & u_{1,3} & \cdots & u_{1,n} \\ 0 & 1 & u_{2,3} & \cdots & u_{2,n} \\ 0 & 0 & 1 & \cdots & u_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

En este caso, la matriz U contendrá unos en su diagonal principal.

Una vez obtenidas las matrices L y U, se hacen artificios matemáticos, sustituyendo las nuevas matrices en $A\bar{x} = \bar{b}$, obteniendo $LU\bar{x} = \bar{b}$.

Ahora se supone $U\bar{x} = \bar{y}$, y sustituyendo esta última ecuación en $LU\bar{x} = \bar{b}$, se obtiene $L\bar{y} = \bar{b}$.

Utilizando los valores obtenidos de $\bar{y}$, de la ecuación $U\bar{x} = \bar{y}$, se obtiene la solución del sistema.

Para comprender la forma en que se obtienen las matrices L y U, así como $\bar{y}$, se partirá de un sistema de 3 variables:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} = \begin{bmatrix} l_{1,1} & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} \end{bmatrix} \begin{bmatrix} 1 & u_{1,2} & u_{1,3} \\ 0 & 1 & u_{2,3} \\ 0 & 0 & 1 \end{bmatrix}$$

Desarrollando la multiplicación de L y U, se tiene que:

$$\begin{array}{lll} a_{1,1} = l_{1,1} & a_{1,2} = l_{1,1}u_{1,2} & a_{1,3} = l_{1,1}u_{1,3} \\ a_{2,1} = l_{2,1} & a_{2,2} = l_{2,1}u_{1,2} + l_{2,2} & a_{2,3} = l_{2,1}u_{1,3} + l_{2,2}u_{2,3} \\ a_{3,1} = l_{3,1} & a_{3,2} = l_{3,1}u_{1,2} + l_{3,2} & a_{3,3} = l_{3,1}u_{1,3} + l_{3,2}u_{2,3} + l_{3,3} \end{array}$$

Con base en estas ecuaciones, se pueden obtener los valores de *L* y *U* de la siguiente forma:

$$l_{1,1} = a_{1,1} \qquad u_{1,2} = \frac{a_{1,2}}{l_{1,1}} \qquad u_{1,3} = \frac{a_{1,3}}{l_{1,1}}$$

$$l_{2,1} = a_{2,1} \qquad l_{2,2} = a_{2,2} - l_{2,1}u_{1,2} \qquad u_{2,3} = \frac{a_{2,3} - l_{2,1}u_{1,3}}{l_{2,2}}$$

$$l_{3,1} = a_{3,1} \qquad l_{3,2} = a_{3.2} - l_{3,1}u_{1,2} \qquad l_{3,3} = a_{3,3} - l_{3,1}u_{1,3} - l_{3,2}u_{2,3}$$

Con los valores obtenidos para la matriz *L*, se pueden calcular los valores de $\bar{y}$:

$$\begin{bmatrix} l_{1,1} & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix},$$

de donde:

$$l_{1,1}y_1 = b_1 \quad \rightarrow \quad y_1 = \frac{b_1}{l_{1,1}}$$

$$l_{2,1}y_1 + l_{2,2}y_2 = b_2 \quad \rightarrow \quad y_2 = \frac{b_2 - l_{2,1}y_1}{l_{2,2}}$$

$$l_{3,1}y_1 + l_{3,2}y_2 + l_{3,3}y_3 = b_3 \quad \rightarrow \quad y_3 = \frac{b_3 - l_{3,1}y_1 - l_{3,2}y_2}{l_{3,3}}$$

Por último, de $U\bar{x} = \bar{y}$, se obtienen los valores de $\bar{x}$

$$\begin{bmatrix} 1 & u_{1,2} & u_{1,3} \\ 0 & 1 & u_{2,3} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix},$$

de donde:

$$x_1 + u_{1,2}x_2 + u_{1,3}x_3 = y_1 \quad \rightarrow \quad x_1 = y_1 - u_{1,2}x_2 - u_{1,3}x_3$$

$$x_2 + u_{2,3}x_3 = y_2 \quad \rightarrow \quad x_2 = y_2 - u_{2,3}x_3$$

$$x_3 = y_3$$

EJEMPLO 2.13 RESOLVER EL SIGUIENTE SISTEMA DE ECUACIONES, UTILIZANDO EL MÉTODO DE LU.

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 8 & 3 \\ 1 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 23.3 \\ 7.3 \\ 45 \end{bmatrix}$$

Solución

Utilizando las ecuaciones obtenidas anteriormente, se calcularan los valores de las matrices *L* y *U*.

$$l_{1,1} = a_{1,1} = 1 \qquad u_{1,2} = \frac{a_{1,2}}{l_{1,1}} = \frac{2}{1} = 2 \qquad u_{1,3} = \frac{a_{1,3}}{l_{1,1}} = \frac{3}{1} = 3$$

$$l_{2,1} = a_{2,1} = 2 \qquad l_{2,2} = a_{2,2} - l_{2,1}u_{1,2} = 8 - (2)(2) = 4 \qquad u_{2,3} = \frac{a_{2,3} - l_{2,1}u_{1,3}}{l_{2,2}} = \frac{3 - (2)(3)}{4} = -\frac{3}{4}$$

$$l_{3,1} = a_{3,1} = 1 \qquad l_{3,2} = a_{3.2} - l_{3,1}u_{1,2} = 1 - (1)(2) = -1 \qquad l_{3,3} = a_{3,3} - l_{3,1}u_{1,3} - l_{3,2}u_{2,3} = 5 - (1)(3) - (-1)(-0.75) = 1.25$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 4 & 0 \\ 1 & -1 & 1.25 \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -0.75 \\ 0 & 0 & 1 \end{bmatrix}$$

Con estos valores ahora se obtiene $\bar{y}$

$$y_1 = \frac{b_1}{l_{1,1}} = \frac{23.3}{1} = 23.3$$

$$y_2 = \frac{b_2 - l_{2,1}y_1}{l_{2,2}} = \frac{7.3 - (2)(23.3)}{4} = -9.82$$

$$y_3 = \frac{b_3 - l_{3,1}y_1 - l_{3,2}y_2}{l_{3,3}} = \frac{45 - (1)(23.3) - (-1)(-9.82)}{1.25} = 9.5$$

$$\bar{y} = \begin{bmatrix} 23.3 \\ -9.82 \\ 9.5 \end{bmatrix}$$

Por último se obtiene la solución a partir de $U\bar{x} = \bar{y}$

$x_3 = y_3 = 9.5$
$x_2 = y_2 - u_{2,3}x_3 = -9.82 - (-0.75)(9.5) = -2.7$
$x_1 = y_1 - u_{1,2}x_2 - u_{1,3}x_3 = 23.3 - (2)(-2.7) - (3)(9.5) = 0.2$

$$\bar{x} = \begin{bmatrix} 0.2 \\ -2.7 \\ 9.5 \end{bmatrix}$$

Fórmula general de LU

A continuación se presenta la forma general para obtener los valores de las matrices *L* y *U*
Para la diagonal principal:

$$u_{i,i} = 1$$
$$l_{i,i} = a_{i,i} - \sum_{s=1}^{i-1} \left(l_{i,s} * u_{s,i} \right)$$

Para los demás coeficientes de la matriz:

$$l_{i,j} = a_{i,j} - \sum_{s=1}^{j-1} \left(l_{i,s} * u_{s,j} \right)$$
$$u_{i,j} = \frac{a_{i,j} - \sum_{s=1}^{i-1} \left(l_{i,s} * u_{s,j} \right)}{l_{i,i}}$$

Haciendo un análisis comparativo de este método con los previamente vistos, se puede observar que al momento de separar la matriz *A* en *L* y *U*, se está aplicando el método de Eliminación Gaussiana sobre A, normalizando cada renglón pivote, y justamente las constantes por las que se multiplican los renglones pivote se almacenan en la matriz *L*. Por lo que se deduce de que si existe algún punto en el método de Eliminación Gaussiana en el que el sistema no se puede resolver por que se encuentra un cero, en la diagonal principal, este método no puede resolver el sistema hasta no iniciar con otro arreglo de los renglones de la matriz.

EJEMPLO 2.14 RESOLVER EL SIGUIENTE SISTEMA DE ECUACIONES, UTILIZANDO EL MÉTODO DE LU.

$$\begin{bmatrix} -1 & -0.5 & 1 & 1 \\ 2 & 1 & 5 & 4 \\ 8 & 3 & 1 & 3 \\ 7 & 1 & 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -1.52 \\ 0.25 \\ -12.39 \\ 0.65 \end{bmatrix}$$

Solución

Debido a que este sistema al ser resuelto por Eliminación Gaussiana, tiene el defecto de encontrar un cero en la diagonal principal, se reordenará . Se propone:

$$\begin{bmatrix} 8 & 3 & 1 & 3 \\ 2 & 1 & 5 & 4 \\ -1 & -0.5 & 1 & 1 \\ 7 & 1 & 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -12.39 \\ 0.25 \\ -1.52 \\ 0.65 \end{bmatrix}$$

Se inicia el método obteniendo la primera columna *L* y el primer renglón *U*

$$l_{1,1} = a_{1,1} = 8$$
$$l_{2,1} = a_{2,1} = 2$$
$$l_{3,1} = a_{3,1} = -1$$
$$l_{4,1} = a_{4,1} = 7$$

$$u_{1,2} = \frac{a_{1,2}}{l_{1,1}} = \frac{3}{8} = 0.375$$
$$u_{1,3} = \frac{a_{1,3}}{l_{1,1}} = \frac{1}{8} = 0.125$$
$$u_{1,4} = \frac{a_{1,4}}{l_{1,1}} = \frac{3}{8} = 0.375$$

Se continúa con la segunda columna de *L* y el segundo renglón de *U*

$$l_{2,2} = a_{2,2} - l_{2,1}u_{1,2} = 1 - (2)\left(\frac{3}{8}\right) = \frac{1}{4} = 0.25$$
$$l_{3,2} = a_{3,2} - l_{3,1}u_{1,2} = -\frac{1}{2} - (-1)\left(\frac{3}{8}\right) = -\frac{1}{8} = -0.125$$
$$l_{4,2} = a_{4,2} - l_{4,1}u_{1,2} = 1 - (7)\left(\frac{3}{8}\right) = -\frac{13}{8} = -1.625$$

$$u_{2,3} = \frac{a_{2,3} - l_{2,1}u_{1,3}}{l_{2,2}} = \frac{5 - (2)\left(\frac{1}{8}\right)}{\frac{1}{4}} = \frac{38}{2} = 19$$

$$u_{2,4} = \frac{a_{2,4} - l_{2,1}u_{1,4}}{l_{2,2}} = \frac{4 - (2)\left(\frac{3}{8}\right)}{\frac{1}{4}} = \frac{26}{2} = 13$$

Ahora se calcula la tercera columna de *L* y el tercer renglón de *U*

$$l_{3,3} = a_{3,3} - l_{3,1}u_{1,3} - l_{3,2}u_{2,3} = 1 - (-1)\left(\frac{1}{8}\right) - \left(-\frac{1}{8}\right)(19) = \frac{7}{2} = 3.5$$

$$l_{4,3} = a_{4,3} - l_{4,1}u_{1,3} - l_{4,2}u_{2,3} = 3 - (7)\left(\frac{1}{8}\right) - \left(-\frac{13}{8}\right)(19) = \frac{264}{8} = 33$$

$$u_{3,4} = \frac{a_{3,4} - l_{3,1}u_{1,4} - l_{3,2}u_{2,4}}{l_{3,3}} = \frac{1 - (-1)\left(\frac{3}{8}\right) - \left(-\frac{1}{8}\right)(13)}{\frac{7}{2}} = \frac{6}{7} = 0.85714$$

Por último se calcula el valor de $l_{4,4}$

$$l_{4,4} = a_{4,4} - l_{4,1}u_{1,4} - l_{4,2}u_{2,4} - l_{4,3}u_{3,4} = 2 - (7)\left(\frac{3}{8}\right) - \left(-\frac{13}{8}\right)(13) - (33)\left(\frac{6}{7}\right) = -\frac{109}{14} = -7.7857$$

Por lo que las matrices quedan:

$$L = \begin{bmatrix} 8 & 0 & 0 & 0 \\ 2 & 0.25 & 0 & 0 \\ -1 & -0.125 & 3.5 & 0 \\ 7 & -1.625 & 33 & -7.7857 \end{bmatrix} \qquad U = \begin{bmatrix} 1 & 0.375 & 0.125 & 0.375 \\ 0 & 1 & 19 & 13 \\ 0 & 0 & 1 & 0.85714 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ahora de $L\bar{y} = \bar{b}$, se obtienen los valores de $\bar{y}$

$$y_1 = \frac{b_1}{l_{1,1}} = \frac{-12.39}{8} = -1.54875$$

$$y_2 = \frac{b_2 - l_{2,1}y_1}{l_{2,2}} = \frac{0.25 - (2)(-1.54875)}{0.25} = \frac{3.3475}{0.25} = 13.39$$

$$y_3 = \frac{b_3 - l_{3,1}y_1 - l_{3,2}y_2}{l_{3,3}} = \frac{-1.52 - (-1)(-1.54875) - (-0.125)(13.39)}{3.5} = -0.39857$$

$$y_4 = \frac{b_4 - l_{4,1}y_1 - l_{4,2}y_2 - l_{4,3}y_3}{l_{4,4}} = \frac{0.65 - (7)(-1.54875) - (-1.625)(13.39) - (33)(-0.39857)}{-7.7857}$$

$$= -5.96$$

Quedando el vector *y* como:

$$\bar{y} = \begin{bmatrix} -1.54875 \\ 13.39 \\ -0.39857 \\ -3.1751 \end{bmatrix}$$

Por último se obtiene $\bar{x}$

$$x_4 = y_4 = -5.96$$
$$x_3 = y_3 - u_{3,4}x_4 = -0.39857 - (0.85714)(-5.96) = 4.71$$
$$x_2 = y_2 - u_{2,3}x_3 - u_{2,4}x_4 = 13.39 - (19)(4.71) - (13)(-5.96) = 1.38$$
$$x_1 = y_1 - u_{1,2}x_2 - u_{1,3}x_3 - u_{1,4}x_4 = -1.54875 - (0.375)(1.38) - (0.125)(4.71) - (0.375)(-5.96)$$
$$= -0.42$$

Por lo que la solución del sistema de ecuaciones es:

$$\bar{x} = \begin{bmatrix} -0.42 \\ 1.38 \\ 4.71 \\ -5.96 \end{bmatrix}$$

EJEMPLO 2.15 ELABORAR UN PROGRAMA QUE OBTENGA LA SOLUCIÓN DE UN SISTEMA DE ECUACIONES DE 3 VARIABLES, UTILIZANDO EL MÉTODO DE LU.

-Algoritmo de obtención de las matrices L y U

DESDE k = 1 HASTA n
  U(k,k)=1
  L(k,k) = $A(k,k) - \sum_{s=1}^{k-1} L(k,s) * U(s,k)$
  DESDE i = (k+1) HASTA n
    $L(i,k) = A(i,k) - \sum_{s=1}^{k-1} L(i,s) * U(s,k)$
  FIN
  DESDE j = (k+1) HASTA n
    $U(k,j) = \dfrac{A(k,j) - \sum_{s=1}^{k-1} L(k,s) * U(s,j)}{L(k,k)}$
  FIN
FIN

-Explicación de variables

A: Arreglo bidimensional de tipo real, utilizado para almacenar los coeficientes de las variables del sistema.
LU: Arreglo bidimensional de tipo real, utilizado para almacenar los coeficientes de las matrices L y U en esquema compacto.
b: Arreglo unidimensional de tipo real, utilizado para almacenar los términos independientes del sistema.
y: Arreglo unidimensional de tipo real, utilizado para almacenar los valores del vector *y* utilizado en el método de descomposición de LU.
x: Arreglo de tipo real, utilizado para almacenar la solución del sistema.
suma:
i: Variable de tipo entero, utilizada como control en ciclos y como índice del arreglo.
j: Variable de tipo entero, utilizada como control en ciclos y como índice del arreglo.
k: Variable de tipo entero, utilizada como control en ciclos y como índice del arreglo.
MAX: Variable de tipo entero, utilizada como control en ciclos.

-Código

```
#include<stdio.h>

void main(){
   float A[3][3],LU[3][3],b[3],y[3],x[3],suma;
   int i,j,k,MAX=3;

   /*Lectura de la matriz A y del vector b */
   for(i=0;i<MAX;i++){
      for(j=0;j<MAX;j++){
         printf("Dame el valor de A(%d,%d): ",i,j);
         scanf("%g",&A[i][j]);
      }
      printf("Dame el valor de b(%d): ",i);
      scanf("%g",&b[i]);
   }

   /*Se inician los valores de los elementos de la matriz LU
     con 0 */
   for(i=0;i<MAX;i++){
      for(j=0;j<MAX;j++){
         LU[i][j]=0;
      }
   }

   /*Se calculan los elementos de la matriz LU */
   for(i=0;i<MAX;i++){
      /*Calculo de la columna L*/
      for(j=i;j<MAX;j++){
```

```
         suma=0;
         for(k=0;k<j;k++){
            suma=suma+LU[j][k]*LU[k][i];
         }
         LU[j][i]=A[j][i]-suma;
         printf("L(%d,%d): %.2f\n",j,i,LU[j][i]);
      }

      /*Calculo de renglon U*/
      for(j=(i+1);j<MAX;j++){
         suma=0;
         for(k=0;k<i;k++){
            suma=suma+LU[i][k]*LU[k][j];
         }
         LU[i][j]=(A[i][j]-suma)/LU[i][i];
      }
   }

   /*Se imprime la matriz LU*/
   printf("\n\n");

   for(i=0;i<MAX;i++){
      for(j=0;j<MAX;j++){
         printf("%.3f\t",LU[i][j]);
      }
      printf("\n");
   }

   /*Se calculan los elementos del vector y */
   y[0]=b[0]/LU[0][0];
   printf("\n\ny(0)= %.3f\n",y[0]);

   y[1]=(b[1]-LU[1][0]*y[0])/LU[1][1];
   printf("y(1)= %.3f\n",y[1]);
   y[2]=(b[2]-LU[2][0]*y[0]-LU[2][1]*y[1])/LU[2][2];
   printf("y(2)= %.3f\n\n\n",y[2]);

   /* Se obtienen los valores del vector x */
   x[2]=y[2];
   x[1]=y[1]-LU[1][2]*x[2];
   x[0]=y[0]-LU[0][1]*x[1]-LU[0][2]*x[2];

   /* Se imprime la solución del sistema */
   for(i=(MAX-1);i>=0;i--){
      printf("x(%d)= %.2f\n",i,x[i]);
   }
}
```

-Análisis de código

Se inicia con un ciclo de lectura de los elementos de la matriz A, así como de los elementos del vector b.

```
for(i=0;i<MAX;i++){
   for(j=0;j<MAX;j++){
      printf("Dame el valor de A(%d,%d): ",i,j);
      scanf("%g",&A[i][j]);
   }
   printf("Dame el valor de b(%d): ",i);
   scanf("%g",&b[i]);
}
```

Para el uso eficiente de memoria, se utilizará una sola matriz (LU) de forma compacta para los elementos de las matrices L y U, es decir:

$$LU = \begin{bmatrix} l_{0,0} & u_{0,1} & u_{0,2} & u_{0,3} \\ l_{1,0} & l_{1,1} & u_{1,2} & u_{1,3} \\ l_{2,0} & l_{2,1} & l_{2,2} & u_{2,3} \\ l_{3,0} & l_{3,1} & l_{3,2} & l_{3,3} \end{bmatrix}$$

Por esta razón, en el algoritmo propuesto en este programa, cuando se calculan los valores de *L*, se utiliza el índice *j* como primer índice, y cuando se obtienen los valores de *U*, se utiliza como segundo índice.

```
for(i=0;i<MAX;i++){
   /*Calculo de la columna L*/
   for(j=i;j<MAX;j++){
      suma=0;
      for(k=0;k<j;k++){
         suma=suma+LU[j][k]*LU[k][i];
      }
      LU[j][i]=A[j][i]-suma;
      printf("L(%d,%d): %.2f\n",j,i,LU[j][i]);
   }


   /*Calculo de renglon U*/
   for(j=(i+1);j<MAX;j++){
      suma=0;
      for(k=0;k<i;k++){
         suma=suma+LU[i][k]*LU[k][j];
      }
      LU[i][j]=(A[i][j]-suma)/LU[i][i];
```

```
   }
}
```

Luego se imprime en pantalla la matriz LU resultante, y se implementan las ecuaciones para un sistema de 3x3 obtenidas anteriormente.

```
y[0]=b[0]/LU[0][0];
printf("\n\ny(0)= %.3f\n",y[0]);

y[1]=(b[1]-LU[1][0]*y[0])/LU[1][1];
printf("y(1)= %.3f\n",y[1]);

y[2]=(b[2]-LU[2][0]*y[0]-LU[2][1]*y[1])/LU[2][2];
printf("y(2)= %.3f\n\n\n",y[2]);

x[2]=y[2];
x[1]=y[1]-LU[1][2]*x[2];
x[0]=y[0]-LU[0][1]*x[1]-LU[0][2]*x[2];


for(i=(MAX-1);i>=0;i--){
   printf("x(%d)= %.2f\n",i,x[i]);
}
```

2.3.4 Método de Gauss-Seidel

Este método, a diferencia de los demás métodos vistos para la solución de sistemas de ecuaciones, es un método iterativo. El método consiste en despejar, de cada una de las ecuaciones, las variables del sistema; es decir, de la primer ecuación despejar x_1, de la segunda ecuación despejar x_2, y así consecutivamente. Posteriormente, a partir de una condición inicial de las variables, que generalmente es un valor inicial de cero, se van sustituyendo nuevos valores en cada ecuación generada.

A continuación se muestra el análisis para un sistema de 3 incógnitas.

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 &= b_2 \\ a_{3,1}x_1 + a_{3,2}x_2 + a_{3,3}x_3 &= b_3 \end{aligned}$$

Se despejan las variables de cada ecuación:

$$x_1 = \frac{b_1 - a_{1,2}x_2 - a_{1,3}x_3}{a_{1,1}}$$
$$x_2 = \frac{b_2 - a_{2,1}x_1 - a_{2,3}x_3}{a_{2,2}}$$
$$x_3 = \frac{b_3 - a_{3,1}x_1 - a_{3,2}x_2}{a_{3,3}}$$

Ahora, las iteraciones se realizan de la siguiente forma:

$$x_1(n) = \frac{b_1 - a_{1,2}x_2(n-1) - a_{1,3}x_3(n-1)}{a_{1,1}}$$
$$x_2(n) = \frac{b_2 - a_{2,1}x_1(n) - a_{2,3}x_3(n-1)}{a_{2,2}}$$
$$x_3(n) = \frac{b_3 - a_{3,1}x_1(n) - a_{3,2}x_2(n)}{a_{3,3}}$$

El método se detiene cuando todas las diferencias entre las variables de la iteración actual con la anterior, sean menor que una tolerancia definida; es decir:

$$\left(\left|x_1(n) - x_1(n-1)\right| < tol\right) \& \& \left(\left|x_2(n) - x_2(n-1)\right| < tol\right) \& \& \left(\left|x_3(n) - x_3(n-1)\right| < tol\right)$$

Nota. Existe una restricción muy importante que hay que hacer notar para que este método funcione: en la diagonal principal de la matriz de coeficientes *A*, deben estar los valores más grandes de los coeficientes.

EJEMPLO 2.16 RESOLVER EL SIGUIENTE SISTEMA DE ECUACIONES, UTILIZANDO EL MÉTODO DE GAUSS-SEIDEL. CONSIDERAR UNA TOLERANCIA DE 0.001.

$$\begin{array}{rcrcrcl} x_1 & + & 2x_2 & + & 3x_3 & = & 23.3 \\ 2x_1 & + & 8x_2 & + & 3x_3 & = & 7.3 \\ x_1 & + & x_2 & + & 5x_3 & = & 45 \end{array}$$

Solución

En este caso, como en la diagonal principal de la matriz de coeficientes *A* se encuentran los valores más grandes; se despejan las variables de las ecuaciones:

$$x_1 = 23.3 - 2x_2 - 3x_3$$
$$x_2 = \frac{7.3 - 2x_1 - 3x_3}{8}$$
$$x_3 = \frac{45 - x_1 - x_2}{5}$$

Partiendo de la condición inicial de cero, se realiza la primera iteración:

$$x_1(1) = 23.3 - 2x_2(0) - 3x_3(0) = 23.3 - 2(0) - 3(0) = 23.3$$
$$x_2(1) = \frac{7.3 - 2x_1(1) - 3x_3(0)}{8} = \frac{7.3 - 2(23.3) - 3(0)}{8} = \frac{-39.3}{8} = -4.91$$
$$x_3(1) = \frac{45 - x_1(1) - x_2(1)}{5} = \frac{45 - (23.3) - (-4.91)}{5} = \frac{26.61}{5} = 5.32$$

Verificando la condición para terminar las iteraciones:

$$\left(|x_1(1) - x_1(0)| < 0.001\right) \&\& \left(|x_2(1) - x_2(0)| < 0.001\right) \&\& \left(|x_3(1) - x_3(0)| < 0.001\right)$$
$$\left(|23.3 - 0| < 0.001\right) \&\& \left(|-4.91 - 0| < 0.001\right) \&\& \left(|5.32 - 0| < 0.001\right) \Rightarrow \textit{FALSO}$$

Se puede observar que no se cumple, por lo que se continúa:

$$x_1(2) = 23.3 - 2x_2(1) - 3x_3(1) = 23.3 - 2(-4.91) - 3(5.32) = 17.16$$
$$x_2(2) = \frac{7.3 - 2x_1(2) - 3x_3(1)}{8} = \frac{7.3 - 2(17.16) - 3(5.32)}{8} = \frac{-39.3}{8} = -5.37$$
$$x_3(2) = \frac{45 - x_1(2) - x_2(2)}{5} = \frac{45 - (17.16) - (-5.37)}{5} = \frac{26.61}{5} = 6.64$$

Se verifica la condición:

$$\left(|x_1(2) - x_1(1)| < 0.001\right) \&\& \left(|x_2(2) - x_2(1)| < 0.001\right) \&\& \left(|x_3(2) - x_3(1)| < 0.001\right)$$
$$\left(|17.16 - 23.3| < 0.001\right) \&\& \left(|-5.37 - (-4.91)| < 0.001\right) \&\& \left(|6.64 - 5.32| < 0.001\right) \Rightarrow \textit{FALSO}$$

Al no cumplirse, se sigue iterando.

En la iteración 51, se tiene:

$$x_1(51) = 0.2050839$$
$$x_2(51) = \text{-}2.700901096$$
$$x_3(51) = 9.499163439$$

En este punto, se cumple computacionalmente, que la diferencia entre las iteraciones 50 y 51, son menores que 0.001.

EJEMPLO 2.17 RESOLVER EL SIGUIENTE SISTEMA DE ECUACIONES, UTILIZANDO EL MÉTODO DE GAUSS-SEIDEL. CONSIDERE UNA TOLERANCIA DE 0.001.

$$\begin{bmatrix} -1 & -0.5 & 1 & 1 \\ 2 & 1 & 5 & 4 \\ 8 & 3 & 1 & 3 \\ 7 & 1 & 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -1.52 \\ 0.25 \\ -12.39 \\ 0.65 \end{bmatrix}$$

Solución

Como se puede observar, no se cumple la restricción que en la diagonal principal se encuentren los valores más grandes; por lo que haciendo, un intercambio de renglones de la matriz, el nuevo sistema queda:

$$\begin{bmatrix} 7 & 1 & 3 & 2 \\ 8 & 3 & 1 & 3 \\ 2 & 1 & 5 & 4 \\ -1 & -0.5 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0.65 \\ -12.39 \\ 0.25 \\ -1.52 \end{bmatrix}$$

Por lo que despejando las variables de cada una de las ecuaciones, se tiene:

$$x_1 = \frac{0.65 - x_2 - 3x_3 - 2x_4}{7}$$
$$x_2 = \frac{-12.39 - 8x_1 - x_3 - 3x_4}{3}$$
$$x_3 = \frac{0.25 - 2x_1 - x_2 - 4x_4}{5}$$
$$x_4 = \frac{-1.52 + x_1 + 0.5x_2 - x_3}{1}$$

Iteración 1

Teniendo en cuenta que la condición inicial es que todas las variables valen cero, y conforme se van obteniendo nuevos valores se van sustituyendo en las siguientes ecuaciones:

$$x_1(1) = \frac{0.65 - (0) - 3(0) - 2(0)}{7} = 0.09286$$

$$x_2(1) = \frac{-12.39 - 8(0.09286) - (0) - 3(0)}{3} = -4.37762$$
$$x_3(1) = \frac{0.25 - 2(0.09286) - (-4.37762) - 4(0)}{5} = 0.88838$$
$$x_4(1) = \frac{-1.52 + (0.09286) + 0.5(-4.37762) - (0.88838)}{1} = -4.50433$$

Al no cumplirse la condición de terminación del método, se sigue iterando:

Iteración 2

$$x_1(2) = \frac{0.65 - (-4.37762) - 3(0.88838) - 2(-4.50433)}{7} = 1.62445$$
$$x_2(2) = \frac{-12.39 - 8(1.62445) - (0.88838) - 3(-4.50433)}{3} = -4.25366$$
$$x_3(2) = \frac{0.25 - 2(1.62445) - (-4.25366) - 4(-4.50433)}{5} = 3.85442$$
$$x_4(2) = \frac{-1.52 + (1.62445) + 0.5(-4.25366) - (3.85442)}{1} = -5.8768$$

Comparando los valores de la segunda iteración con la primera, no se cumple la condición de terminación, por lo que se sigue iterando.

Iteración 3

$$x_1(3) = \frac{0.65 - (-4.25366) - 3(3.85442) - 2(-5.8768)}{7} = 0.72771$$
$$x_2(3) = \frac{-12.39 - 8(0.72771) - (3.85442) - 3(-5.8768)}{3} = -1.47858$$
$$x_3 = \frac{0.25 - 2(0.72771) - (-1.47858) - 4(-5.8768)}{5} = 4.75607$$
$$x_4(3) = \frac{-1.52 + (0.72771) + 0.5(-1.47858) - (4.75607)}{1} = -6.28764$$

Como sigue sin cumplirse la condición, se continúa el método.

Después de varias iteraciones:

Iteración 14	Iteración 15
$x_1 = -0.41965$	$x_1 = -0.4199$
$x_2 = 1.37926$	$x_2 = 1.37989$
$x_3 = 4.71016$	$x_3 = 4.71013$
$x_4 = -5.96018$	$x_4 = -5.9601$

En este punto como la diferencia entre variables es menor a la tolerancia dada, se detiene el método, con lo que se ha encontrado una aproximación a la solución real del sistema de ecuaciones.

EJEMPLO 2.18 ELABORAR UN PROGRAMA QUE RESUELVA EL SISTEMA DE ECUACIONES DEL EJEMPLO 2.13, UTILIZANDO EL MÉTODO DE GAUSS-SEIDEL

-Explicación de variables

x1: Arreglo unidimensional de tipo entero, utilizado para almacenar los valores de la iteración anterior.
x2: Arreglo unidimensional de tipo entero, utilizado para almacenar los valores de la iteración actual.
tolerancia: Variable de tipo real, utilizada para almacenar la tolerancia permitida para finalizar el método.
i: Variable de tipo entero, utilizada como control de ciclos, como índice del arreglo y para indicar la iteración en la que se obtiene el resultado.
MAX: Variable de tipo entero, utilizada como control de ciclos.
res: Variable de tipo entero, utilizada como control de ciclos.

-Código

```
#include<stdio.h>

float x_1(float x[]);
float x_3(float x[]);
float x_2(float x[]);

void main(){
   float x1[3],x2[3],tolerancia=0.001;
   int i,MAX=3,res=1;

   for(i=0;i<MAX;i++){
      x1[i]=x2[i]=0;
   }
   i=0;
   while(res){
      printf("\n\nIteracion %d\n",i++);
      x1[0]=x2[0];
      x2[0]=x_1(x2);
      printf("x(0)= %.2f\n",x2[0]);
      x1[1]=x2[1];
      x2[1]=x_2(x2);
      printf("x(1)= %.2f\n",x2[1]);
      x1[2]=x2[2];
      x2[2]=x_3(x2);
```

```
      printf("x(2)= %.2f\n",x2[2]);


      if(((x2[0]-x1[0])<tolerancia)&&
       ((x2[1]-x1[1])<tolerancia)&&
       ((x2[2]-x1[2])<tolerancia)){
         res=0;
      }
   }
   printf("\n\nLa solucion aproximada del sistema es:\n");
   for(i=0;i<MAX;i++){
      printf("x(%d)= %.2f\n",i,x2[i]);
   }
}

float x_1(float x[]){
   float res;
   res=23.3-2*x[1]-3*x[2];
   return res;
}

float x_2(float x[]){
   float res;
   res=(7.3-2*x[0]-3*x[2])/8;
   return res;
}

float x_3(float x[]){
   float res;
   res=(45-x[0]-x[1])/5;
   return res;
}
```

-Análisis del código

Este método, al ser iterativo, requiere de un valor de tolerancia, que en este caso está definido como 0.001, de esta forma se inicia el método, cambiando el valor de las variables uno por uno en el vector de la iteración anterior, es decir *x1* y se van obteniendo los nuevos valores mandando como parámetro el vector de los valores de la iteración actual. Imprimiendo en cada iteración los valores calculados.

```
while(res){
   printf("\n\nIteracion %d\n",i++);
   x1[0]=x2[0];
   x2[0]=x_1(x2);
   printf("x(0)= %.2f\n",x2[0]);
```

```
   x1[1]=x2[1];
   x2[1]=x_2(x2);
   printf("x(1)= %.2f\n",x2[1]);
   x1[2]=x2[2];
   x2[2]=x_3(x2);
   printf("x(2)= %.2f\n",x2[2]);

   if(((x2[0]-x1[0])<tolerancia)&&
      ((x2[1]-x1[1])<tolerancia)&&
      ((x2[2]-x1[2])<tolerancia)){
         res=0;
   }
}
```

Claro que se deben definir por separado las ecuaciones obtenidas desde el sistema para cada variable, en este caso se tienen 3 funciones definidas por cada variable del sistema.

```
float x_1(float x[]){
   float res;
   res=23.3-2*x[1]-3*x[2];
   return res;
}

float x_2(float x[]){
   float res;
   res=(7.3-2*x[0]-3*x[2])/8;
   return res;
}

float x_3(float x[]){
   float res;
   res=(45-x[0]-x[1])/5;
   return res;
}
```

2.4 Interpolación numérica

Muchas veces en ingeniería cuando se modelan sistemas, no se puede tener una ecuación exacta y correcta que pueda describir el comportamiento de un sistema, por lo que se recurre a modelar sistemas mediante estados que ya se conocen, es decir, si se trata de modelar la respuesta de un sistema eléctrico, se hacen pruebas de control, mediante las cuales a cierta entrada se observa qué salida tiene. Estos estados se pueden visualizar como puntos dentro de un sistema coordenado, por ejemplo se podría tener lo siguiente:

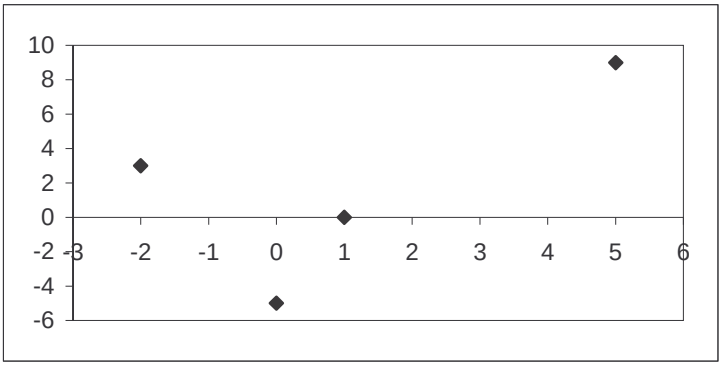


FIGURA 2.14 Ejemplo de estados de un sistema.

Claramente se puede observar, que los estados mostrados en la gráfica, no pertenecen a una ecuación en particular, debido a que en el mundo matemático pueden existir varias funciones que contengan dichos puntos. Sin embargo, utilizando los métodos numéricos para la interpolación, se puede generar una ecuación aproximada que pueda describir el sistema.

2.4.1 Diferencias Divididas de Newton

En este método, a partir de un conjunto de puntos dado, se calcula diferencias entre todos los puntos, es decir se obtienen las pendientes de las rectas que unen a todos los puntos entre sí. Todas estas diferencias generan una tabla, la cual contendrá los coeficientes a utilizar para generar un polinomio que se aproxime a una curva que describa al sistema.

Cabe hacer notar, que entre más puntos se tengan, el grado del polinomio será mayor.

La fórmula general del polinomio de diferencias divididas de Newton es:

$$f(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \ldots + b_n(x - x_0)(x - x_1)\ldots(x - x_{n-1})$$

Antes de describir la fórmula general para calcular las diferencias, es conveniente deducirla a partir de algunas ecuaciones.

Como se indicó anteriormente, entre más puntos se tengan el grado del polinomio será mayor.

Si se tiene un solo punto, el polinomio en cuestión, sólo tendría un término independiente, por lo que: $b_0 = y_0 = f(x_0)$.

Ahora, suponiendo que se tienen 2 puntos $\{(x_0, y_0), (x_1, y_1)\}$, el polinomio en cuestión sería una línea recta, es decir: $f(x) = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$, donde $b_1 = \frac{y_1 - y_0}{x_1 - x_0} = f[x_1, x_0]$

Si se tienen 3 puntos $\{(x_0, y_0), (x_1, y_1), (x_2, y_2)\}$, el grado del polinomio resultante, será 2, por lo que el polinomio sería de la forma:

$$f(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1),$$

donde los coeficientes b_0 y b_1, son los que se calcularon anteriormente. Para calcular el coeficiente b_2, se sustituye el punto (x_2,y_2), y los coeficientes b_0 y b_1, quedando:

$$y_2 = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x_2 - x_0) + b_2(x_2 - x_0)(x_2 - x_1)$$

Despejando $b_2(x_2-x_0)$:

$$b_2(x_2 - x_0) = \frac{y_2 - y_0 - \dfrac{y_1 - y_0}{x_1 - x_0}(x_2 - x_0)}{x_2 - x_1}$$

Sumando y restando y_1 en el numerador; y realizando álgebra:

$$\begin{aligned} b_2(x_2 - x_0) &= \frac{y_2 - y_1 + y_1 - y_0 - \dfrac{y_1 - y_0}{x_1 - x_0}(x_2 - x_0)}{x_2 - x_1} \\ &= \frac{y_2 - y_1}{x_2 - x_1} + \frac{y_1 - y_0}{x_2 - x_1} - \frac{y_1 - y_0}{x_2 - x_1}\left(\frac{x_2 - x_0}{x_1 - x_0}\right) \\ &= \frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_2 - x_1}\left(\frac{x_2 - x_0}{x_1 - x_0} - 1\right) \\ &= \frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_2 - x_1}\left(\frac{x_2 - x_0 - (x_1 - x_0)}{x_1 - x_0}\right) \\ &= \frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_2 - x_1}\left(\frac{x_2 - x_1}{x_1 - x_0}\right) \end{aligned}$$

Por último se tiene:

$$b_2(x_2 - x_0) = \frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}$$

Despejando b_2:

$$b_2 = \frac{\dfrac{y_2 - y_1}{x_2 - x_1} - \dfrac{y_1 - y_0}{x_1 - x_0}}{(x_2 - x_0)} = \frac{f[x_2, x_1] - f[x_1, x_0]}{x_2 - x_0} = f[x_2, x_1, x_0]$$

EJEMPLO 2.19 OBTENER EL POLINOMIO DE DIFERENCIAS DIVIDIDAS DE NEWTON, DE LOS PUNTOS DADOS A CONTINUACIÓN,

x	y
-3	5.941
-1	0.368
0.1	-11.502
1.2	5.693
2.1	39.722

Solución

De los puntos anteriores, se calculan las diferencias:

a) Primeras diferencias

x	y	Primer Diferencia
-3	5.941	$f[x_1,x_0]=\frac{f(x_1)-f(x_0)}{x_1-x_0}=\frac{0.368-5.941}{-1-(-3)}=-2.7865$
-1	0.368	$f[x_2,x_1]=\frac{f(x_2)-f(x_1)}{x_2-x_1}=\frac{-11.502-0.368}{0.1-(-1)}=-10.791$
0.1	-11.502	$f[x_3,x_2]=\frac{f(x_3)-f(x_2)}{x_3-x_2}=\frac{5.693-(-11.502)}{1.2-0.1}=15.632$
1.2	5.693	$f[x_4,x_3]=\frac{f(x_4)-f(x_3)}{x_4-x_3}=\frac{39.722-5.693}{2.1-1.2}=37.81$
2.1	39.722	

b) Segundas diferencias

x	y	1° Dif	Segunda Diferencia
-3	5.941	-2.7865	$f[x_2,x_1,x_0]=\frac{f[x_2,x_1]-f[x_1,x_0]}{x_2-x_0}=\frac{-10.791-(-2.7865)}{0.1-(-3)}=-2.5821$
-1	0.368	-10.791	$f[x_3,x_2,x_1]=\frac{f[x_3,x_2]-f[x_2,x_1]}{x_3-x_1}=\frac{15.632-(-10.791)}{1.2-(-1)}=12.01045$
0.1	-11.502	15.632	$f[x_4,x_3,x_2]=\frac{f[x_4,x_3]-f[x_3,x_2]}{x_4-x_2}=\frac{37.81-15.632}{2.1-0.1}=11.089$

1.2	5.693	37.81
2.1	39.722	

c) Terminando la tabla

x	y	1° Dif	2° Dif	3° Dif	4° Dif
-3	b_0 = 5.941	b_1 = -2.7865	b_2 = -2.5821	b_3 = 3.4744	b_4 = -0.7395
-1	0.368	-10.791	12.01045	-0.2972	
0.1	-11.502	15.632	11.089		
1.2	5.693	37.81			
2.1	39.722				

Para obtener el polinomio de interpolación, se sustituyen los coeficientes *b*, en la ecuación general, quedando:

$$f(x) = 5.941 - 2.7865(x+3) - 2.5821(x+3)(x+1) + 3.4744(x+3)(x+1)(x-0.1) - 0.7395(x+3)(x+1)(x-0.1)(x-1.2)$$

Haciendo álgebra y simplificando, queda la siguiente expresión:

$$f(x) = -0.7395x^4 + 1.47775x^3 + 12.50622x^2 - 1.55237x - 11.47334$$

Obteniendo la gráfica:

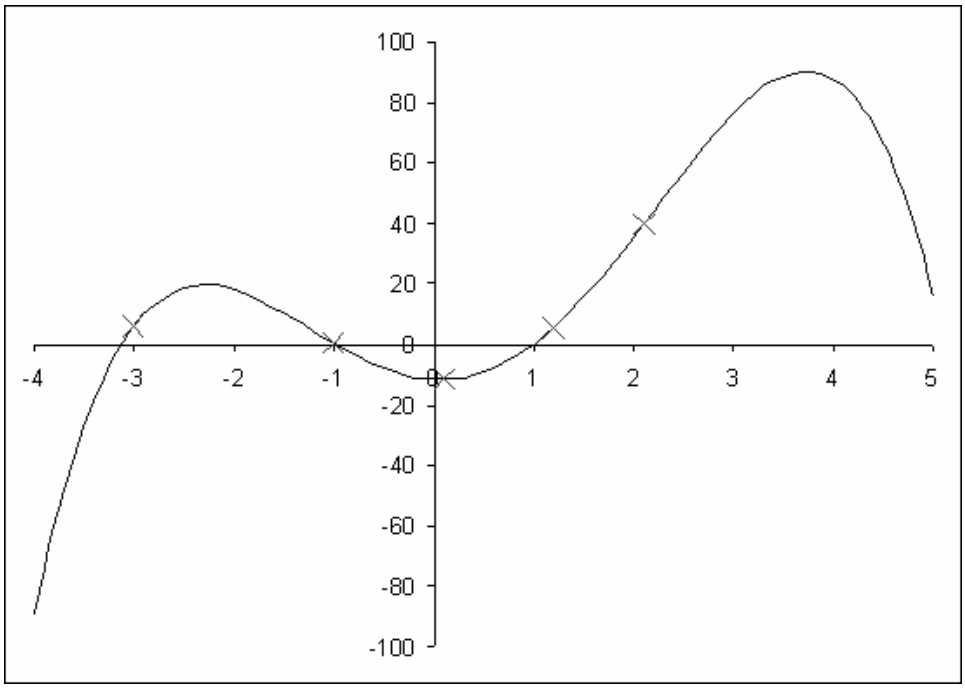


FIGURA 2.15 Gráfica del polinomio obtenido por el método de Newton del ejemplo 2.19.

Se puede observar que los puntos originales están contenidos en la curva obtenida con el método.

EJEMPLO 2.20 ELABORAR UN PROGRAMA QUE RESUELVA LA ECUACIÓN DEL EJEMPLO 2.19

-Explicación de variables

x: Arreglo unidimensional de tipo real, utilizado para almacenar la componente en X de los pares de puntos.
dif: Arreglo bidimensional de tipo de real, utilizado para almacenar en la primer columna la componente en Y de los pares de puntos, y las diferencias en las siguientes columnas.
i: Variable de tipo entero, utilizado como control de ciclos y como índice del arreglo.
j: Variable de tipo entero, utilizado como control de ciclos y como índice del arreglo.

-Código

```
#include<stdio.h>

void main(){
   float x[5],dif[5][5];
   int i,j;

   for(i=0;i<5;i++){
      printf("Dame el valor de x(%d): ",i);
      scanf("%g",&x[i]);
      printf("Dame el valor de y(%d): ",i);
      scanf("%g",&dif[i][0]);
   }

   for(i=1;i<5;i++){
      for(j=0;j<=(4-i);j++){
         dif[j][i]=(dif[j+1][i-1]-dif[j][i-1])/(x[j+i]-x[j]);
      }
   }

   printf("\n\n");

   for(i=0;i<5;i++){
      for(j=0;j<(5-i);j++){
         printf("%.2f\t",dif[i][j]);
      }
      printf("\n");
   }
}
```

-Análisis del código

El programa inicia solicitando los pares de puntos

```
for(i=0;i<5;i++){
   printf("Dame el valor de x(%d): ",i);
   scanf("%g",&x[i]);
   printf("Dame el valor de y(%d): ",i);
   scanf("%g",&dif[i][0]);
}
```

A continuación se obtienen las diferencias, almacenándolas dentro de una matriz

```
for(i=1;i<5;i++){
   for(j=0;j<=(4-i);j++){
      dif[j][i]=(dif[j+1][i-1]-dif[j][i-1])/(x[j+i]-x[j]);
   }
}
```

Una vez calculadas las diferencias se manda a imprimir en pantalla la tabla de diferencias, donde en el primer renglón se tienen los coeficientes a sustituir en la ecuación general.

```
for(i=0;i<5;i++){
   for(j=0;j<(5-i);j++){
      printf("%.2f\t",dif[i][j]);
   }
   printf("\n");
}
```

2.4.2 Interpolación de Lagrange

Además del método de Diferencias Divididas de Newton para interpolación, existe otro método el cual se basa de igual forma, en un conjunto de puntos dado, del que se obtiene un polinomio que pasa por los puntos dados.

La forma general del polinomio de Lagrange es:

$$f(x) = y_0 L_0 + y_1 L_1 + \ldots + y_n L_n = \sum_{i=0}^{n} y_i L_i$$

Donde:

$$L_i = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{(x - x_j)}{(x_i - x_j)}$$

Es decir, si se tienen dos puntos, el polinomio de interpolación, sería:

$$f(x) = y_0 \frac{(x - x_1)}{(x_0 - x_1)} + y_1 \frac{(x - x_0)}{(x_1 - x_0)}$$

Si se tuvieran tres puntos:

$$f(x) = y_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + y_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + y_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

Y así sucesivamente.

Cabe destacar que si se tienen *n* puntos, el polinomio que se genera es de grado *n-1*.

EJEMPLO 2.21 OBTENER EL POLINOMIO DE LAGRANGE, A PARTIR DE LOS PUNTOS DADOS A CONTINUACIÓN.

x	y
-3	5.941
-1	0.368
0.1	-11.502
1.2	5.693
2.1	39.722

Solución

Primero se obtiene la forma general del polinomio, teniendo en cuenta que son 5 puntos.

$$\begin{aligned} f(x) = {} & y_0 \frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)(x_0-x_4)} + y_1 \frac{(x-x_0)(x-x_2)(x-x_3)(x-x_4)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)(x_1-x_4)} \\ & + y_2 \frac{(x-x_0)(x-x_1)(x-x_3)(x-x_4)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)(x_2-x_4)} + y_3 \frac{(x-x_0)(x-x_1)(x-x_2)(x-x_4)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)(x_3-x_4)} \\ & + y_4 \frac{(x-x_0)(x-x_1)(x-x_2)(x-x_3)}{(x_4-x_0)(x_4-x_1)(x_4-x_2)(x_4-x_3)} \end{aligned}$$

Ahora se sustituyen los puntos en la ecuación, quedando:

$$\begin{aligned} f(x) = {} & 5.941 \frac{(x+1)(x-0.1)(x-1.2)(x-2.1)}{(-3+1)(-3-0.1)(-3-1.2)(-3-2.1)} + 0.368 \frac{(x+3)(x-0.1)(x-1.2)(x-2.1)}{(-1+3)(-1-0.1)(-1-1.2)(-1-2.1)} \\ & - 11.502 \frac{(x+3)(x+1)(x-1.2)(x-2.1)}{(0.1+3)(0.1+1)(0.1-1.2)(0.1-2.1)} + 5.693 \frac{(x+3)(x+1)(x-0.1)(x-2.1)}{(1.2+3)(1.2+1)(1.2-0.1)(1.2-2.1)} \\ & + 39.722 \frac{(x+3)(x+1)(x-0.1)(x-1.2)}{(2.1+3)(2.1+1)(2.1-0.1)(2.1-1.2)} \end{aligned}$$

Calculando las constantes del denominador:

$$f(x) = 5.941\frac{(x+1)(x-0.1)(x-1.2)(x-2.1)}{132.804} + 0.368\frac{(x+3)(x-0.1)(x-1.2)(x-2.1)}{-15.004}$$
$$-11.502\frac{(x+3)(x+1)(x-1.2)(x-2.1)}{7.502} + 5.693\frac{(x+3)(x+1)(x-0.1)(x-2.1)}{-9.1476}$$
$$+39.722\frac{(x+3)(x+1)(x-0.1)(x-1.2)}{28.458}$$

Dividiendo las constantes:

$$f(x) = 0.045(x+1)(x-0.1)(x-1.2)(x-2.1) - 0.0245(x+3)(x-0.1)(x-1.2)(x-2.1)$$
$$-1.5332(x+3)(x+1)(x-1.2)(x-2.1) - 0.6223(x+3)(x+1)(x-0.1)(x-2.1)$$
$$+1.3958(x+3)(x+1)(x-0.1)(x-1.2)$$

Multiplicando los binomios y haciendo una simplificación, el polinomio resultante es:

$$f(x) = -0.7395x^4 + 1.4777x^3 + 12.5062x^2 - 1.5522x - 11.4732$$

Graficando esta función, se tiene:

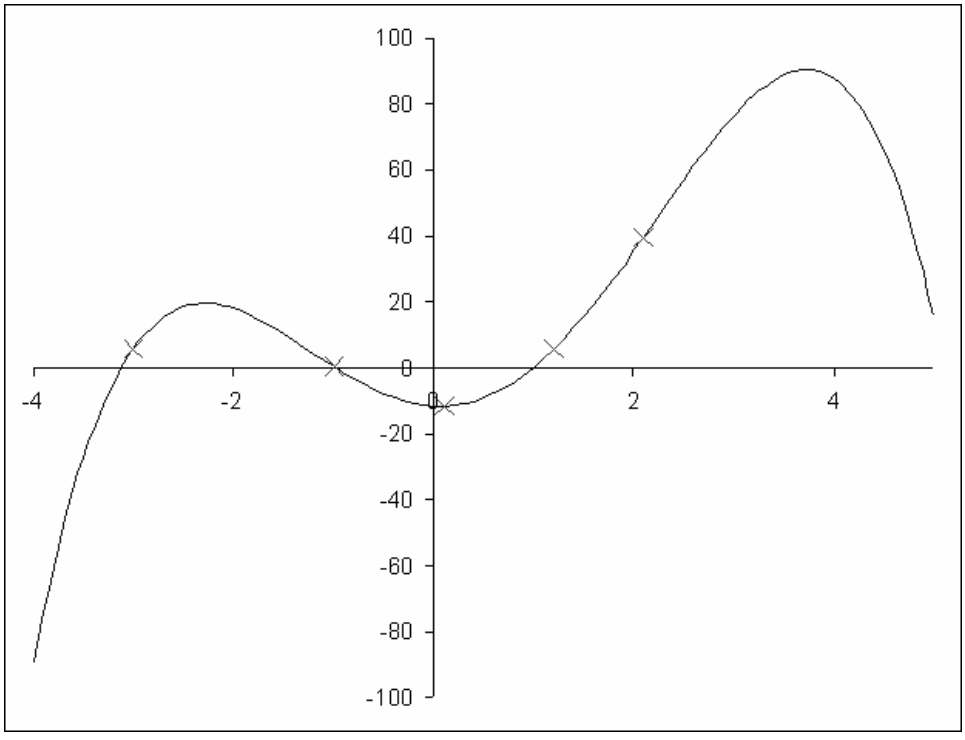


FIGURA 2.15 Gráfica del polinomio obtenido por el método de Lagrange del ejemplo 2.21.

EJEMPLO 2.22 ELABORAR UN PROGRAMA QUE OBTENGA EL POLINOMIO DE LAGRANGE, DEL CONJUNTO DE PUNTOS DEL EJEMPLO 2.21

-Explicación de variables

x: Arreglo unidimensional de tipo real, utilizado para almacenar las *x*'s de los pares de puntos.
y: Arreglo unidimensional de tipo real, utilizado para almacenar las *y*'s de los pares de puntos.
div_x: Arreglo unidimensional de tipo real, utilizado para almacenar las constantes del denominador, generadas por el método de Lagrange.
cte: Arreglo unidimensional de tipo real, utilizado para almacenar la división entre las *y*'s y las constantes del denominador.
i: Variable de tipo entero, utilizada como control de ciclos y como índice del arreglo.
j: Variable de tipo entero, utilizada como control de ciclos y como índice del arreglo.

-Código

```
#include<stdio.h>

void main(){
   float x[5],y[5],div_x[5],cte[5];
   int i,j;

   x[0]=-3;
   x[1]=-1;
   x[2]=0.1;
   x[3]=1.2;
   x[4]=2.1;

   y[0]=5.941;
   y[1]=0.368;
   y[2]=-11.502;
   y[3]=5.693;
   y[4]=39.722;

   for(i=0;i<5;i++){
      div_x[i]=1;
   }

   for(i=0;i<5;i++){
      for(j=0;j<5;j++){
         if(i!=j){
            div_x[i]=div_x[i]*(x[i]-x[j]);
         }
      }
   }
```

```
   for(i=0;i<5;i++){
      printf("divisor %d: %.4f\n",i,div_x[i]);
   }

   for(i=0;i<5;i++){
      cte[i]=y[i]/div_x[i];
   }

   printf("\nLa ecuacion general queda como:\n\n");

   for(i=0;i<5;i++){
      if(cte[i]>0){
         printf("+%.3f",cte[i]);
      }
      else{
         printf("%.3f",cte[i]);
      }
      for(j=0;j<5;j++){
         if(i!=j){
            if(x[j]<0){
               printf("(x+%.1f)",(0-x[j]));
            }
            else{
               printf("(x%.1f)",(0-x[j]));
            }
         }
      }
      printf("\n");
   }
}
```

-Análisis del código

En el código, debido a que se pretende resolver el ejemplo 2.21, se tienen líneas de código con los valores de *x* y *y* de los pares de puntos, y a continuación se inicializa el vector ***div_x*** en uno para realizar la función:

$$L_i = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{(x - x_j)}{(x_i - x_j)}$$

Pero sólo se multiplicarán los términos $(x_i - x_j)$, por lo que el código que realiza dicha multiplicación es:

```
for(i=0;i<5;i++){
   for(j=0;j<5;j++){
      if(i!=j){
        div_x[i]=div_x[i]*(x[i]-x[j]);
      }
   }
}
```

Luego se mandan imprimir las constantes obtenidas con el método:

```
for(i=0;i<5;i++){
   printf("divisor %d: %.4f\n",i,div_x[i]);
}
```

Y se tiene un ciclo de impresión de la ecuación general con las constantes:

```
printf("\nLa ecuacion general queda como:\n\n");

for(i=0;i<5;i++){
   if(cte[i]>0){
      printf("+%.3f",cte[i]);
   }
   else{
      printf("%.3f",cte[i]);
   }
   for(j=0;j<5;j++){
      if(i!=j){
         if(x[j]<0){
            printf("(x+%.1f)",(0-x[j]));
         }
         else{
            printf("(x%.1f)",(0-x[j]));
         }
      }
   }
   printf("\n");
}
```

En conclusión, comparando los métodos vistos de interpolación, se puede observar que en ambos casos la función pasa por los puntos dados. Aunque la forma en que se obtienen estas aproximaciones puede llegar a ser complicado, ambos métodos son igualmente programables.

Ejercicios Propuestos

1) Elaborar un programa que utilice el método de bisección para obtener una de las raíces de la siguiente función: $f(x)=\ln(x)\cos(e^x)+x^3$. Considerar un intervalo inicial de [0.01,1] y una tolerancia de 0.00001.

2) Elaborar un programa que utilice el método de bisección para obtener una de las raíces de la siguiente función: $f(x)=x^5-69.472x^4+559.3038x^3+4057.4x^2-24715.94x-69.6075$ Considerar un intervalo inicial de [2,5] y una tolerancia de 0.00001.

3) Elaborar un programa que utilice el método de bisección para obtener una de las raíces de la siguiente función: $f(x)=x^4-16.50887x^2+27.7777$ Considerar un intervalo inicial de [1,2] y una tolerancia de 0.00001.

4) Elaborar un programa que utilice el método de bisección para obtener la raíz de la siguiente función: $f(x)=\frac{\cos(x+8)}{e^x}+30$. Considerar como intervalo inicial [-4,-3.5] y una tolerancia de 0.00001.

5) Elaborar un programa que utilice el método de bisección para obtener la raíz de la siguiente función: $f(x)=xe^x+\frac{\ln(x)}{x}$. Considerar como intervalo inicial [0.1,0.5] y una tolerancia de 0.00001.

6) Elaborar un programa que utilice el método de bisección para obtener la raíz de la siguiente función: $f(x)=\frac{\sin(e^{\cos(x)})}{x}-0.5$. Considerar como intervalo inicial [1,2] y una tolerancia de 0.00001.

7) Elaborar un programa que utilice el método de Newton-Raphson para obtener una de las raíces de la siguiente función: $f(x)=\ln(x)\cos(e^x)+x^3$. Considerar como valor inicial $x_0=1$ y una tolerancia de 0.00001.

8) Elaborar un programa que utilice el método de Newton-Raphson para obtener una de las raíces de la siguiente función: $f(x)=x^5-69.472x^4+559.3038x^3+4057.4x^2-24715.94x-69.6075$. Considerar como valor inicial $x_0=2$ y una tolerancia de 0.00001.

9) Elaborar un programa que utilice el método de Newton-Raphson para obtener una de las raíces de la siguiente función: $f(x)=x^4-16.50887x^2+27.7777$. Considerar como valor inicial $x_0=2$ y una tolerancia de 0.00001.

10) Elaborar un programa que utilice el método de Newton-Raphson para obtener la raíz de la siguiente función: $f(x)=\frac{\cos(x+8)}{e^x}+30$. Considerar como valor inicial $x_0=-4$ y una tolerancia de 0.00001.

11) Elaborar un programa que utilice el método de Newton-Raphson para obtener la raíz de la siguiente función: $f(x)=xe^x+\frac{\ln(x)}{x}$. Considerar como valor inicial $x_0=1$ y una tolerancia de 0.00001.

12) Elaborar un programa que utilice el método de Newton-Raphson para obtener la raíz de la siguiente función: $f(x)=\frac{\sin(e^{\cos(x)})}{x}-0.5$ Considerar como valor inicial $x_0=1$ y una tolerancia de 0.00001.

13) Elaborar un programa que utilice el método de Eliminación Gaussiana, para obtener la solución del siguiente sistema de ecuaciones:

$$\begin{bmatrix} 1 & 1 & 0 & 3 \\ 2 & 5 & 8 & 11 \\ 3 & 0 & 1 & 1 \\ 4 & 8 & 4 & 8 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}=\begin{bmatrix} 5.2829 \\ 37.4273 \\ 4.3001 \\ 32.8608 \end{bmatrix}$$

14) Elaborar un programa que utilice el método de Eliminación Gaussiana, para obtener la solución del siguiente sistema de ecuaciones:

$$\begin{bmatrix} 6 & 2 & 1 & 5 \\ -2 & -1 & 0 & -1 \\ 4 & -3 & 1 & 5 \\ 1 & 3 & 2 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}=\begin{bmatrix} -3.575 \\ -0.9624 \\ -3.8215 \\ -3.4662 \end{bmatrix}$$

15) Modifique el programa para Eliminación Gaussiana, de tal forma que resuelva la restricción de un cero en el renglón pivote.

16) Utilizando el programa anterior, obtener la solución del siguiente sistema de ecuaciones:

$$\begin{bmatrix} 3 & 6 & 9 & 12 \\ 2 & 6 & 4 & 4 \\ 4 & 9 & 11 & 20 \\ 1 & 5 & 6 & 5 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}=\begin{bmatrix} -35.8077 \\ 9.3682 \\ -65.6738 \\ -13.3597 \end{bmatrix}$$

17) Utilizando el programa del inciso 15, resolver el siguiente sistema de ecuaciones:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 3 & 2 & 4 & 5 \\ 6 & 7 & 5 & 9 \\ 4 & 8 & 12 & 15 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 4.0778 \\ 12.4989 \\ 30.2013 \\ -0.2788 \end{bmatrix}$$

18) Elaborar un programa que resuelva el sistema del inciso 13, utilizando el método de Gauss-Jordan.

19) Elaborar un programa que resuelva el sistema del inciso 14, utilizando el método de Gauss-Jordan.

20) Modifique el programa para Gauss-Jordan, de tal forma que resuelva la restricción de un cero en el renglón pivote.

21) Utilizando el programa anterior, resolver el sistema del inciso 16.

22) Utilizando el programa del inciso 20, resolver el sistema del inciso 17.

23) Elaborar un programa que resuelva el sistema del inciso 13, utilizando el método de LU.

24) Elaborar un programa que resuelva el sistema del inciso 14, utilizando el método de LU.

25) Modificando los renglones del sistema del inciso 16, resolver el sistema utilizando el método de LU.

26) Modificando los renglones del sistema del inciso 17, resolver el sistema utilizando el método de LU.

27) Elaborar un programa que resuelva el sistema del inciso 13, utilizando el método de Gauss-Seidel.

28) Elaborar un programa que resuelva el sistema del inciso 14, utilizando el método de Gauss-Seidel.

29) Elaborar un programa que resuelva el sistema del inciso 16, utilizando el método de Gauss-Seidel.

30) Elaborar un programa que resuelva el sistema del inciso 17, utilizando el método de Gauss-Seidel.

31) Utilizando un programa, obtener los coeficientes del polinomio de diferencias divididas de Newton, para los siguientes puntos:

x	$f(x)$
4	-3.7840
8	4.9468
13	2.1
18	-3.7549
22	-0.04425

luego obtener el polinomio resultante.

32) Utilizando un programa, obtenner los coeficientes del polinomio de diferencias divididas de Newton, para los siguientes puntos:

x	$f(x)$
6.89	1.5739
12.0818	-4.1671
17.5	-4.9219
21.997	-0.4383
25.6	3.3183

luego obtener el polinomio resultante.

33) Utilizando un programa, obtener los coeficientes del polinomio de diferencias divididas de Newton, para los siguientes puntos:

x	$f(x)$
1.11	0.6526
6.2198	0.9878
14.093	0.9999
15.94	0.9999
16.366	1

luego obtener el polinomio resultante.

34) Utilizando un programa, obtener los coeficientes del polinomio de Lagrange, para los siguientes puntos:

x	$f(x)$
4	-3.7840
8	4.9468
13	2.1
18	-3.7549
22	-0.04425

luego obtener el polinomio resultante.

35) Utilizando un programa, obtener los coeficientes del polinomio de Lagrange, para los siguientes puntos:

x	$f(x)$
6.89	1.5739
12.0818	-4.1671
17.5	-4.9219
21.997	-0.4383
25.6	3.3183

luego obtener el polinomio resultante.

36) Utilizando un programa, obtener los coeficientes del polinomio de Lagrange, para los siguientes puntos:

x	$f(x)$
1.11	0.6526
6.2198	0.9878
14.093	0.9999
15.94	0.9999
16.366	1

luego obtener el polinomio resultante.

CAPÍTULO 3

FUNDAMENTOS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

Introducción

Los lenguajes computacionales han experimentado una impresionante evolución desde que se construyeron las primeras computadoras. Éstos han permitido escribir programas como una serie de procedimientos que actúan sobre los datos, donde los datos están separados de los procedimientos.

La técnica de la programación estructurada se define en los setenta y fue uno de los métodos más utilizados en el campo de la programación. Uno de los principales conceptos que introduce esta programación es la abstracción, que permite concentrarnos en conocer qué tarea realiza un procedimiento, sin preocuparnos en cómo esta tarea es realizada. Sin embargo a medida que los programas crecen las estructuras de datos sobre las cuales operan los procedimientos cobran mayor importancia.

Los tipos de datos son procesados en muchas funciones dentro de un programa estructurado y cuando se producen cambios en estos tipos de datos, deben hacerse modificaciones en cada sentencia que actúa sobre estos tipos dentro del programa. Ahora dado que muchas funciones acceden a datos compartidos, la disposición de los datos no se puede cambiar sin modificar todas las funciones que los utilizan. En conclusión con la programación estructurada podemos empaquetar código en funciones.

La Programación Orientada a Objetos trata a los datos y a los procedimientos que operan sobre ellos como un solo objeto. Esta programación surge como un paradigma que permite acoplar el diseño de programas a situaciones del mundo real, las entidades centrales son los objetos, que son tipos de datos que encapsulan con el mismo nombre estructuras de datos y las operaciones o algoritmos que manipulan esos datos.

3.1 Paradigmas de programación: imperativa, funcional, lógica, declarativa y orientada a objetos

La historia del desarrollo de los lenguajes de programación ha mostrado hasta el momento una creciente evolución en la que se han ido incorporando elementos que permiten ir creando programas cada vez más sólidos y eficientes y que a la vez permite facilitar la tarea del programador para su desarrollo, mantenimiento y adaptación.

Algunas características básicas incorporadas por algunos lenguajes han sido criticadas y/o analizadas por otros, por esta razón se han perfilando diferentes teorías y grupos de lenguajes que postulaban diversas formas de construir soluciones. A medida que fue avanzando el tiempo, surgió lo que actualmente se definen como “paradigmas”.

Un paradigma de programación es un modelo básico de diseño e implementación de programas. Un modelo que permite desarrollar programas conforme a ciertos principios o fundamentos específicos que se aceptan como válidos. En otras palabras, es una colección de modelos conceptuales que juntos conforman el proceso de diseño, orientan la forma de pensar y solucionar los problemas y, por lo tanto, determinan la estructura final de un programa.

A los paradigmas se les podría clasificar de diversas maneras según los criterios que se prioricen. Pero partiendo de los principios fundamentales de cada paradigma en cuanto a las orientaciones sobre la forma para construir las soluciones, podemos distinguir entre los *de procedimiento* y los *declarativos*, más específicamente, de éstos se derivan otros paradigmas, es decir, los modelos más básicos, que son: el *imperativo, aplicativo, lenguaje con base en reglas y orientado a objetos.*

- Paradigmas de procedimiento u operacionales. Indican el modo de construir la solución, es decir detallan paso a paso el mecanismo para obtenerla.

- Paradigmas declarativos. Describen las características que debe tener la solución. Es decir especifican “qué” se desea obtener pero no requieren indicar “cómo” obtenerla.

Por otra parte, existen otro tipo de clasificaciones de paradigmas con base en criterios diferentes, por ejemplo como de “alto nivel” o “bajo nivel”.

3.1.1 Paradigmas de procedimiento

Estos paradigmas también son llamados *operacionales*, la característica fundamental de estos paradigmas es la secuencia computacional realizada etapa a etapa para resolver el problema.

Los programas realizados con lenguajes de procedimiento deben incluir en su codificación las instrucciones de control para determinar el flujo de la ejecución, como decisiones, iteraciones y otras, conformando, de esta manera, diferentes “algoritmos”.

Actúan modificando repetidamente la representación de sus datos. Utilizan un modelo en el que las variables están estrechamente relacionadas con direcciones de la memoria del ordenador. Cuando se ejecuta el programa, el contenido de estas direcciones se actualiza repetidamente, pues las variables reciben múltiples asignaciones, y al finalizar el trabajo, los valores finales de las variables representan el resultado.

Su mayor dificultad reside en determinar si el valor calculado es una solución correcta del problema, por lo que se han desarrollado multitud de técnicas de depuración y verificación para probar la corrección de los problemas desarrollados basándose en este tipo de paradigmas.

En otras palabras, se basan en “cómo” lograr la solución.

Esta clasificación engloba a los siguientes paradigmas:

- Paradigma Imperativo
- Paradigma Orientado a Objetos

3.1.1.1 Lenguajes imperativos o de procedimiento

Estos lenguajes son controlados mediante mandatos o son orientados a enunciados, es decir, instrucciones. Sabemos que un programa está compuesto por varios enunciados y la ejecución de cada enunciado hace que se cambie el valor de una o varias localidades de memoria. La sintaxis general de este modelo de lenguajes es:

```
enunciado1;
enunciado2;
…
```

Por ejemplo, suponiendo que se tienen que sumar un par de variables para obtener una tercera, se puede representar como acceder a 2 localidades de memoria, combinar estos valores y guardar el resultado en una nueva localidad.

Un programa imperativo, normalmente realiza su tarea ejecutando repetidamente una secuencia de pasos elementales, ya que en este modelo la única forma de ejecutar algo complejo es repitiendo una secuencia de instrucciones.

La programación requiere la construcción de "algoritmos", a modo de receta, método, técnica, procedimiento o rutina, que se definen como conjuntos finitos de sentencias, ordenadas de acuerdo a sus correspondientes estructuras de control, que marcan el flujo de ejecución de operaciones para resolver un problema específico.

Muchos lenguajes de uso amplio manejan este modelo, tal es el caso de los siguientes: C, Fortran, Algol, PL/I, Pascal, Ada, Cobol. Este modelo procede del hardware de la computadora convencional que ejecuta instrucciones en forma secuencial.

EJEMPLO 3.1 ELABORAR UN PROGRAMA QUE SUME 2 NÚMEROS UTILIZANDO EL PARADIGMA IMPERATIVO O PROCEDIMIENTO.

-Desarrollo

Seleccionando el lenguaje Pascal para la elaboración del programa, el código se presenta a continuación; puede observarse que está constituido por sólo instrucciones.

```
(* Programa para sumar dos numeros *)
programa sumar_dos_numeros;
{ Esta funcion devuelve la suma de los enteros a y b }
functionsumar(a, b : integer) : integer;
begin
     sumar := a + b
end;

{ Linea principal del programa }
var
   a, b : integer;

begin
     write('Introduce dos numeros enteros y pulsa enter: ');
     readln(a,b);
     writeln('La suma de ', a, ' y ', b, ' es ', sumar(a,b))
end.
```

3.1.1.2 Orientada a objetos.

El paradigma de objetos surge con base en el paradigma imperativo, provocando un giro importante en sus principios y consideraciones básicas, en el que más que dejarlos de lado, los reorganiza y capitaliza dentro de un modelo más amplio, con otros conceptos característicos.

La programación orientada a objetos se convierte cada día en un lenguaje de suma importancia. Este lenguaje trabaja de la siguiente manera: se tienen objetos complejos de datos y posteriormente se designa un conjunto limitado de funciones para que operen con esos datos. Llamamos objetos complejos a aquellos que se designan de objetos más simples y que además heredan propiedades del objeto más sencillo.

Al construir objetos concretos de datos, un programa orientado a objetos gana la eficiencia de los lenguajes imperativos, y al construir clases de funciones que utilizan un conjunto limitado de objetos y datos, se construye la flexibilidad y confiabilidad del modelo aplicativo.

El lenguaje originario y paradigmático de la programación en objetos es Smalltalk. Actualmente existen otros lenguajes más conocidos, como Java. Hay también algunos lenguajes como el C++ o el Eiffel, Visual Basic que son extensiones de otros lenguajes que fueron diseñados básicamente como imperativos, pero que tienen extensiones que incorporan, en mayor o menor medida, los principios del paradigma de objetos.

Otros lenguajes que soportan este paradigma:

- Simula
- Self (Objetos sin clases)
- Python
- C#

EJEMPLO 3.2 ELABORAR UN PROGRAMA QUE SUME 2 NÚMEROS UTILIZANDO EL PARADIGMA ORIENTADO A OBJETOS

-Desarrollo

Seleccionando el lenguaje Java para la elaboración del programa, el código se presenta a continuación; puede observarse que se define una *clase* (Suma) de la cual se pueden derivar *objetos.*

```
public class Suma
{
     public static void main(String args[])
     {
          int num=Integer.parseInt(args[0]);
          int num2=Integer.parseInt(args[1]);
          int sum=num+num2;
          System.out.println("El resultado es " +sum);
     }
}
```

3.1.2 Paradigmas declarativos

Los paradigmas declarativos están basados en desarrollar programas especificando o "declarando" un conjunto de proposiciones, condiciones, restricciones, afirmaciones, ecuaciones o transformaciones que caracterizan al problema y describen su solución.

Con estas características, el sistema utiliza mecanismos internos de control que evalúan y relacionan adecuadamente dichas especificaciones, para obtener la solución. No se necesita

de la puntualización de los pasos a seguir para alcanzar una solución, ni instrucciones de control que conformen algoritmos.

Estos paradigmas permiten manejar variables para almacenar valores intermedios, pero no para actualizar estados de información. Sus variables se relacionan con posiciones de memoria. Las variables son usadas en expresiones, funciones o procedimientos, se unifican con diferentes valores.

Estos paradigmas especifican la solución sin indicar cómo construirla, ya que eliminan la necesidad de probar que el valor calculado es el valor solución. En otras palabras, se basan en "qué" es necesario especificar. Y dentro esta clasificación se engloban los siguientes paradigmas:

- Paradigma Funcional
- Paradigma Lógico

3.1.2.1 Lenguajes aplicativos o funcionales

En este modelo, el resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que una composición produce el resultado deseado. Así, un programa es un conjunto de funciones que cooperan entre ellas para el logro de un objetivo común.

Este tipo de lenguajes de programación lo que hace es examinar la función que el programa está representando, y no va revisando enunciado por enunciado. Lo que busca este tipo de lenguajes es encontrar la función que encuentre la respuesta mediante la combinación de varias variables específicas. Los lenguajes que hacen énfasis en este tipo de acción, se les llama *lenguajes aplicativos* o *funcionales.*

La sintaxis que se maneja generalmente en este lenguaje de programación es similar a:

$\texttt{funcion}_n(\ldots\texttt{funcion}_2(\texttt{funcion}_1(\texttt{datos}))\ldots)$

Ejemplos de este tipo de lenguajes de programación son: LISP y ML.

EJEMPLO 3.3 ELABORAR UN PROGRAMA QUE SUME 2 NÚMEROS UTILIZANDO EL PARADIGMA APLICTIVO O FUNCIONAL

- Desarrollo

Seleccionando el lenguaje LISP para la elaboración del programa, el código se presenta a continuación:

```
(+ 5 1)
```

- Explicación del código

Para evaluarla LISP procede de la siguiente manera:

a. Lee la expresión completa (+ 5 1)
b. La interpreta como una llamada a una función y la identifica como SUMAR <+>
c. Interpreta 5 como primer argumento y 1 como segundo. El paréntesis de cierre le indica que no hay más argumentos.
d. La función <+> se evalúa para 5 y 1, devolviendo 6 como resultado, que a falta de otro destino es impreso en pantalla.

Programar en LISP significa llamar a funciones.

3.1.2.2 Lenguajes con base en reglas

El paradigma *lógico* o *basado en reglas* se caracteriza principalmente por la aplicación de las reglas de la lógica para inferir conclusiones a partir de datos. El paradigma lógico, que resultó una apasionante novedad en la década de los 70, tiene como característica diferenciadora el hecho de manejarse de manera declarativa y con la aplicación de las reglas de la lógica. Esto significa que en lugar de basarse, como en el caso de los paradigmas procedurales, en el planteo del algoritmo para la resolución del problema, se basa en expresar todas las condiciones del problema y luego buscar un objetivo dentro de las declaraciones realizadas.

Si tenemos conocimiento de la información y las condiciones del problema, la ejecución de un programa consiste en la búsqueda de un objetivo dentro de las declaraciones realizadas. Esta forma de tratamiento de la información permite pensar la existencia de “programas inteligentes” que puedan responder, no por tener en la base de datos todos los conocimientos, sino por poder inferirlos a través de la deducción.

Un programa lógico contiene una base de conocimiento sobre la que se hacen consultas. La base de conocimiento está formada por hechos, que representan la información del sistema expresada como relaciones entre datos, y reglas lógicas que permiten deducir consecuencias a partir de combinaciones entre los hechos y, en general, otras reglas. De esta manera, una clave de la programación lógica es poder expresar apropiadamente todos los hechos y reglas necesarios que definen el dominio de un problema.

Los lenguajes con base en reglas se ejecutan verificando la presencia de una cierta condición y, cuando se satisface, ejecutan una acción apropiada.

El lenguaje más común con base en reglas es Prolog, que es conocido también como el *lenguaje de programación lógico*, ya que las condiciones básicas son ciertas clases de expresiones lógicas de predicados.

Cuando nos referimos a lenguajes basados en reglas estamos hablando de que existe un conjunto de filtros que se aplican al almacenamiento de datos.

La ejecución de un lenguaje con base en reglas es muy parecido a la del lenguaje imperativo, sólo que los enunciados no son secuenciales. Las condiciones que se tienen, determinan el orden de ejecución.

La sintaxis de este tipo de lenguajes es dada de la siguiente manera.

$$\begin{aligned} &condición_1 \rightarrow acción_1 \\ &condición_2 \rightarrow acción_2 \\ &\vdots \\ &condición_n \rightarrow acción_n \end{aligned}$$

En ocasiones suele suceder que la acción esté al lado izquierdo, es decir, que se lea *acción si condición*.

El lenguaje mejor conocido en este tipo de lenguajes es Prolog, pero existen otros programas que emplean este paradigma. La aplicación comercial común de tablas de decisión es una forma de programación con base en reglas.

EJEMPLO 3.4 ELABORAR UN PROGRAMA QUE SUME 2 NÚMEROS UTILIZANDO EL PARADIGMA CON BASE EN REGLAS

- Desarrollo

Seleccionando el lenguaje Prolog para la elaboración del programa, el código se presenta a continuación:

```
/* La suma de dos números usando el predicado recursivo: */

suma(X,Y,Z) (X+Y=Z).

suma(0,X,X).                          % 0+X=X

suma(s(X),Y,Z) :- suma(X,s(Y),Z).  % (X+1)+Y=Z <== X+(Y+1)=Z
```

Las técnicas de análisis sintáctico y herramientas como YACC (todavía otro compilador de compiladores) para el análisis sintáctico de programas son técnicas con base en reglas que emplean la sintaxis formal del programa como condición.

3.2 Conceptos manejados en la programación orientada a objetos.

En esta sección se describirán los componentes básicos de un lenguaje de programación orientado a objetos: objetos, clases y métodos; además se plantearán la relación entre dichos componentes así como su modo de acceso. Estos conceptos son propios de este paradigma de programación, por lo que quienes sólo han programado con lenguajes imperativos o funcionales, les resultará totalmente fuera de lo común.

3.2.1 Objetos

Un objeto es una representación detallada, concreta y particular de un "algo". Tal representación determina su identidad, su estado y su comportamiento particular en un momento dado. La identidad de un objeto le permite ser distinguido de entre otros y esto se da gracias al nombre que cada objeto posee.

Un objeto, visto desde la programación orientada a objetos, es una representación de objetos que se tienen en la realidad. Por ejemplo, cuando se mira una pelota, lo que se observa son todas aquellas características que hacen que ésta sea una pelota, es decir, su forma, color, textura, tipo de pelota, etc. Además, siendo un objeto como tal, tiene características y realiza acciones; en el caso de la pelota, ésta puede rebotar, rodar, etc.

Entonces, puntualizando, un objeto es un conjunto de atributos y características que describen al mismo, además de contener un conjunto de métodos que describen lo que puede realizar dicho objeto. Por ejemplo:

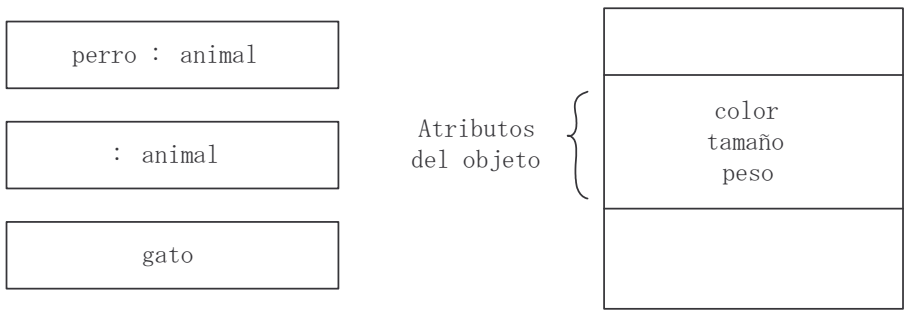


FIGURA 3.1 Representación gráfica de un objeto

De la figura anterior podemos observar que tenemos 3 objetos específicos: perro, cualquier animal y un gato, estos objetos, como se mencionó tienen características específicas. Por ejemplo, cualquier animal, como el perro y/o el gato tienen características específicas tales como un color, un tamaño y un peso, y eso es lo que se trata de mostrar en la figura anterior.

El concepto de objeto se ve mas claro en el siguiente ejemplo.

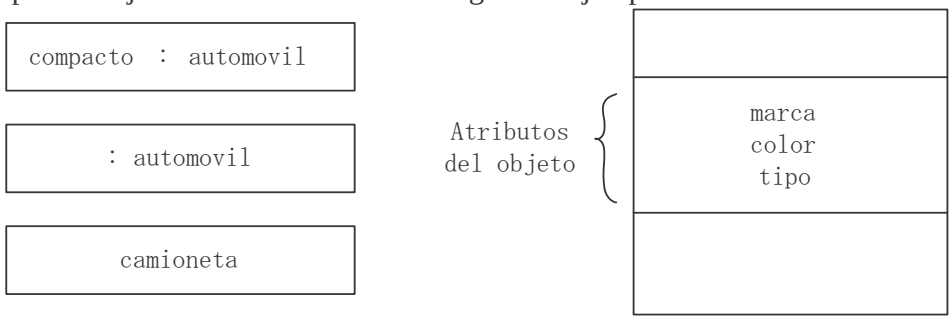


FIGURA 3.2 Segunda representación gráfica de un objeto

Observando el par de ejemplos anteriores, se puede ver que cada uno de los objetos tiene características particulares; en el segundo caso, sabemos que un automóvil compacto, una camioneta o un automóvil cualquiera, como objetos tienen características específicas tales como marca, color y tipo, las cuales los hacen diferente a los demás.

3.2.2 Métodos

Un método describe las acciones que realiza un objeto. Además, la manipulación de los atributos de los objetos se hace por medio de los métodos, es decir, a través de los métodos se asignan o modifican los valores a aquellas características internas del objeto. Los métodos también permiten la interacción con otros objetos por medio de *mensajes*, los cuales se describirán más adelante.

En la representación gráfica de los objetos, los métodos se incluyen en un recuadro colocado debajo del recuadro de los atributos. Por convención, para distinguir el nombre de un atributo con el de un método, a éste último lleva como sufijo un par de paréntesis.

Siguiendo la línea de los ejemplos anteriores, la especificación de los métodos que realizan dichos objetos se muestra en la figura 3.3.

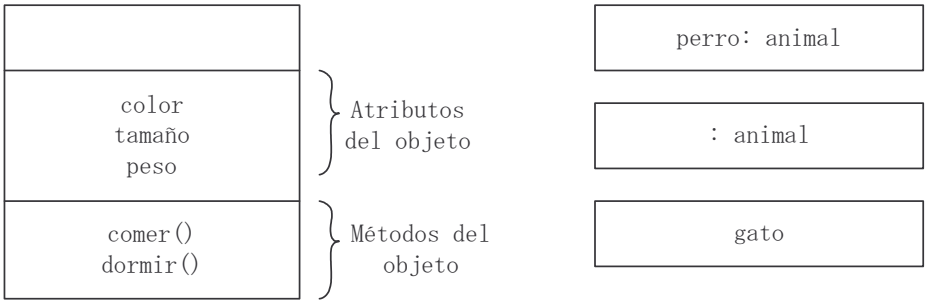


FIGURA 3.3 Representación gráfica del concepto método

De manera específica, se sabe que todo animal, ya sea perro, gato u otro, realizan las actividades tales como comer y dormir; éstas se conocen como “métodos de *x* objeto”, en este caso, métodos de un animal.

En el caso de los automóviles, se sabe que todo automóvil recorre, transporta, etc.; estas actividades son los métodos de dicho objeto. La figura 3.4 muestra la representación del objeto Automóvil con sus métodos y atributos.

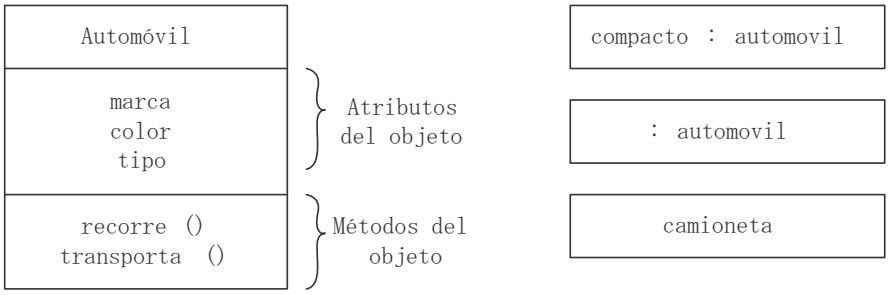


FIGURA 3.4 Representación gráfica del concepto método

3.2.3 Clase

Una clase es la generalización de varios objetos en común, por ejemplo, se puede tener una clase llamada ***Automóvil***, y de ésta se pueden generar varios objetos, que pueden ser diferentes en cuanto a valores de sus atributos, pero siguen compartiendo la misma esencia.

Por lo tanto, una clase se puede ver como un objeto abstracto, que reúne todas las características esenciales (atributos y métodos), para poder generar objetos con dichas características.

Por convención el nombre de la clase se inicia con mayúscula.

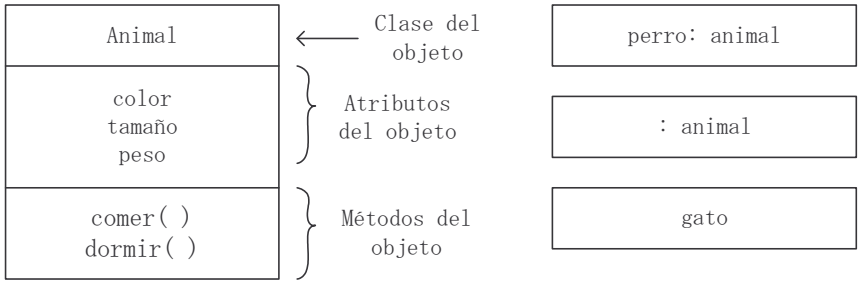


FIGURA 3.5 Representación de la clase Animal

En la figura 3.5 se puede observar que los objetos *perro, animal y gato,* son instancias de una misma clase, ya que, como se mencionó, una clase es una generalización de un tipo específico de objetos; de la misma manera, en la figura 3.6 observamos las diferentes instancias de la clase Automóvil. Una *instancia* es la concreción de una clase. Cada uno de los objetos tiene sus propias características definidas en la clase de la cual son instanciados y comparten la misma implementación de los métodos.

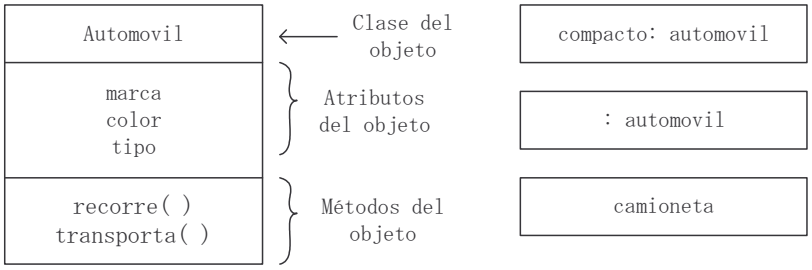


FIGURA 3.6 Representación de la clase Automóvil

3.2.4 Mensajes

Cuando se crean objetos de una clase, los objetos por sí solos no sirven, es necesario invocar a sus métodos para que realicen las acciones para las que fueron creados. Esas llamadas o invocaciones a sus métodos se realizan por medio de *mensajes*, y de esta misma forma pueden trabajar algunos métodos que dependen de otros.

Gráficamente, un mensaje se muestra como una línea dirigida, incluyendo casi siempre el nombre de su operación:

Por ejemplo:

FIGURA 3.7 Representación gráfica *mensajes*

Los objetos interactúan enviándose mensajes unos a otros. Tras la recepción de un mensaje el objeto actuará

3.2.5 Herencia

Una de las herramientas más poderosas en la programación es la reutilización de código, y la forma en que se puede hacer en los lenguajes orientados a objetos es por medio de la herencia.

Cuando se está programando, y se requieren clases que tengan los atributos de otra clase, además de otros, lo que se hace es generar una subclase que herede de una superclase (más

adelante se hablará de subclases y superclases) todos sus atributos y métodos, y además se le pueden agregar otros atributos y métodos propios de la nueva clase.

Se le llama superclase a aquella clase que contiene atributos y métodos que pueden ser utilizados por más de un objeto; esta clase es importante ya que gracias a ella se optimiza el código, pues se está reutilizando.

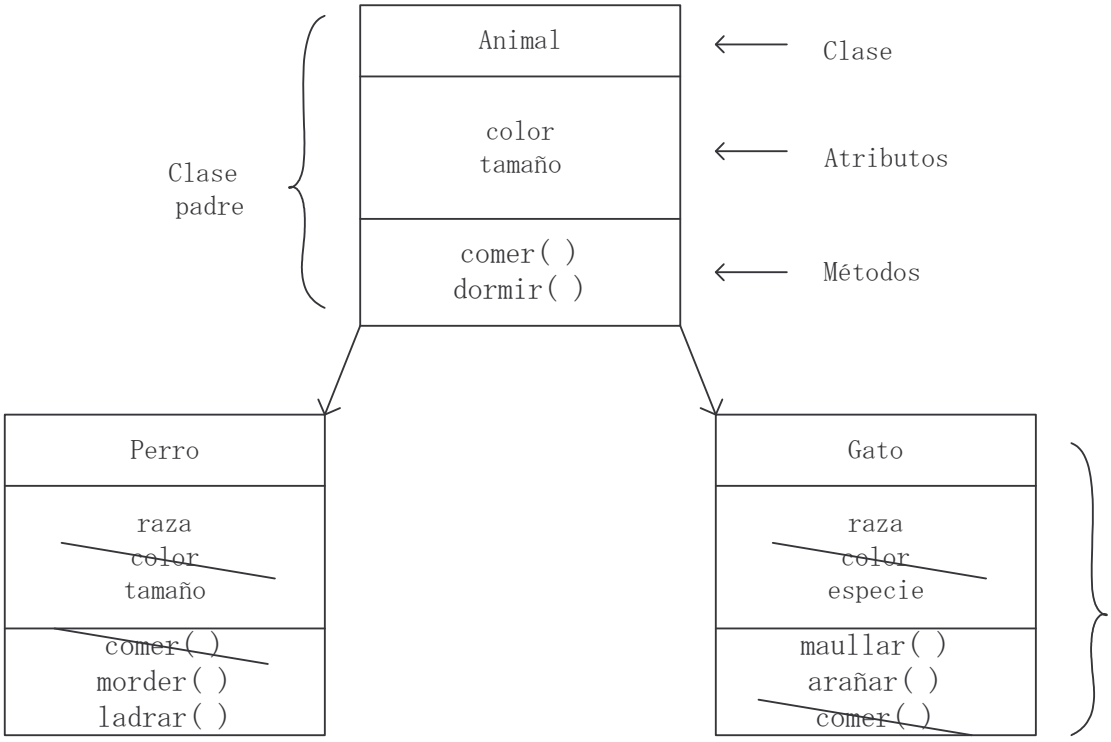


FIGURA 3.8 Representación de la herencia de la clase Animal

En la figura anterior, se puede observar la funcionalidad de la herencia. Existe la clase Animal con ***atributos*** color y tamaño, y ***métodos*** comer y dormir, adicionalmente están las clases Perro y Gato, los cuales heredan los atributos de la clase Animal, ya que ambas clases son animales y tienen los mismos atributos y métodos, razón por la cual no es necesario volver a escribir estos datos en las siguientes clases puesto que ya se heredan de la clase padre. Es aquí donde se observa más claramente la manera en que se disminuye y se reutiliza el código.

3.2.6 Encapsulamiento

Un término que es muy importante en la programación orientada a objetos, es el encapsulamiento. Un claro ejemplo del uso de este término es en los métodos ya que éstos encierran a los atributos y la única forma de acceder a ellos es por medio de dichos métodos.

El encapsulamiento, básicamente es la forma en que se oculta la estructura interna de un objeto (atributos y métodos), y permite manipular al objeto como la unidad mínima.

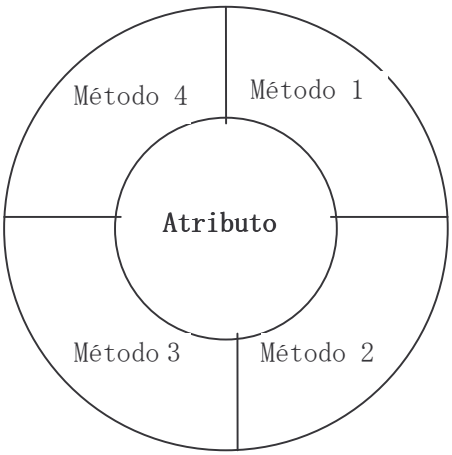


FIGURA 3.9

En la figura podemos observar el funcionamiento del encapsulamiento, ya que muestra claramente que si deseamos llegar a los atributos, primero es necesario pasar por los métodos. Es una forma de proteger la información

3.2.7 Polimorfismo

A diferencia de la programación estructurada, en la programación orientada a objetos, hay un concepto llamado polimorfismo que nos permite tener varios métodos, con el mismo nombre, pero cada método se diferencia de los demás por el número y/o tipo de parámetros que recibe.

Un objeto sólo tiene una forma, la que se le asigna cuando es creado, pero la referencia a ese objeto es polimórfica ya que puede referirse a objetos de diferentes clases. Para que esta referencia sea polimórfica debe haber necesariamente una relación de herencia entre las clases.

Por ejemplo, del diagrama de clases anterior, de la clase Animal donde ejemplificamos la herencia:

- Una referencia a objeto de la clase Perro también puede ser una referencia a objeto de la clase Animal.

- Una referencia a objeto de la clase Gato también puede ser una referencia a objeto de la clase Animal.

3.3 Diseño de programación orientada a objetos. Notación UML.

Es importante mencionar, antes de empezar a explicar el contenido de este tema, que la parte primordial de éste es el *modelado.*

El *modelado* nos ayuda a comprender mejor un sistema que vamos a desarrollar, investigar o analizar; muchas veces la realización de un buen *modelado* nos permite ver simplificaciones del problema o problemas, disminuyendo con esto la complejidad del mismo.

Se modela no sólo en el área de sistemas, también en la industria, en oficinas y hasta en la propia casa, ya que *modelar es simplificar la realidad*. Todo sistema puede ser descrito desde diferentes perspectivas, utilizando diferentes modelos. El simple hecho de modelar *algo* se ayuda a comprender mejor el sistema que se desarrollará, realizará o analizará, sobretodo si se refiere a sistemas complejos, ya que éste nos permitirá comprender mejor el sistema y pues quizá en su totalidad.

Existen 4 principios básicos de modelado:

El primero, muy importante, es *la elección del modelado* ya que éste tendrá una enorme influencia sobre cómo se solucionará o resolverá el problema, por lo tanto, es importante elegir bien el modelo o modelos.

El segundo, es *la expresión del modelo*, esto es que el modelo pueda ser interpretado desde diferentes puntos de vista (niveles de precisión)

El tercero, *el modelo mejor construido es el que estará más ligado a la realidad,* todos los modelos simplifican la realidad, pero es importantes verificar que no esté ausente detalle alguno.

Cuarto y último, *no construir un solo modelo, no es suficiente,* aunque el problema parezca muy trivial, no basta con un modelo, es necesario tener un conjunto de varios modelos casi independientes, para así abordar el problema lo mejor posible.

Revisada esta pequeña introducción a lo que es el modelado, podemos entrar a abordar lo que más nos interesa del presente tema, el *Modelado Orientado a Objetos.*

Cuando se pretende hacer un programa para que realice algo en específico, hay ciertos pasos que hay que seguir, para que se pueda tener una documentación correcta y ordenada de lo que se está haciendo, estos pasos son:

a) Leer y analizar el enunciado del problema.
b) Identificación de las clases necesarias.
c) Definición de atributos y métodos.
d) Elaborar el diagrama de clases.
e) Codificar.

Como se puede observar, los pasos son distintos a los que generalmente se llevan a cabo cuando se utiliza la programación estructurada; ahora en lugar de hacer un diagrama de flujo, lo que se hace es definir las clases, los atributos y los métodos necesarios, para poder realizar un diagrama de clases.

En este tema lo que se abordará, son los conceptos básicos y una visión general de la notación UML, haciendo énfasis en los diagramas de clases, y cómo aplicarlos en el diseño orientado a objetos.

El Lenguaje Unificado de Modelado, o UML por sus siglas en inglés, es una herramienta que se está utilizando actualmente para escribir planos de software.Un lenguaje de modelado es un lenguaje en el que sus reglas y vocabulario se enfocan en la representación física y conceptual de un sistema, por lo tanto, un lenguaje de modelado como lo es UML es un lenguaje estándar.

UML permite visualizar, especificar, construir y documentar sistemas, no sólo de esta rama sino también en sistemas que no son software. En este capítulo se empleará UML en la elaboración de sistemas orientados a objetos y muy particularmente en los programas de dicho paradigma.

Para entender UML es necesario tener un modelo conceptual de lenguaje, y esto requiere tener 3 elementos principales:

- Bloques básicos de construcción de UML
- Reglas para combinar estos bloques básicos.
- Y mecanismos comunes que se aplican con UML

Una vez que se han comprendido estos elementos, se pueden leer y entender modelos conceptuales UML además de poder crear algunos modelos básicos.

UML contiene 3 clases de bloques de construcción:

1. Elementos
2. Relaciones
3. Diagramas

Los elementos son abstracciones que son ciudadanos de primera clase en un modelo; las relaciones ligan los elementos entre sí; los diagramas agrupan colecciones interesantes de elementos.

Existen cuatro tipos de *elementos en UML*: estructurales, de comportamiento, de agrupación y de anotación, éstos, son los bloques básicos que son utilizados en la construcción de sistemas orientados a objetos por UML.

Los elementos estructurales, los llamados nombres de los modelos, son la parte estática de un modelo, representan cosas materiales o conceptuales. Un ejemplo de clase estructural es una ***clase***, ya que ésta muestra la descripción de objetos que tienen en común los mismos atributos y realizan las mismas acciones u operaciones.

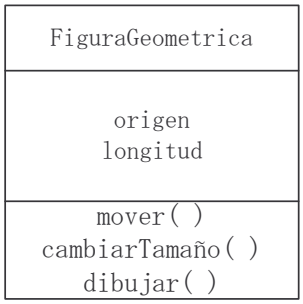


FIGURA 3.10 Elemento estructural de un objeto

Los elementos de comportamiento, son las partes no estáticas de UML, son las acciones de un modelo, representan comportamiento, son mejor conocidos como verbos.
Por ejemplo:

FIGURA 3.11 Ejemplo de un elemento de comportamiento conocido como verbo

El ejemplo de la figura 3.10 muestra lo que se conoce como ***interacción***, que no es otra cosa que el conjunto de mensajes intercambiados entre objetos; ésta además de indicar una acción incluida en el mensaje, sirve también para enlazar, es decir, conectar objetos. Podemos observar entonces, que un mensaje se representa con una línea dirigida, y la acción a realizar.

Otro tipo de elemento de comportamiento es el de ***máquina de estados***, el cual es un comportamiento que especifica la secuencia de estados por la que pasa un objeto, se representa con un rectángulo con esquinas redondeadas, incluyendo la acción; por ejemplo, los estados mostrados en la figura 3.11 , son elementos de comportamiento.

FIGURA 3.12 Elementos de comportamiento de máquina de estados

Los *elementos de agrupación*, son las partes que organizan a los elementos de un modelo UML. Son como unas cajas en las cuales se puede descomponer un modelo.

Los *elementos de anotación* son las partes que explican los modelos UML. Son comentarios que sirven para explicar, describir o aclarar algún elemento de un modelo. El tipo único y principal es el llamado *nota* la cual se representa de la siguiente manera

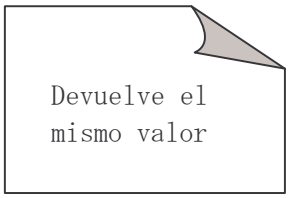


FIGURA 3.13 Ejemplo de elemento de anotación

Ahora veamos otro bloque de construcción UML, las *relaciones UML*, de las cuales existen cuatro principales tipos: de dependencia, de asociación, de generalización y realización, estas relaciones, son bloques básicos en la construcción de modelos UML.

La relación de *dependencia*, es una relación semántica (significado) entre dos elementos, en la cual, el cambio a un elemento (independiente) puede afectar la semántica del otro elemento (dependiente). Esta dependencia se representa mediante una línea discontinua, a veces dirigida, y en ocasiones con una etiqueta, como lo muestra el siguiente ejemplo:

FIGURA 3.14 Representación de relación de dependencia

La relación de *asociación* describe enlaces que conectan a los objetos. Gráficamente se representa como una línea continua, en ocasiones dirigida, con etiqueta u otros elementos, como se muestra a continuación:

FIGURA 3.15 Representación de relación de asociación

Las relaciones de *generalización* son llamadas también de especialización/generalización en las que los objetos del elemento *hijo* pueden sustituir a los objetos del elemento *padre* (o general), de tal manera que el hijo comparta la misma estructura y comportamiento del

padre. Se representa gráficamente mediante una línea continua con una punta de flecha vacía que se dirige (o apunta) al padre, como se muestra a continuación:

FIGURA 3.16 Representación de relación de generalización

Las relaciones de *realización* son utilizadas en la semántica entre clasificadores, en donde un clasificador especifica un contrato que otro clasificador garantiza, se cumplirá. Se representa gráficamente mediante una combinación de relación de dependencia y generalización.

FIGURA 3.17 Representación de realización de realización

3.3.1 Diagramas UML

Un diagrama es la representación gráfica de un conjunto de elementos, en el que se muestran como tales los elementos y sus relaciones. Los diagramas se utilizan para poder entender, visualizar y resumir un sistema desde diferentes puntos de vista, además de facilitar el entendimiento del mismo.

UML tiene nueve tipos de diagramas:

1. Diagrama de clases
2. Diagrama de objetos
3. Diagrama de casos de uso
4. Diagrama de secuencia
5. Diagrama de colaboración
6. Diagrama de estados (statechart)
7. Diagrama de actividades
8. Diagrama de componentes
9. Diagrama de despliegue

El *diagrama de clases* muestra al conjunto de clases, interfaces y relaciones. Estos diagramas son los más comúnmente utilizados en los sistemas orientados a objetos, por lo tanto son los diagramas que describiremos a detalle para emplearlos en el desarrollo de programas orientados a objetos.

El *diagrama de objetos* muestra sólo a los objetos y sus relaciones, representan instancias de los elementos que se encuentran en los diagramas de clases. Estos diagramas son importantes en el modelado del comportamiento de un sistema.

El *diagrama de casos de uso* muestra a los casos de uso y actores (variación especial de clases) y a su vez sus relaciones.

Existen los llamados *diagramas de interacción* que muestran como su nombre lo indica interacciones entre sus objetos y sus relaciones, incluyen también mensajes. Dentro de estos diagramas se encuentran los *diagrama de secuencia y colaboración*, que se encargan de resaltar la ordenación temporal de mensajes, y resaltar la organización de la recepción de mensajes y envío de objetos, respectivamente.

El *diagrama de estados* muestra a la máquina de estados, contiene estados, eventos y actividades.

El *diagrama de actividades* es un caso especial del diagrama de estados, ya que aquí sólo se muestra el flujo de actividades en el sistema.

El *diagrama de componentes* contiene las dependencias y también organización entre el conjunto de elementos.

El *diagrama de despliegue* muestra a los nodos en tiempo de ejecución los componentes que tiene cada uno de ellos.

Estos diagramas son los que se utilizan con mayor frecuencia, y como se mencionó anteriormente, se dará especial énfasis al estudio de *diagrama de clases*.

Ahora bien, un modelo es una representación abstracta de un sistema, el cual, contiene los aspectos más relevantes del propósito de nuestro sistema. Con esta información se generan los diagramas, los cuales son de diversos tipos como:

- Diagrama de Clases.

  Este modelo, representa la solución del software, los elementos que lo conforman son las clases, y las relaciones entre clases (asociaciones).

- Diagrama de Casos de Uso.

  Es una representación de la forma en que trabaja, o se desea que trabaje, un sistema. Este diagrama no es propiamente un enfoque orientado a objetos, pero es una técnica para capturar requisitos.

- Diagrama de Estados.

  Muestra una especie de diagrama de flujo, dentro del cual se muestra cómo interactúa un objeto dentro de una aplicación.

- Diagrama de Secuencia.

  Muestra una especie de diagrama de estados, y es usado para describir un método, un caso de uso, o un proceso del sistema.

Éstos son algunos de los diagramas más comunes dentro de la notación UML, existen otros diagramas, pero no son de utilidad para el objetivo de este capítulo. De los tipos de diagramas que se mencionaron anteriormente, el más común es el diagrama de clases, el cual muestra las clases que están involucradas en la realización de una acción dentro de nuestro sistema.

Enfocándonos ya en el objetivo del estudio de sistemas orientados a objetos, para este tipo de sistemas, las clases y los objetos son la herramienta fundamental.

Ya se ha hablado anteriormente de conceptos tales como clase, objeto y atributos, por lo que eso facilitará la correcta comprensión de las siguientes definiciones.

Primeramente es necesario explicar los elementos básicos que se utilizan en la elaboración de un diagrama de clases, ya que éste es el diagrama correspondiente a los *sistemas orientados a objetos*.

Los elementos que con mayor frecuencia se utilizarán a partir de este momento son: clase, nombres, atributos y métodos u operaciones, como se muestra en la figura 3.18.

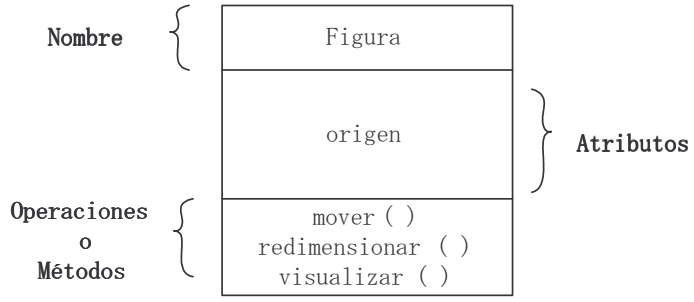


FIGURA 3.18 Gráfica de una clase en UML

3.3.2 Clase

Una clase es la descripción de un conjunto de objetos que comparten las mismas características, esto es, que contienen los mismos atributos, realizan las mismas actividades u operaciones y mantienen las mismas relaciones, además de contener la misma semántica.

Gráficamente una clase se representa mediante un rectángulo, una caja con 3 divisiones, en las cuales, se resalta las partes más importantes de una abstracción: nombre, atributos y operaciones (o métodos) de una clase.

3.3.3 Nombre

Cada clase ha de tener un nombre que hace se distinga de otras clases, este nombre es una cadena de texto. Normalmente el nombre de una clase inicia con letra mayúscula; si contiene varias palabras, éstas deberán unirse y cada una deberá iniciar con mayúscula, por ejemplo: Cliente, Ecuación, SistemaEcuación, SensorTemperatura.

La figura 3.19 muestra gráficamente la representación del nombre de la clase.

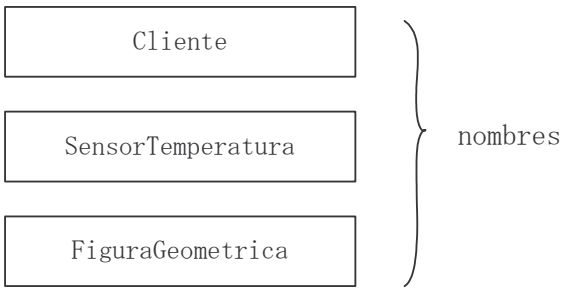


FIGURA 3.19 Gráficas de nombres de clases en UML

3.3.4Atributos

Un atributo es una propiedad que contiene la clase, representa alguna propiedad del elemento que se está modelando y que además es compartida por todos los elementos de esa clase. Un atributo es una abstracción de un tipo de dato que puede tener un objeto de la clase, un objeto de una clase tendrá siempre valores específicos para cada atributo de la clase.

El nombre de un atributo es texto, al igual que una clase. Normalmente un atributo se escribe con letras minúsculas pero si está compuesto por más de una palabra, las siguientes, iniciarán con letra mayúscula; por ejemplo: nombre, valor, nombreAlumno valorNegativo.

Gráficamente los atributos se encuentran justamente debajo del nombre de la clase, como se muestra en la figura 3.19.

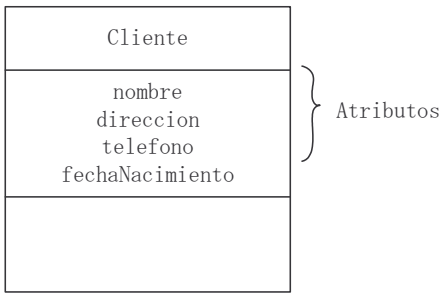


FIGURA 3.20 Representación gráfica de una clase con atributos en UML

3.3.5 Métodos u operaciones

Un método es la implementación de una acción que puede ser realizada por cualquier objeto de la clase para que éste muestre un comportamiento. Concretamente, un método es una abstracción de algo (una acción) que puede realizar un objeto de la clase y que puede ser compartido con los demás objetos de la misma clase. El nombre de un método u operación, puede ser texto también, en la práctica el nombre de estos métodos es un verbo.

Los métodos u operaciones se escriben de la misma manera que un atributo, sólo que al final del texto se escriben paréntesis, por ejemplo: mover() estáVacío().

En la figura 3.21 se observa los métodos de la clase Triángulo, los cuales son mover ( ), cambiarTamaño( ), dibujar( )

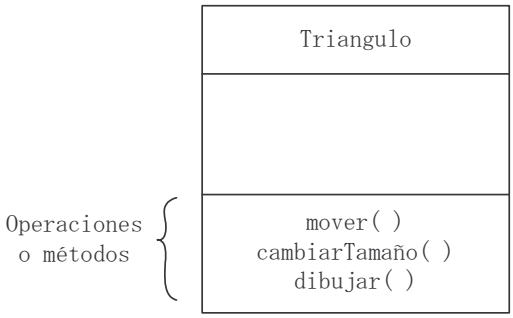


FIGURA 3.21 Representación gráfica de una clase con métodos en UML

En la figura 3.22 se muestra una gráfica completa de una clase en UML, se pueden observar las tres partes que la componen: el nombre de la clase, sus atributos y sus métodos.

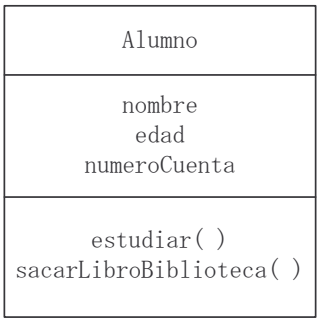


FIGURA 3.22 Gráfica completa de una clase en UML

La figura muestra que esta clase tiene por nombre “Alumno”; como atributos “nombre”, “edad”, “numeroDeCuenta”; y finalmente como métodos u operaciones tiene “estudiar()” y “sacarLibroDeBiblioteca()”.

Dentro del diagrama, observamos que la clase consta del ***nombre de la clase*** en la parte superior, los ***atributos*** en la parte central, y los ***métodos*** en la parte inferior del recuadro, como se explicó anteriormente. Tanto el nombre de la clase, como los nombres de los atributos y métodos, llevan la nomenclatura que se utiliza en el paradigma orientado a objetos, para facilitar su comprensión. Estas reglas se deben recordar perfectamente; sintetizadas son las siguientes

- No debe existir ningún espacio en blanco, ni guiones bajos dentro de los nombres.
- En el caso de las clases, la primera letra de cada palabra que conforma el nombre, deberá ser mayúscula, por ejemplo: Alumno, Profesor, MetodosNumericos, etc.
- Para los nombres de atributos, las letras de todo el nombre deberán ser minúsculas; en el caso de que incluya varias palabras, a partir de la segunda deberá iniciar con mayúscula; por ejemplo: nombre, edad, numeroDeCuenta, etc.
- Los nombres de métodos, siguen la misma nomenclatura que con los atributos, la diferencia es que los métodos llevan un par de paréntesis después del nombre.

3.3.6 Relaciones entre clases

En el modelado orientado a objetos hay 3 tipos importantes de relaciones: de ***dependencia***, las cuales representan las relaciones de uso entre ellas; de ***generalización***, las cuales conectan clases generales con sus especializaciones, se conocen como relaciones subclase/superclase hijo/padre, y las de ***asociación*** que representan relaciones estructurales entre objetos. UML tiene una representación gráfica para cada uno de estos tipos de dependencia.

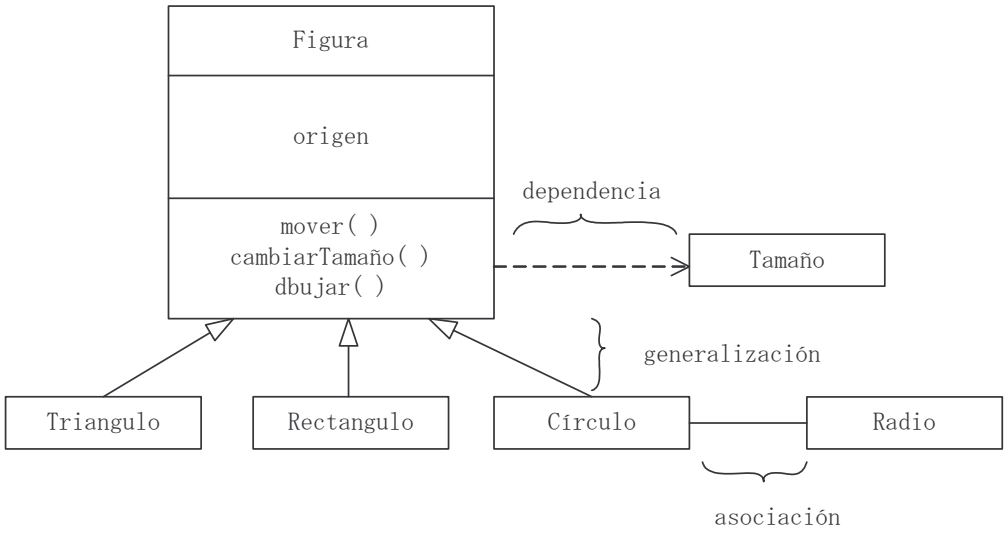


FIGURA 3.23 Ejemplo de uso de los tres tipos de relaciones entre clases

Se iniciará definiendo la palabra *relación*, la cual indica simplemente la conexión entre elementos. Gráficamente se representa con una línea, que dependiendo del tipo de relación se emplea un formato de línea diferente.

Relación de dependencia. Ésta declara un cambio en cuanto a la especificación de un elemento, esto es, que puede afectar a otro elemento que la utiliza, pero no necesariamente de manera inversa. Gráficamente se representa mediante una línea discontinua dirigida hacia el elemento del cual se depende. Las dependencias se utilizarán cuando queramos indicar que un elemento utiliza a otro. En algunos casos, las dependencias se utilizan en el contexto de las clases, para así indicar que una clase utiliza a la otra como argumento en una operación. Por ejemplo.

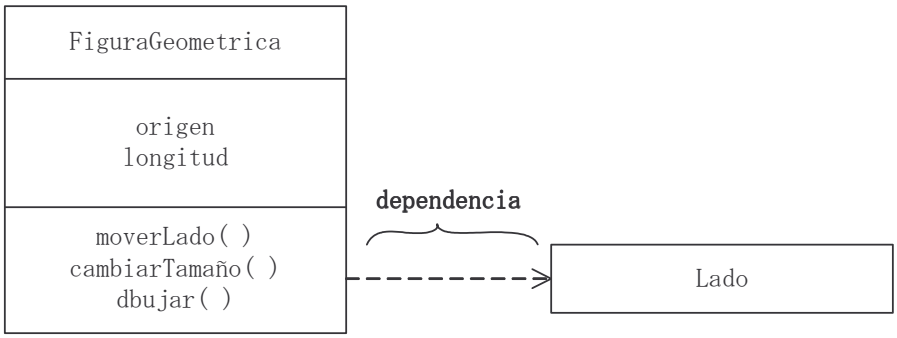


FIGURA 3.24 Representación de la relación de dependencia en UML

Relación de generalización. Es la relación que existe entre un elemento general (superclase o padre) y un caso más específico de ese elemento (subclase o hijo). Conocida como relación “es-un-tipo-de”. Este tipo de relación indica que los objetos hijos se pueden emplear en cualquier lugar donde llegue a estar el padre, pero no de manera inversa, es decir, el hijo puede sustituir al padre.

Una clase hija hereda las propiedades de la clase padre, es decir, todos sus atributos y métodos. Un método de un hijo, con el mismo contexto que el método del padre, redefine la operación del padre, esto se conoce comúnmente como *polimorfismo.* Gráficamente se representa con una línea continua dirigida, con punta de flecha vacía que apunta al padre siempre:

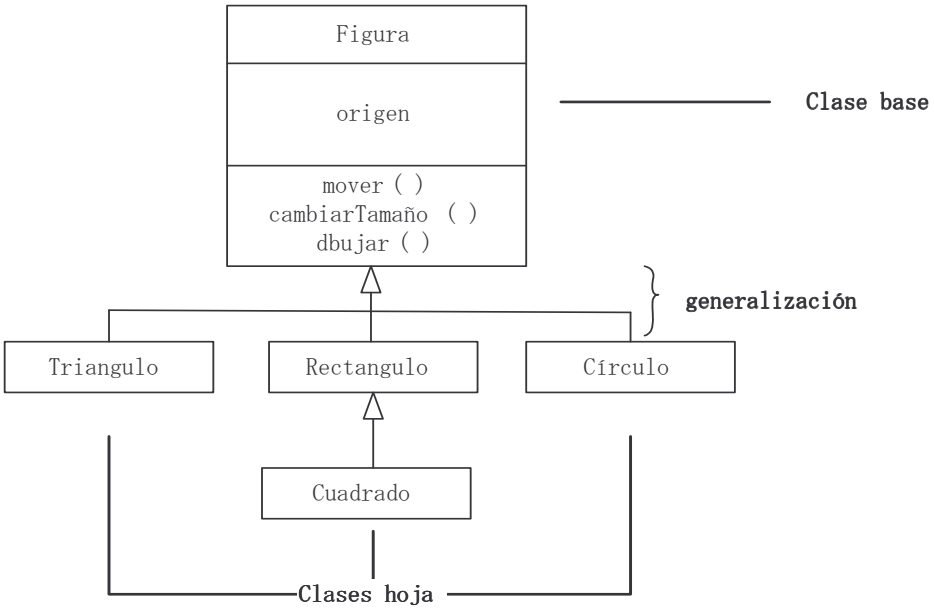


FIGURA 3.25 Representación de la relación de generalización y tipos de clase resultantes

Existen varios tipos de herencia. En una clase puede haber o no padres, pero también puede haber más de uno. Una clase que no tiene padres y que tiene uno o más hijos se le llama *clase base*. Una clase que no tiene hijos se le llama *clase hoja*, una clase con un solo padre se le llama *clase simple* y una clase con más de un padre se le llama *herencia múltiple.*

Relación de asociación. Ésta es una relación estructural que muestra que los objetos de un elemento están conectados con los elementos de otro. Cuando se tiene una asociación entre dos clases podemos acceder desde el objeto de una clase al objeto de la otra clase y viceversa. Gráficamente se representa con una línea continua que conecta a la misma o diferentes clases.

Esta dependencia en particular tiene cuatro adornos que se aplican a las asociaciones.

Nombre, una asociación lo puede tener, ya que le sirve para describir la naturaleza de la relación; para que no exista ambigüedad en el significado, se le puede proporcionar una dirección al nombre por medio de una flecha, la cual apuntará a la dirección a la cual se desea se lea el nombre.

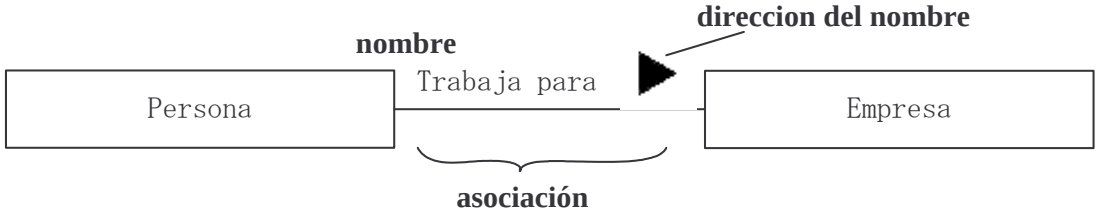


FIGURA 3.26 Representación de la relación de asociación con nombre

Rol, en una asociación cada clase tiene un rol específico, es sencillamente la etiqueta o marca que presenta cada clase de un extremo al otro.

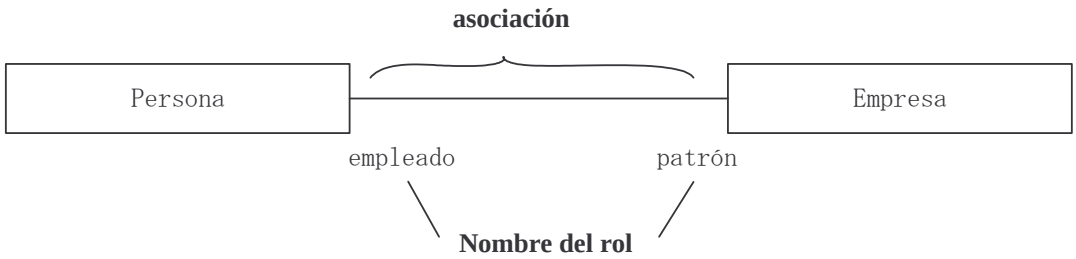


FIGURA 3.27 Representación de la relación de asociación con rol

Multiplicidad, representa la relación estructural entre objetos, señala cuántos objetos pueden conectarse a través de una instancia de asociación.

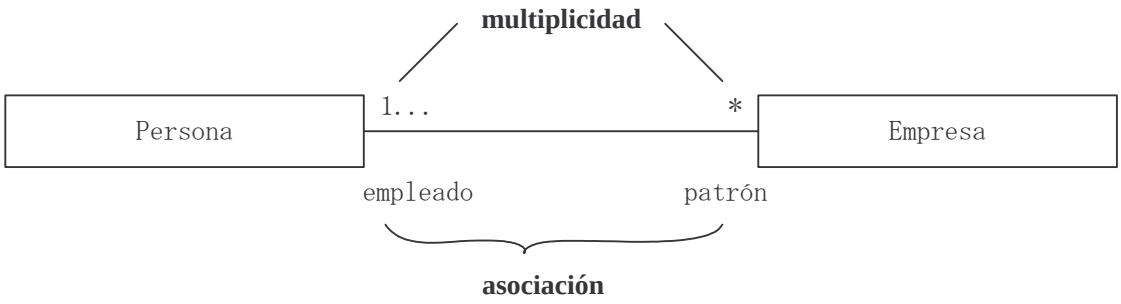


FIGURA 3.28 Representación de la relación de asociación con rol y multiplicidad

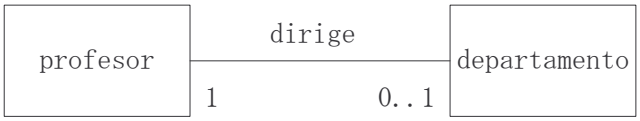
La multiplicidad de una asociación determina cuántos objetos de cada tipo intervienen en una relación: “el número de instancias de una clase que se relacionan con UNA instancia de la otra clase”.

- Cada asociación tiene 2 multiplicidades (una para cada extremo de la relación)
- Para especificar la multiplicidad de una asociación hay que indicar la multiplicidad mínima y máxima (mínima..máxima) de acuerdo a la siguiente tabla

MULTIPLICIDAD	SIGNIFICADO
1	Uno y solo uno
0 . . 1	Cero o uno
N . . M	Desde N hasta M
*	Cero o varios
0 . . *	Cero o varios
1 . . *	Uno o varios (al menos uno)

- La multiplicidad mínima es cero, la relación es opcional.
- Una multiplicidad mayor o igual a 1 establece una relación obligatoria.

Por ejemplo:

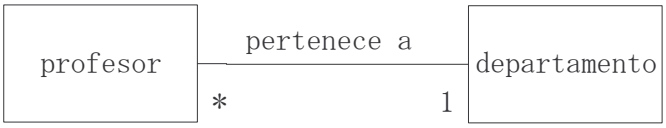


a)

FIGURA 3.29 a) Ejemplo de representación multiplicidad

En la figura a) se puede observar que la relación de multiplicidad es la siguiente:

Un profesor puede o no dirigir un departamento
A un departamento lo dirige uno y solo un profesor.



b)

FIGURA 3.29 b) Ejemplo de representación multiplicidad

En la figura b) se puede observar que la relación de multiplicidad es la siguiente:

Todo profesor pertenece a un departamento.
A un departamento pueden pertenecer varios profesores

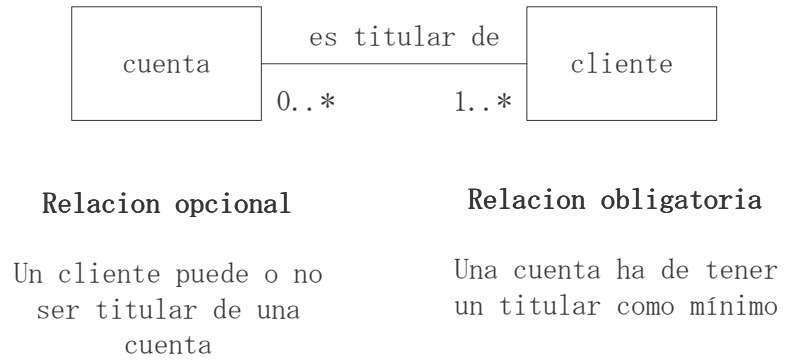


FIGURA 3.29 c) Ejemplo de representación de multiplicidad

En la figura c) se ejemplifica la relación de multiplicidad que existe cuando se tienen relaciones del tipo “Cero o varios” y “Uno o varios (al menos uno)”.

Agregación, una asociación entre dos clases representa una relación estructural entre “iguales” es decir, ambas están conceptualmente en el mismo nivel, ninguna es más importante que la otra.

Cuando se quiere modelar una clase que lo es todo y ésta consta a su vez de elementos más pequeños, se dice que tenemos una relación de agregación la cual representa una relación de tipo “tiene-un”. Gráficamente se representa añadiendo a una asociación normal, un rombo vacío en la parte del todo.

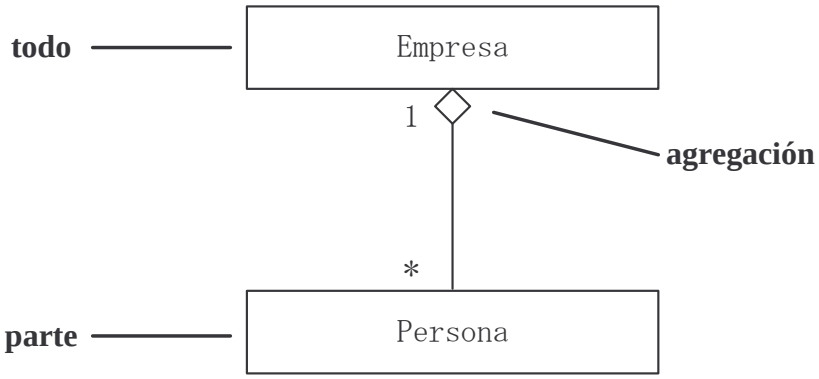


FIGURA 3.30 Representación de la relación de agregación

En la figura 3.30 se muestra claramente el concepto de *agregación*, y se interpreta el ejeplo de la siguiente manera, a una empresa le pertenecen muchos departamentos (de acuerdo a la multiplicidad, mencionada anteriormente) y, un departamento pertenece necesariamente a una empresa.

Un modelado de dependencia simple es el tipo más común de relaciones, una clase sólo utiliza a otra y como parámetro de operación.

3.3.7 Herencia

En un diagrama de clases, cuando se desea indicar que una clase es subclase de otra, lo que se hace es utilizar una flecha que se caracteriza por tener en la punta un triángulo vacío, como se mencionó anteriormente. Por ejemplo, se tiene la clase Persona, de la cual se creará la clase Alumno, el diagrama de clases queda de la siguiente manera:

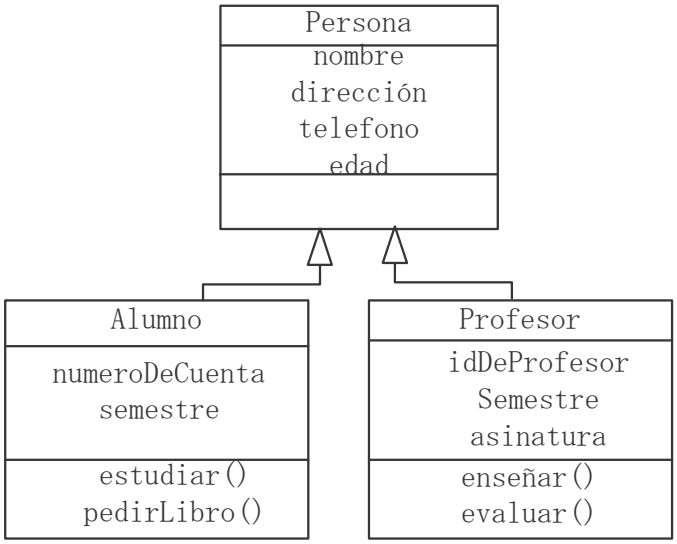


FIGURA 3.31 Representación de herencia en UML

Si se creara un objeto de la nueva clase Alumno, además de *numeroDeCuenta* y *semestre*, tendría los atributos heredados de Persona, es decir, *nombre*, *direccion*, *telefono* y *edad*.

En la figura 3.32 se observa un ejemplo completo del manejo de clases y relaciones, en un diagrama de clases. Revisándolo a detalle se puede obtener la siguiente información:

- Uno o varios vuelos tienen de 1 a 2 pilotos y de 1 a 2 pilotos están en muchos vuelos.
- Un vuelo puede tener muchas reservaciones y varias reservaciones tienen un solo vuelo.
- Uno o varios vuelos tienen un solo avión y un avión tiene muchos vuelos
- Uno o varios vuelos están en una y solo una compañía Aérea y una compañía Aérea tiene muchos vuelos
- Un avión, puede tener de 1 a 5 motores, 1 mínimo.
- Avión comercial o militar son un tipo de avión.
- Avión de pasajeros y avión de carga son un tipo de avión comercial.

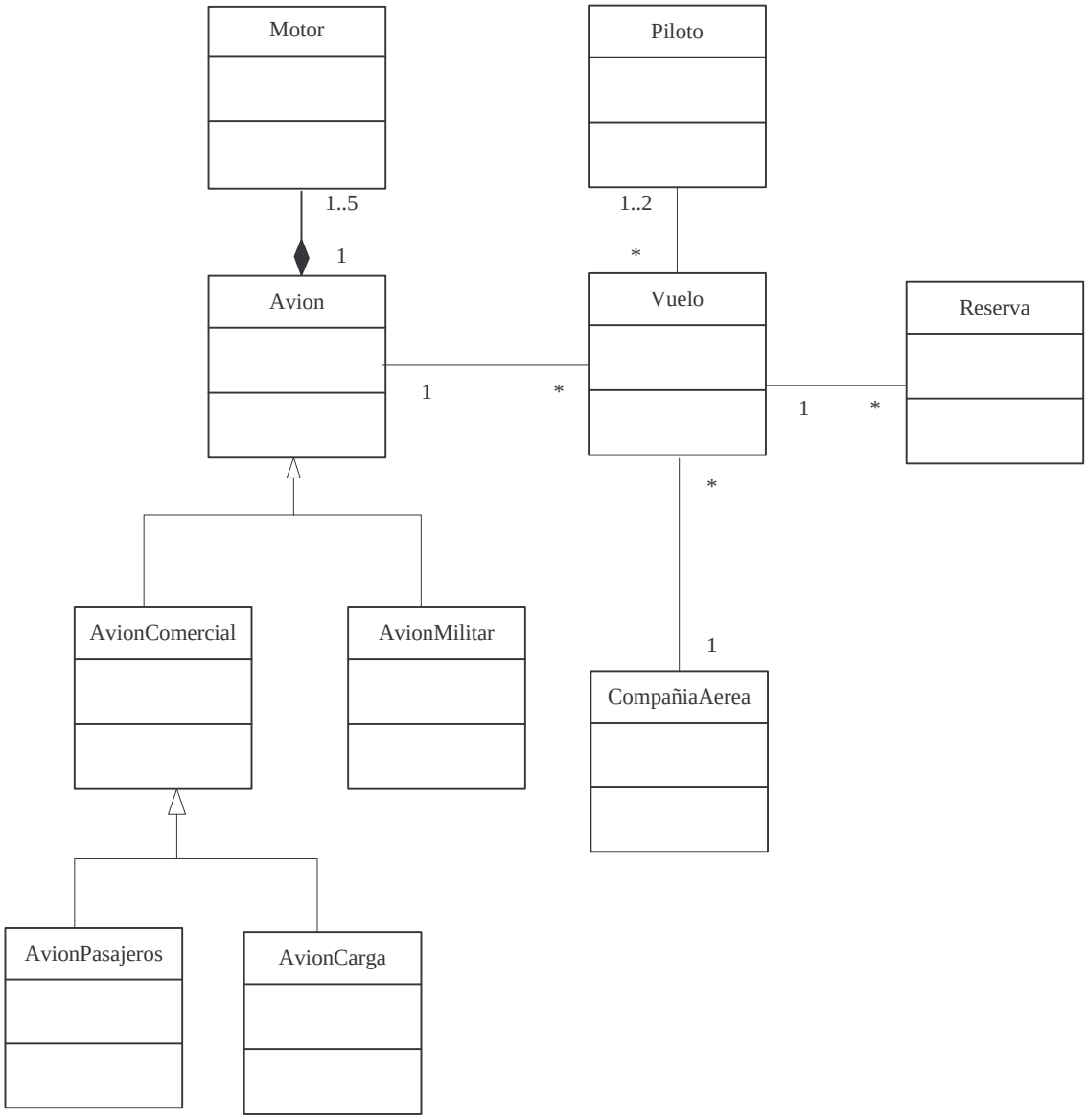


FIGURA 3.32 Manejo de diagrama de clases

EJEMPLO 3.5 ELABORAR UN DIAGRAMA DE CLASES PARA CALCULAR LAS RAÍCES REALES DE UNA ECUACIÓN DE 2^o GRADO.

-Análisis y solución del problema

Las raíces reales de la expresión $ax^2 + bx + c = 0$ se obtienen a través de la fórmula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Trabajando como se ha venido haciendo en los capítulos anteriores, se elaborará primeramente el pseudocódigo; por lo cual se deben definir los tipos de valores que se van a utilizar, las variables que se emplearán y las operaciones que se aplicarán.

Se necesitan tres valores inicialmente, *a*, *b* y *c*, los cuales son variables de tipo real y con la restricción de que *a* debe ser diferente de *cero*, estos valores son los coeficientes de la ecuación. También se utilizarán dos variables más, X1 y X2, las cuales almacenarán losl valores de las raíces reales de la ecuación.

En cuanto a los procedimientos, se identifican los siguientes: se leen los valores de *a*, *b* y *c*, después se calcula el discriminante, dependiendo de este valor se decidirá si finaliza la ejecución o si continúa, ya que si el valor del discriminante es mayor o igual a cero, calculamos el valor de X1 y X2, en caso contrario, finaliza el procedimiento.

-Explicación de las variables

a, *b* y *c*: son variables de tipo real.
D: variable de tipo real que almacenará el discriminante de la ecuación.
X1: variable de tipo real, contendrá la primera raíz obtenida.
X2: variable de tipo real, contendrá la segunda raíz obtenida.

-Pseudocódigo

1. Leer valores de *a*, *b* y *c*.
2. Calcular *D*
  2.1. *b*^2 – 4 * *a* * *c*
3. Si $D \geq 0$ entonces
  3.1. Hacer: $X_1 \leftarrow ((-b) + D\text{^}0.5)/2 * a$ y
    $X_2 \leftarrow ((-b) - D\text{^}0.5)/2 * a$
  3.2. Imprimir el valor de las raíces reales obtenidas.
4. {Fin del condicional del paso 3}

-Diagrama de Clases

Con base en el análisis se pueden identificar las siguientes clases:

Una clase ***ResolverEcSegundoGrado***, la cual tendrá como atributos los ***valores reales*** de ***a***, ***b*** y ***c***, así como las raíces ***X1*** y ***X2*** las cuales serán también reales; como métodos tendrá ***resolverCondicional( )*** y ***solucion( )***. El método ***resolverCondicional( )*** invoca al método

calculaDiscriminante( ) de la clase *Discriminante*, haciendo uso de atributos de la clase *ResolverEcSegundoGrado*. El método *solución( )* invoca a los métodos *calculoRaizX1( )* y *calculoRaizX2( )* de la clase *Raices*, y al método *imprimirPantalla( )* de la clase *ImpresionResultado*.

Las relaciones que existen entre las clases son de dependencia, ya que los métodos de la clase *ResolverEcSegundoGrado* necesitan hacer uso de los métodos de las otras tres clases.

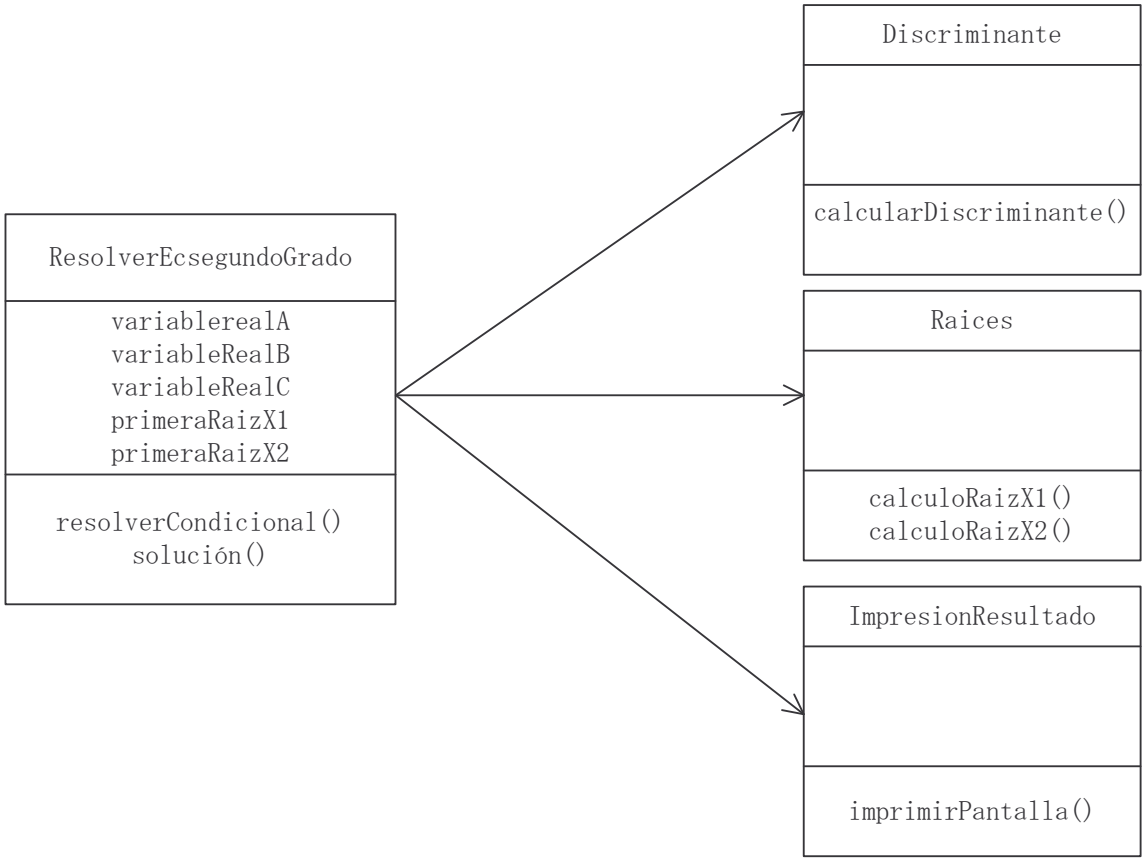


FIGURA 3.33 Diagrama de clases del ejemplo 3.5.

EJEMPLO 3.6 ELABORAR UN DIAGRAMA DE CLASES PARA DETERMINAR SI UN NÚMERO ENTERO DADO ES POSITIVO, NEGATIVO O NULO.

- Análisis y solución del problema

Primero se debe definir el tipo de valores y/o variables que se utilizarán así como las operaciones que se emplearán.

Se necesita para este caso sólo un valor, *num*, el cual será la variable entera que representará al número que se ingrese.

Se recibirá el valor de *num* el cual pasará por un condicional, mismo que determinará su resultado o significado. Si el valor *num* que se ingresa es mayor que cero, se imprimirá *positivo*, y se terminará el procedimiento, caso contrario, se llevará a cabo otra condicional en la cual, si el valor de *num* es igual a cero se imprimirá *nulo* o si es menor que cero, se imprimirá *negativo*, finalizando con el resultado obtenido el procedimiento.

El problema debe realizar lo siguiente:

corrida	dato ingresado	imprimir
1	9	positivo
2	-4	negativo
3	1	positivo
4	0	nulo
4	-2	negativo

-Explicación de las variables

num: es una variable de tipo entera.

-Pseudocódigo

{*num* es una variable de tipo entero}
1. Leer *num.*
2. Si *num* > 0
   entonces
   Imprimir “Positivo”
   sino
   2.1 Si *num* = 0
   entonces
   Imprimir “Nulo”
   sino
   Imprimir “Negativo”
   2.2 {fin del condicional del paso 2.1}
3. {fin del condicional del paso 2}

-Diagrama de clases

Con base en el análisis se pueden definir las siguientes clases:

Una clase *PositivoNegativoNulo*, la cual tendrá como atributo el *valor entero* de *num*; y dos métodos *analizar( ) y concluir( ).* El método *analizar( )* invocará a la clases *Positivo* y *NegativoNulo*, que tomando el atributo de la clase *PositivoNegativoNulo*, realizarán los métodos *verificaCondicionalUno( )* y *verificaCondicionalDos( )* respectivamente. El método *concluir( )* invocará a la clase *ImprimeResultado* para utilizar el método

imprimeEnPantalla( ). La relación que existe entre estas tres clases es de dependencia, ya que la clase *PositivoNegativoNulo* necesita usar los métodos de *Positivo, NegativoNulo* e *ImprimeResultado.*

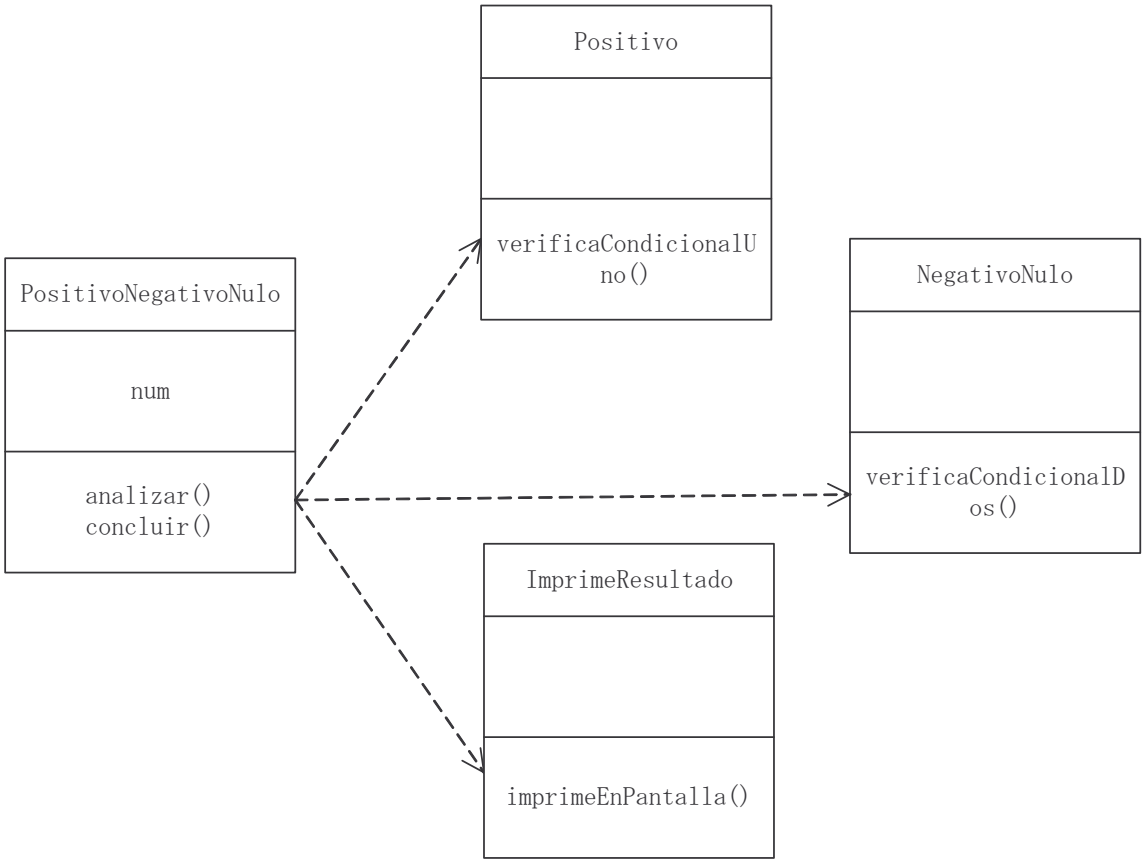


FIGURA 3.34 Diagrama de clases del ejemplo 3.6

Los diagramas de clases de los ejemplos 3.5 y 3.6 no son la solución única de dichos problemas. Si bien se pudo haber empleado una sola clase para cada diagrama, se decidió utilizar varias clases con la finalidad de ejercitar el manejo de relaciones entre clases.

Ejercicios Propuestos

1) Utilizando el paradigma de procedimiento imperativo, elaborar un programa que realice la suma de dos matrices de 3x3, utilizando un lenguaje de programación que corresponda a este paradigma, explicarlo detalladamente, haciendo especial énfasis en que realizamos "instrucciones".

2) Proporcionar un ejemplo en el cual se utilice el paradigma declarativo funcional (LISP)

3) Representar gráficamente, la clase Area, para polígonos regulares.

4) Crear una clase que calcule la velocidad v y a partir de ella calcular la Energía cinética, $E_c = \frac{1}{2} m v^2$. Representar este procedimiento gráficamente, mostrando la herencia que existe entre ellas.

5) Elaborar un diagrama de clases en el cual utilice la relación de *dependencia* de UML

6) Elaborar un diagrama de clases en el cual utilice la relación de *asociación* de UML

7) Elaborar un diagrama de clases en el cual utilice la relación de *generalización* de UML

8) Elaborar un diagrama de clases en el cual utilice la relación de *realización* de UML

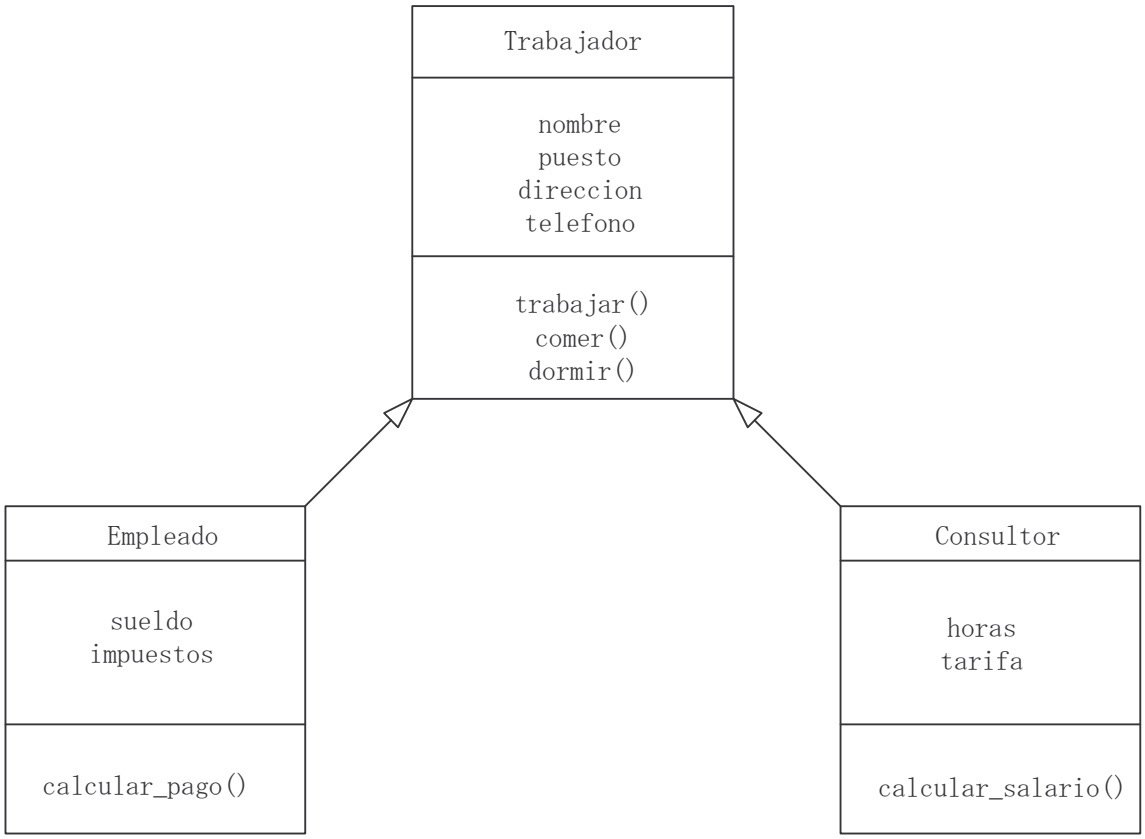
9) A manera de repaso, mencionar cuáles son los diagramas más utilizados por UML e indique cuáles son sus principales características.

10) Elaborar el diagrama de clases para el siguiente enunciado: "La suma de dos números".

11) Elaborar el diagrama de clases del ejemplo 3.2.

12) Sea el siguiente enunciado: "En una microempresa se tienen cerca de 50 empleados, en ella, se lleva un registro de nombre y salario de cada uno de ellos. Existen 4 departamentos con diferentes empleados cada uno y sólo dos de ellos son el gerente y subgerente, los demás son subordinados de ellos". Realizar el diagrama de clases de este enunciado, enfatizando la herencia entre clases.

Describir el siguiente diagrama de clases, utilizando los conceptos vistos en este capítulo.



13) En una escuela, se maneja la siguiente información: se lleva un registro de entradas al plantel, en la cual todos, sin excepción alguna, registran su nombre, teléfono y edad; pero cada persona tiene un cargo u ocupación diferente, ya que existen profesores, alumnos y personal de limpieza, así como visitantes, los cuales desempeñan actividades como enseñar, aprender, limpiar o simplemente visitar, información que también tienen que registrar. Realizar el diagrama de clases que modele las acciones anteriores, utilizando correctamente las ***relaciones*** entre ellas.

CAPÍTULO 4

PROGRAMACIÓN ORIENTADA A OBJETOS

Introducción

De los diversos paradigmas de programación descritos en el capítulo anterior, el paradigma orientado a objetos es el que ha cobrado más adeptos entre los desarrolladores de sistemas en los últimos años. En este capítulo se dará una introducción a la programación orientada a objetos, se diseñarán diagramas y jerarquías de clases, y se elaborarán programas que irán de lo sencillo a lo complejo utilizando el lenguaje de programación Java. Por ello se estudiarán, las características propias de este lenguaje orientado a objetos, la estructura básica de un programa, el manejo de excepciones, los tipos de clase, y las interfaces.

4.1 Teoría del Diseño de Jerarquía de Clases.

Árbol

Un concepto muy útil en la programación, sobre todo en la programación orientada a objetos (POO), es el árbol. Un árbol es la representación ordenada de información, y consta de nodos, de los cuales salen ramificaciones a otros niveles, un ejemplo de un árbol se muestra en la figura 4.1

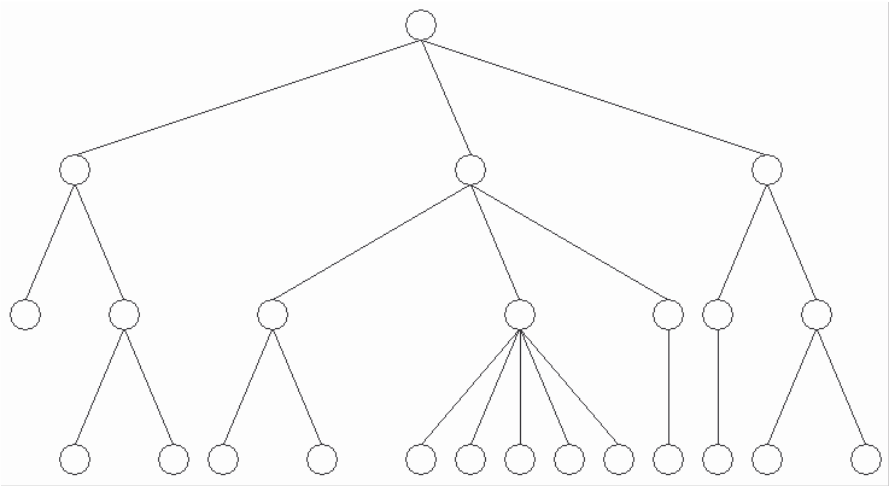


FIGURA 4.1 Ejemplo de un árbol

En la figura 4.1 cada círculo representa un nodo, y cada línea representa la relación que existe entre nodos. Es importante notar, que pueden o no existir ramificaciones en un nodo, y que cada nodo proviene de una sola ramificación del nivel superior.

Diseño de jerarquía de clases

Uno de los mayores retos a los que se enfrenta un programador cuando tiene que elaborar un sistema que resuelva un determinado problema, es que éste deba ser eficiente y libre de errores, lo que se llama comúnmente un sistema robusto. Para lograr este reto, el programador debe seguir una serie de técnicas, metodologías, estándares y procedimientos, que le permita crear código que cumpla con los requerimientos exigidos.

Un programador experimentado cuenta con códigos que han sido probados y que resuelven problemas muy específicos, ya sea que los haya elaborado él mismo u otros programadores. Estos códigos, que pueden estar incluidos en programas, funciones, métodos o clases, se pueden emplear en la realización de programas más grandes. La ventaja de *reutilizar* código probado radica en que se obtendrá un sistema robusto.

Uno de los objetivos del paradigma orientado a objetos es, precisamente, la reutilización de código que ha sido probado. Comúnmente este código se encuentra en clases o en métodos de clases por lo que debe existir una interrelación de clases.

Por esta razón, cuando se documenta el desarrollo de un software, se debe contar con un buen diagrama de clases que permita comprender correctamente la relación que existe entre las clases involucradas en la resolución del problema. Por esto, utilizando el esquema de árbol descrito anteriormente, el diseño de jerarquía de clases muestra la organización de todas las clases a utilizar en la solución del problema, percibiendo además, de una manera clara, la herencia de clases. De esta forma, se puede tener un buen diagrama de clases y por lo mismo un programa óptimo.

Para elaborar una jerarquía de clases, se deben seguir cuatro pasos, los cuales permitirán identificar todas aquellas clases que se irán agrupando y de esta forma obtener el árbol correspondiente.

a) Análisis del problema

Dado que el enunciado del problema es lo único que se tiene, éste se debe analizar para obtener el tipo de información que se manipulará; no es información como dato, sino qué clases se identifican para la resolución del problema. De esta forma se deberá generar una lista con dichas clases incluyendo una breve descripción de cada una. Si el problema es demasiado complejo, debido a que maneja muchas formas de información, se deberán incluir ejemplos.

b) Diagramas de clases (definición de datos)

La clave de este paso es trasladar la información obtenida en el paso anterior en un diagrama de clases, definiendo atributos y métodos, así como las relaciones que existen entre las clases.

Para determinar las relaciones de herencia que existen entre las clases, es recomendable seguir una serie de pasos iterativos, los cuales permitirán construir de manera adecuada una jerarquía de clases. Estos pasos son:

1. *Análisis*: Analizar todas las clases que provengan de un mismo padre. (Inicialmente todas las clases pertenecen a un mismo padre)
2. *Agrupación*: De acuerdo a características compartidas por las clases, dividir las clases en grupos.
3. *Nueva Clase*: Establecer una nueva clase que contenga dichas características. Ésta corresponderá a la clase padre y las clases del mismo grupo serán las clases hijas.

Se debe tener en cuenta que en una jerarquía de clases, entre más abajo en el árbol se encuentren las clases, éstas son más específicas; por el contrario, si las clases se encuentran en los niveles superiores, éstas son más generales o abstractas.

c) Definiciones de clase y declaraciones de propósito

En este paso se debe transformar el diagrama de clases en código, es decir generar los programas de cada clase, indicando dentro del programa la relación de herencia. Esto se verá más adelante en el capítulo.

Además de generar el código correspondiente, se debe incluir, en comentario, el propósito de la clase que se está escribiendo, de ser posible en una sola línea; si el problema es complejo se puede escribir en más líneas.

d) Ejemplos (representación e interpretación)

Una vez completados los pasos anteriores, ahora se utilizan los ejemplos propuestos en el análisis del problema, para verificar si es posible representar dichos ejemplos por medio de objetos creados por las clases del problema.

Jerarquía en Java

El lenguaje de programación Java tiene una jerarquía de clases predefinida, ya que en sí es la base del lenguaje. En esta jerarquía están incluidas las clases que definen los elementos básicos del lenguaje, como son los tipos de datos, las operaciones matemáticas y las operaciones de entrada y salida, entre otros.

Todas las clases del lenguaje de programación Java, provienen de la clase *Object*. Un ejemplo de una ramificación de esta clase se muestra en la figura 4.2, la cual, en realidad, es un subárbol del árbol completo de la jerarquía de clases predefinida en Java. Se puede

observar, que en este subárbol, las clases que están más abajo son más específicas, mientras que las superiores, son más abstractas. Las clases de este subárbol se agrupan en un *paquete* llamado *java.math*. El concepto de *paquete* en Java así como su manipulación se detallarán más adelante en este capítulo.

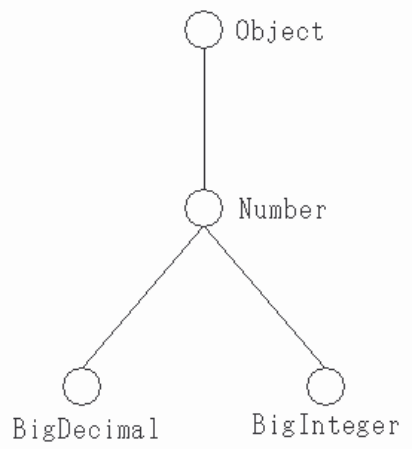


FIGURA 4.2 Jerarquía de clases en Java del paquete *java.math*.

EJEMPLO 4.1 CONSTRUIR UN DIAGRAMA DE JERARQUÍA DE CLASES QUE CONTENGA LOS ARTÍCULOS LIBRO Y REVISTA.

-Análisis y solución del problema

Del enunciado del problema, se identifican, en primera instancia, las clases Libro y Revista.
Descripción breve de cada clase:
Libro: artículo de lectura conformado por hojas de papel impresas y reunidas en un volumen encuadernado.
Revista: artículo de lectura publicado generalmente por periodos en donde se presentan escritos de varios autores.

-Diagrama de clases

Definición de atributos y métodos para cada clase.

A la clase Libro se le definen los siguientes atributos:
autor, titulo, fechaDePublicacion, editorial, edicion.

A la clase Revista se le definen los siguientes atributos:
titulo, editorial, fechaDePublicacion, numeroDePublicacion.

Por la sencillez del enunciado no se definen métodos a ninguna de las clases.

Considerando esta información, se realizará la serie de pasos iterativos para la creación de la jerarquía de clases solicitada.

1. *Análisis*

Se puede observar que las clases Libro y Revista tienen atributos en común, los cuales son: titulo, editorial y fechaDePublicacion

2. *Agrupación*

Estas dos clases se pueden agrupar en una sola por los atributos en común.

3. *Nueva Clase*

Se puede generar una clase superior o padre, llamada Publicacion, de la cual se derivarán las clases Libro y Revista, obteniendo un árbol como se muestra en la figura 4.3.

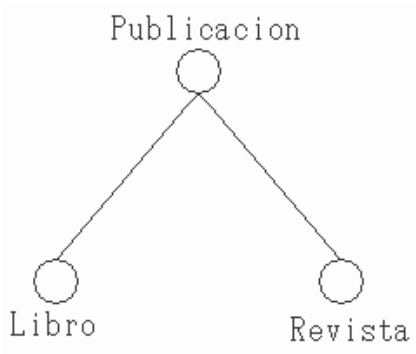


FIGURA 4.3 Jerarquía de clases del ejemplo 4.1.

Como después de esta agrupación ya no se puede hacer ninguna reagrupación al repetir los 3 pasos, se detiene la iteración.

Por lo tanto, los atributos de Publicacion serán: titulo, editorial y fechaDePublicacion; mientras que los atributos de Libro serán: autor y edicion; y el de Revista será: numeroDePublicacion.

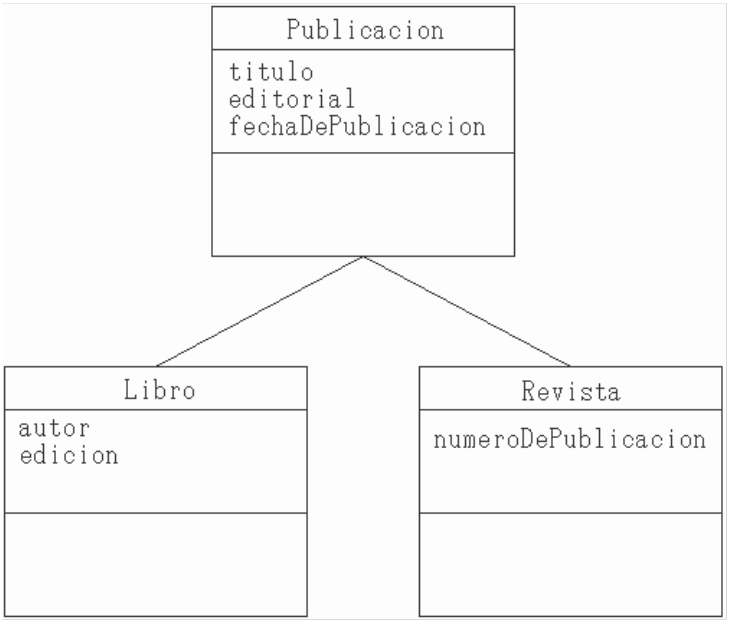


FIGURA 4.4 Diagrama de clase del ejemplo 4.1.

Lenguajes orientados a objetos

Una vez que se tiene listo el diagrama de clases, el siguiente paso es transformarlo a código, así el programa empieza a tomar forma. Antes de abordar las características de un programa en lenguaje orientado a objetos, es conveniente mencionar algunas reglas que se han adoptado por convención y que deberán aplicarse al momento de escribir un programa.

Nomenclatura de los lenguajes orientados a objetos

Hay ciertas reglas que todo programador debe seguir, de tal forma que cuando alguien externo observe el código sepa qué se está haciendo. Por esta razón, cuando se escriben programas en lenguajes orientados a objetos, hay una nomenclatura que debe utilizarse al momento de escribir los nombres de las clases, de los atributos y de los métodos. Estas reglas son:

1. No debe existir ningún espacio en blanco, ni guiones bajos dentro de los nombres.

2. En el caso de las clases, la primera letra de cada palabra que conforma el nombre, deberá ser mayúscula, por ejemplo: Libro, CuentaDeAhorros, MaquinaDeRefrescos, etc.

3. Para los nombres de atributos, las letras de todo el nombre deberán ser minúsculas; en el caso de que incluya varias palabras, a partir de la segunda deberá iniciar con mayúscula; por ejemplo: color, textura, parteReal, parteImaginaria, distanciaEntreDosPuntos, etc.

4. Los nombres de métodos, siguen la misma nomenclatura que con los atributos, la diferencia es que los métodos llevan un par de paréntesis después del nombre, por ejemplo: correr( ), sumarDosVectores( ), calcularInteres( ), etc.

Introducción al lenguaje Java

El lenguaje de programación Java surge de la necesidad de contar con un leguaje capaz de trabajar en la red de redes Internet de una forma transparente para el usuario. Debido a que los sistemas que conforman esta mega-red operan bajo distintos sistemas operativos, los programas escritos en este nuevo lenguaje debían poder ejecutarse en cualquiera de estos sistemas.

Sun Microsystems es la empresa creadora de Java que se basa en algunos desarrollos que fue fortaleciendo; finalmente lo presenta en agosto de 1995. Java puede interactuar con varias APIs (Application Programming Interface) que son conjuntos de convenciones de programación que definen cómo se invoca un servicio desde un programa; por lo que Java resulta ser una herramienta muy completa. Algunas APIs de la plataforma Java son: Java Enterprise, JDBC API, Java RMI, Java IDL, JavaBeans, Java Servlet API, Java Speech, JME, Java Smart Card y Embedded Java.

No es intención de este apartado describir a detalle al lenguaje Java ya que sólo se utilizará como herramienta de implementación de programas orientados a objetos y no como un elemento de estudio exhaustivo.

La estructura base para escribir un programa en Java es:

```
class NombreDeClase{
   tipo_dato atributoUno;
   tipo_dato atributoDos;

   public static void main(String args[]){
   }

   tipo_dato metodoUno(tipo_dato argUno,…){
   }
   tipo_dato metodoDos(tipo_dato argUno,…){
   }
}
```

Como se puede observar, un programa escrito en Java, define una clase. La definición de una clase está dada por la palabra reservada ***class*** y el nombre de la clase; le sigue el cuerpo de la clase delimitada por { }. En el cuerpo de la clase se deben declarar todos los atributos

indicando su tipo de dato, es decir *int*, *float*, *double*, etc; así como declarar los métodos que se utilizarán, indicando, de igual forma, el tipo de dato que devuelve como resultado de su ejecución; si el método no devuelve nada, se deberá poner la palabra *void*.

Cuando la clase servirá como punto de entrada al sistema, se deberá definir un método *main*, el cual siempre se define como:

```
public static void main(String args[]){
}
```

Si existe una relación de herencia con respecto a otra clase, se deberá declarar al momento de definirla; esto se hace con la palabra *extends* seguido del nombre de la clase padre.

```
class NombreClase extends NombreClasePadre{
}
```

EJEMPLO 4.2 ESCRIBIR UN PROGRAMA EN JAVA QUE IMPRIMA EN PANTALLA "HOLA MUNDO".

-Análisis y solución del problema

Se requiere una clase llamada HolaMundo, y debido a que su única función es imprimir un mensaje en pantalla, no es necesario definir atributos, sólo el método *main*.

-Diagrama de clase

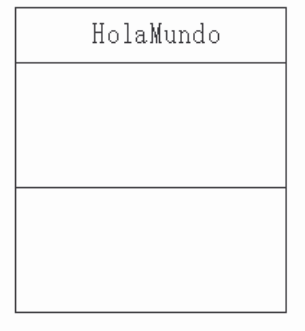


FIGURA 4.5 Diagrama de clase del ejemplo 4.2.

-Código

```
class HolaMundo{
   public static void main(String args[]){
      System.out.println("Hola Mundo");
   }
}
```

-Explicación del código

Se puede observar que este programa inicia definiendo una clase. Esto ocurre siempre que se elabora un programa escrito en Java. En este caso, se definió una clase que se llama

HolaMundo:

```
class HolaMundo{
}
```

Como sólo contendrá el método *main*, éste se define siempre de la siguiente forma:

```
public static void main(String args[]){
}
```

Un equivalente al *printf( )* del lenguaje C, en Java es:

```
System.out.println("Hola Mundo");
```

Donde `System.out` indica que la salida será en pantalla y el método `println()`, recibe lo que se imprimirá.

Compilación y ejecución de un programa en lenguaje Java

Debido a que Java puede trabajar en diferentes plataformas como Windows, Linux, MacOS, Solaris, etc., cuenta con una *máquina virtual* (JVM, por sus siglas en inglés); esto hace que para ejecutar un programa escrito en Java, sea necesario realizar dos procesos. El primero consiste en compilar el programa generando un archivo con la extensión *.class* escrito en ByteCode. Este código es un código intermedio entre el lenguaje de máquina del procesador y Java.

En el segundo proceso, la Máquina Virtual de Java instalada en cada equipo donde se ejecutará el programa, interpreta el ByteCode del archivo *.class* generando el código en lenguaje de máquina del procesador local y ejecutándolo al mismo tiempo.

Es así que la secuencia de pasos para compilar y ejecutar un programa escrito en Java es el siguiente:

- guardar el código fuente con el mismo nombre de la clase y con la extensión *.java*. Por ejemplo, si la clase es HolaMundo, el nombre del archivo deberá ser HolaMundo.java.

- abrir una ventana del sistema y situarse en la carpeta donde se encuentre el programa.

- compilar el programa con el comando *javac*. Por ejemplo, para compilar el programa HolaMundo.java, se debe escribir la siguiente instrucción:

>javac HolaMundo.java

si el programa no tiene errores, regresará el símbolo del sistema. Con esto se ha generado el archivo *HolaMundo.class*.

- ejecutar el programa con extensión *.class* usando la orden *java* de la siguiente forma:

>java HolaMundo

Cabe hacer notar que para ejecutar el programa, no es necesario escribir la extensión *.class*.

EJEMPLO 4.3 IMPLEMENTAR UNA CLASE QUE SUME DOS NÚMEROS ENTEROS.

-Análisis y solución del problema

Se debe declarar una clase llamada SumaDosEnteros, debe tener dos atributos de tipo *int* y un método que realice la suma.

-Diagrama de clase

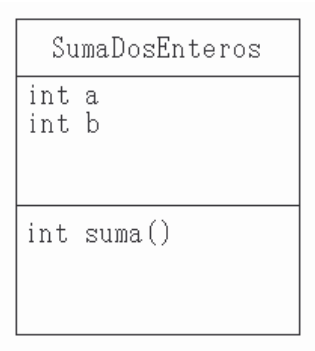


FIGURA 4.6 Diagrama de clase del ejemplo 4.3.

-Código

```
class SumaDosEnteros{
   int a,b;
   public static void main(String args[]){
      SumaDosEnteros sde = new SumaDosEnteros();
      sde.a=3;
      sde.b=6;
```

```
      System.out.println("La suma de "+sde.a+" + "+sde.b+
                         " es: "+sde.suma(sde.a,sde.b));
   }
   int suma(int x, int y){
      int z;
      z=x+y;
      return z;
   }
}
```

-Explicación del código

Con base en el diagrama de clase, la declaración de la clase SumaDosEnteros queda de la siguiente forma:

```
class SumaDosEnteros{
   int a,b;
   int suma(){
   }
}
```

Debido a que se debe realizar una suma, dentro del método suma( ) se implementa la suma, incluyendo la recepción de dos parámetros; resultando el método como:

```
int suma(int x, int y){
   int z;
   z=x+y;
   return z;
}
```

La clase así como se encuentra, puede realizar la suma de dos números enteros, pero la única forma de verificar esto es incluyendo un punto de entrada y salida, por lo cual se requiere de la función *main*.

```
public static void main(String args[]){
}
```

Para utilizar los atributos y métodos de la clase, se debe crear un objeto de la clase, esto se realiza de la siguiente forma:

```
SumaDosEnteros sde = new SumaDosEnteros();
```

Con la palabra *new* se crea un nuevo objeto; a esta palabra le sigue el nombre de la clase como método (por los paréntesis que le acompañan), éste es un método constructor, el cual se describirá más adelante en este capítulo. Hay que hacer notar que por nomenclatura, cuando se genera un objeto de cierta clase, es recomendable nombrarla considerando las

letras mayúsculas de la clase, escribiéndolas en minúsculas; por ejemplo, si la clase es SumaDosEnteros, sus mayúsculas son SDE, por lo tanto el objeto se puede llamar `sde`.

Una vez creado el objeto, se pueden asignar valores a sus atributos utilizando el nombre del objeto seguido del operador punto (.) y el nombre del atributo, en este caso:

```
sde.a=3;
sde.b=6;
```

Para invocar al método del objeto, al igual que con los atributos, se le antepone al nombre del método el nombre del objeto separado por el operador punto. Para este ejemplo, la invocación al método suma se hace de la siguiente forma:

```
sde.suma(sde.a,sde.b);
```

Ahora sólo falta desplegar en pantalla el resultado de la suma, por lo que se utilizará `System.out.println()`, de la siguiente forma:

```
System.out.println("La suma de "+sde.a+" + "+sde.b+" es: "
                   +sde.suma(sde.a,sde.b));
```

A diferencia de C, en Java cuando se manda imprimir información en pantalla utilizando la instrucción `System.out.println()`, no es necesario poner todo el texto entre comillas indicando los lugares donde irá información de variables, y al final poner esas variables en el orden. Al utilizar el operador "+", se concatenan dos cadenas, incluso el valor de una variable se toma como una cadena, por lo que no hay que especificar el formato. De esta forma la salida del programa es:

```
La suma de 3 + 6  es: 9
```

EJEMPLO 4.4 ELABORAR UN PROGRAMA EN JAVA QUE UTILICE LA CLASE SUMADOSENTEROS, PARA OBTENER LOS PRIMEROS OCHO ELEMENTOS DE LA SERIE DE FIBONACCI.

-Análisis y solución del problema

La serie de Fibonacci, es una serie de números enteros que tienen una relación entre sí. La serie comienza con un cero y un uno; para obtener el siguiente elemento, basta con sumar los 2 números anteriores; de esta forma, el tercer elemento de la serie es 0+1=1, el cuarto es 1+1=2, el quinto es 1+2=3, etc.

La serie de Fibonacci con los primeros 9 elementos se muestra a continuación:

0 1 1 2 3 5 8 13 21 …

Para la solución de este problema se identifican dos clases, la clase SumaDosEnteros definida en el ejemplo 4.3 y una clase que se propone como SerieFibonacci. La clase SerieFibonacci tendrá un atributo *temporal* el cual funcionará como enlace entre el cálculo del nuevo elemento y el cambio de valor en los atributos de SumaDosNumeros para lograr obtener la serie.

-Diagrama de clases

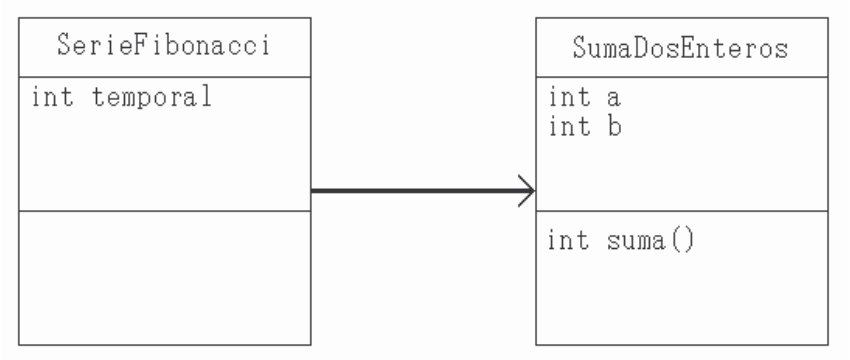


FIGURA 4.7 Diagrama de clase del ejemplo 4.4.

-Código

```
class SerieFibonacci{
   int temporal;
   public static void main(String args[]){
      SerieFibonacci sf = new SerieFibonacci();
      SumaDosEnteros sde = new SumaDosEnteros();
      sde.a=0;
      sde.b=1;
      System.out.print(sde.a+", "+sde.b);
      sf.temporal=sde.suma(sde.a,sde.b);
      sde.a=sde.b;
      sde.b=sf.temporal;
      System.out.print(", "+sde.b);
      sf.temporal=sde.suma(sde.a,sde.b);
      sde.a=sde.b;
      sde.b=sf.temporal;
      System.out.print(", "+sde.b);
      sf.temporal=sde.suma(sde.a,sde.b);
      sde.a=sde.b;
      sde.b=sf.temporal;
      System.out.print(", "+sde.b);
      sf.temporal=sde.suma(sde.a,sde.b);
      sde.a=sde.b;
      sde.b=sf.temporal;
      System.out.print(", "+sde.b);
```

```
      sf.temporal=sde.suma(sde.a,sde.b);
      sde.a=sde.b;
      sde.b=sf.temporal;
      System.out.print(", "+sde.b);
      sf.temporal=sde.suma(sde.a,sde.b);
      sde.a=sde.b;
      sde.b=sf.temporal;
      System.out.print(", "+sde.b);
   }
}
```

-Explicación del código

Del diagrama de clases, se escribe la base para la clase SerieFibonacci, es decir:

```
class SerieFibonacci{
   int temporal;
}
```

Ahora se incluye el método *main*:

```
class SerieFibonacci{
   int temporal;
   public static void main(String args[]){
   }
}
```

Se crean 2 objetos, uno de SerieFibonacci y otro de SumaDosEnteros

```
class SerieFibonacci{
   int temporal;
   public static void main(String args[]){
      SerieFibonacci sf = new SerieFibonacci();
      SumaDosEnteros sde = new SumaDosEnteros();
   }
}
```

NOTA: Cuando se utiliza una clase dentro de otra sin tener relación de herencia, se debe tener cuidado de tener dicha clase en el mismo directorio de la clase que la utiliza.

Ahora se asigna un valor inicial a *a* y a *b*, es decir 0 y 1, respectivamente debido a que son los primeros elementos de la serie; posteriormente se mandan a imprimir.

```
sde.a=0;
sde.b=1;
System.out.print(sde.a+", "+sde.b);
```

Luego se sigue el procedimiento para obtener los valores, es decir en `temporal` se almacena el nuevo valor, mandando como parámetros los valores *a* y *b* al método suma(), después de obtener el nuevo valor, el valor *b* se pasa a *a* y el valor de *temporal* a *b*, para luego mandarlo a imprimir. En este caso se está utilizando el método `print()` en lugar de `println()`, para evitar los saltos de línea con cada valor calculado.

```
sf.temporal=sde.suma(sde.a,sde.b);
sde.a=sde.b;
sde.b=sf.temporal;
System.out.print(", "+sde.b);
```

Este procedimiento se repite 6 veces para completar los 8 elementos

4.2 Control de flujo

Como se puede observar en el ejemplo 4.4, un segmento de código se repite varias veces; para estos casos es más eficiente utilizar una sentencia de repetición. Las sentencias de control de flujo, donde están incluidas las sentencias de repetición, permiten ejecutar ciclos o tomar decisiones de acuerdo a los pasos que llevan a la solución del problema. Estas sentencias de control utilizan expresiones que permiten tomar la decisión de continuar el ciclo o realizar cierta acción.

Expresiones Booleanas

Las expresiones de control de flujo que utilizan operadores relacionales o lógicos, al ser evaluadas, devuelven un valor booleano (falso o verdadero); es por esto que se les llaman expresiones booleanas.
La siguiente tabla muestra los operadores relacionales en Java:

Expresión	Descripción
a = = b	*a* tiene el mismo valor que *b*
a < b	*a* es menor que *b*
a > b	*a* es mayor que *b*
a <= b	*a* es menor o igual que *b*
a >= b	*a* es mayor o igual que *b*
a != b	*a* es diferente que *b*

Los operadores lógicos pueden formar expresiones más complejas ya que une expresiones que contienen operadores relacionales. Los operadores lógicos o condicionales en Java son:

Expresión	Descripción
&& o &	Operación AND, el resultado de esta operación es verdadero, siempre y cuando ambas expresiones sean verdaderas, cuando alguna de ellas es falsa, el resultado es falso. La diferencia entre estos 2 operadores AND, es que cuando se utiliza &&, si la condición 1 es falsa, no se evalúa la condición 2.
|| o |	Operación OR, el resultado de esta operación es verdadera, cuando alguna de las condiciones es verdadera, cuando ambas son falsas el resultado es falso. La diferencia entre estos 2 operadores OR, es que cuando se utiliza ||, si la condición 1 es verdadera, no se evalúa la condición 2.
!	Operación NOT, el objetivo de este operador es negar el valor de la expresión, es decir, el resultado de la operación es falso si la evaluación de la condición es verdadera; y el resultado es verdadero si la evaluación de la condición es falsa.
^	Operación XOR, el resultado de esta operación es verdadera, si una de las condiciones es verdadera y la otra falsa; cuando ambas son verdaderas o falsas, el resultado es falso.

Otro caso de expresión booleana es el de invocar una función cuyo valor devuelto sea un boolean, en cuyo caso este valor se utiliza para la toma de decisiones.

4.2.1 Sentencia IF-ELSE

Esta sentencia se utiliza cuando se requiere determinar la ejecución de distintos fragmentos de código de acuerdo al resultado de la evaluación de una expresión booleana.

La sintaxis utilizada en esta sentencia es:

```
if (expresión)
    proposición1
else
    proposición2
```

Donde `expresión,` es la que da la pauta en la toma de decisión. Esto es, si el resultado de evaluar la `expresión` es verdadero, ejecuta la `proposición1`, en caso contrario

ejecuta la `proposición2`. Una proposición puede ser una sentencia simple o un bloque de instrucciones.

Esta sentencia también puede ser empleada sin el caso `else`, en donde se evalúa la expresión booleana y sólo si es verdadera ejecuta la `proposición1`. La `proposición2` siempre se ejecutará, esto es, independientemente del valor que tenga la expresión booleana. La sintaxis de esta variante es la siguiente:

```
if (expresión)
    proposición1
proposición2
```

Por ejemplo, del siguiente código:

```
if(x < 0)
   b=-x;
b=b+1;
```

Si la expresión $(x < 0)$ es verdadera, es decir que *x* sea menor a cero, se ejecutará $b = -x;$ y continúa ejecutando $b = b + 1;$. Pero, si el resultado de la condición es falso, omite $b = -x;$ y continúa con la instrucción $b = b + 1;$. Como se puede observar, sea cual sea el valor de la expresión booleana, siempre se ejecuta la instrucción $b = b + 1;$.

Cuando se deben ejecutar varias instrucciones, éstas deberán ir entre llaves, por ejemplo:

```
if((a-b)>c){
   c=2*c;
   a=(2*b)/c;
   b=5*a;
}
else{
   b=c/a;
   a=c+b;
}
```

El siguiente código, aunque pareciera todo lo contrario, tiene una sola sentencia dentro de la sentencia *if* más externa, por lo que no requiere el uso de llaves.

```
if(a>3)
   if(b<a)
      c=2;
   else
      c=1;
else
   c=0;
```

Sentencias IF-ELSE anidadas

Las sentencias IF-ELSE pueden estar anidadas, es decir, se puede escribir una sentencia *if* como *proposición1* o *proposición2*, dentro de una sentencia *if* externa, de acuerdo al formato que se ha manejado.

Esto es:

```
if(expresión1)
{
   if(expresión2)
      proposición1
}
else
   proposición2
```

Al llevarse a cabo las sentencias anteriores, se pueden presentar los siguientes casos:

SI	expresión1	expresión 2	∴	proposición1	proposición2
	Falsa	Falsa		No se ejecuta	Sí se ejecuta
	Falsa	Verdadera		No se ejecuta	Sí se ejecuta
	Verdadera	Falsa		No se ejecuta	No se ejecuta
	Verdadera	Verdadera		Sí se ejecuta	No se ejecuta

Cuando se emplea el anidamiento de sentencias *if*, es importante definir correctamente las llaves que limitarán a la proposición1 o a la proposición2, ya que podrían suscitarse confusiones. Por ejemplo, si en el caso anterior se eliminaran las llaves, el código resultante sería:

```
if(expresión1)
   if(expresión2)
      proposición1
else
   proposición2
```

Cuando aparecen varias sentencias if... else en el código, se lleva a cabo la siguiente regla *"cada else se corresponde con el if más próximo que no haya sido emparejado"*, por lo tanto, en el código anterior, el *else* está emparejado con el segundo *if*, cuestión que afectaría notablemente la salida del programa. Para el ejemplo los resultados variarían de la siguiente manera:

SI	expresión1	expresión2	∴	proposición1	proposición2
	falsa	Falsa		No se ejecuta	No se ejecuta
	Falsa	Verdadera		No se ejecuta	No se ejecuta
	Verdadera	Falsa		No se ejecuta	Sí se ejecuta
	verdadera	verdadera		Sí se ejecuta	No se ejecuta

Otro tipo de anidamiento es la sentencia IF-ELSE IF, y se muestra en el siguiente ejemplo:

```
if(a>b)
   System.out.println(a+ " es mayor que  "+b);
else if(a<b)
   System.out.println(a+ " es menor que " +b);
else
   System.out.println(a+ " es igual que " +b);
//continuación del programa
```

En este caso, al ir evaluando las expresiones que indican la condición a cumplir, en el momento en que una se cumpla, ejecuta la proposición correspondiente y se brinca todas las demás expresiones que no se hayan evaluado.

4.2.2 Sentencia SWITCH

Esta sentencia es utilizada cuando se debe ejecutar un fragmento de código de acuerdo con el valor de la ***expresión***, la cual puede devolver cualquier tipo básico. La sintaxis de la sentencia *switch* se muestra a continuación.

```
switch (expresión){
   case expresión-cte1:
       proposición1
   case expresión-cte2:
       proposición2
   case expresión-cte3:
       proposición3
⋮
   default:
       proposición_n
}
```

La expresión de decisión de la sentencia *switch* debe ser una expresión de tipo *char, byte, short o int*. La expresión-cte debe ser de cualquiera de los tipos mencionados anteriormente. Las expresiones constantes así como la expresión de decisión se convierten implícitamente a *int*.

La forma en que opera esta sentencia es comparar el resultado de la expresión con los valores de cada *case* (caso). La sentencia *switch* puede contener tantos *case* como sean necesarios. En caso de no haber una coincidencia, se sigue hasta el *default* (si está especificado) y ejecuta la proposición contenida para ese caso. Hay que hacer notar que cada proposición puede contener una o más instrucciones, pero para finalizar la ejecución del caso, siempre se deberá escribir un *break* ya que esto evita la ejecución de las proposiciones de los siguientes casos.

En una sentencia *switch* es posible hacer declaraciones en cada caso, pero no es posible hacer declaraciones antes del primer *caso*.

En el siguiente ejemplo se muestra el uso de la sentencia *switch*, donde en cada caso se tiene una proposición y un break.

```
switch(mes){
   case 1:
      System.out.println("Enero");
      break;
   case 2:
      System.out.println("Febrero");
      break;
   case 3:
      System.out.println("Marzo");
      break;
   case 4:
      System.out.println("Abril");
      break;
   case 5:
      System.out.println("Mayo");
      break;
   case 6:
      System.out.println("Junio");
      break;
   case 7:
      System.out.println("Julio");
      break;
   case 8:
      System.out.println("Agosto");
      break;
   case 9:
      System.out.println("Septiembre");
      break;
   case 10:
      System.out.println("Octubre");
      break;
   case 11:
      System.out.println("Noviembre");
      break;
   case 12:
      System.out.println("Diciembre");
      break;
   default:
      System.out.println("Mes no válido");
}
```

Como se puede observar, en función del valor que tenga la variable *mes*, se ejecuta una instrucción de acuerdo al *case* correspondiente, pero independientemente del *case* al que acceda, se debe escribir la sentencia *break*, de lo contrario se ejecutan todas las proposiciones a partir del caso por donde comienza. Por esta razón, se puede omitir la sentencia *break* en el *default*.

Cuando se requiere ejecutar una misma instrucción para varios casos diferentes, se ponen los casos en secuencia, como se muestra a continuación:

```
switch(mes){
   case 12: // Continúa
   case 1:  // Continúa
   case 2:
      System.out.println("Estación: Invierno");
      break;
   case 3:  // Continúa
   case 4:  // Continúa
   case 5:
      System.out.println("Estación: Primavera");
      break;
   case 6:  // Continúa
   case 7:  // Continúa
   case 8:
      System.out.println("Estación: Verano");
      break;
   case 9:  // Continúa
   case 10: // Continúa
   case 11:
      System.out.println("Estación: Otoño");
      break;
   default:
      System.out.println("Mes no válido");
}
```

4.2.3 Sentencia FOR

Esta sentencia es utilizada cuando se debe ejecutar un fragmento de código un número definido de veces. La sintaxis de esta sentencia de control de flujo es:

```
for (v1=ci1 , v2=ci2,…; expresión ; progreso-condición)
   proposición
```

donde:
`v1, v2`, …, representan variables de control cuya valor inicial está dada por `ci1, ci2`, …; `expresión` es aquella condición que deberá ser verdadera, para ejecutar el ciclo, en el

momento en que sea falsa, termina la ejecución. `progreso-condición` es una o más expresiones separadas por comas, cuyos valores evolucionan conforme se cumpla la condición de la sentencia `for`.

El funcionamiento de esta sentencia se puede observar en el siguiente fragmento de código.

```
for(int i=1; i<=10; i++)
    System.out.println(i+ "  ");
```

Como se observa en este código, se puede hacer una declaración de variable dentro del ciclo, y no es necesario declararla al inicio del programa. Este tipo de declaración en ciclos se le llama ***instanciación,*** la cual crea una variable que existirá mientras se ejecute el ciclo, una vez terminado, la variable se "destruye". Después de declarar la variable llamada `i` para la ejecución del ciclo `for`, ésta se inicializa con un valor de 1, luego la ***expresión*** (`i>=10`) se evalúa, y mientras sea verdadera, el ciclo se repetirá. La forma en que se modifica el valor de la variable de control `i`, es que se incrementa en uno cada vez que termina un ciclo. En este caso la instrucción que se ejecuta es imprimir en pantalla los número del 1 al 10.

4.2.4 Sentencia WHILE

Esta sentencia es utilizada cuando se debe ejecutar un fragmento de código un número definido de veces. A diferencia de la sentencia ***for***, esta sentencia evalúa una expresión que no necesariamente involucra una variable de control definida para el ciclo. La sintaxis de esta sentencia de control de flujo es:

```
while (expresión)
   proposición
```

Donde se evalúa la `expresión` y si ésta es verdadera, se ejecuta la `proposición` y vuelve a evaluar la `expresión`; este proceso se realiza mientras la `expresión` sea verdadera, en el momento en que ésta sea falsa se sale del ciclo.

Por ejemplo, para mandar a imprimir los números del 1 al 10 utilizando la sentencia de control ***while***, se hace de la siguiente manera:

```
int i=1;
while(i<=10){
   System.out.println(i+" ");
   i++;
}
```

En este código se declara una variable ***i*** asignándole un valor inicial de 1; posteriormente en la sentencia ***while***, se evalúa la expresión, y mientras sea verdadera, es decir que ***i*** sea menor o igual que 10, imprime en pantalla el valor de ***i*** y la incrementa en uno.

Sentencia DO-WHILE

La estructura *while* siempre condiciona la ejecución de la proposición, pero si se desea ejecutar la proposición por la menos una vez antes de evaluar la expresión, se utiliza la estructura *do-while*. Su sintaxis es:

```
do
  proposición
while (expresión);
```

Hay que hacer notar que esta sentencia siempre finaliza con un punto y coma (;) después de la expresión booleana del *while*.

Por ejemplo, cuando se desea recibir un parámetro desde el teclado, pero con la condición de que éste debe ser positivo:

```
double n;
do // ejecutar las sentencias siguientes
{
   System.out,println ("numero: ");
   n=Leer.datodouble();
}
while (n<0); //mientras n sea menor que 0
```

En el código anterior se define una variable *double* que almacenará el valor leído, posteriormente inicia la sentencia de control *do-while*, donde la primera instrucción a ejecutarse es la de imprimir el texto `numero:` y a continuación se obtiene el valor a través del método `Leer.datodouble( )`; después se evalúa la expresión `(n<0)`, si se cumple que *n* sea menor que cero, se seguirá ejecutando el ciclo hasta que se de un número mayor que cero.

Como se puede observar, esta estructura ejecuta por lo menos una vez la proposición (simple o compuesta) antes de evaluar la expresión.

Sentencia BREAK

Como se mencionó en la sentencia *switch*, se emplea la sentencia *break* para limitar el fragmento de código a ejecutar para cada caso. Otra función de la sentencia *break* es romper un ciclo definido por cualquiera de las sentencias *while*, *do* o *for*.

Por ejemplo, del siguiente código:

```
while(x>2){
   x=x/y;
   z++;
   if(z>100)
      break;
}
```

la sentencia ***break*** obliga a salir del ciclo ***while*** cuando se cumpla la expresión contenida en el ***if***. Es decir, el ciclo divide la variable ***x*** entre la variable ***y*** mientras ***x*** sea mayor que dos; por lo que si se tiene una variable de control ***z***, y se observa que el ciclo de repetición se ejecuta más de 100 veces lo termina.

Sentencia CONTINUE

Así como la sentencia ***break*** obliga a salir de un ciclo de repetición, existe otra sentencia que realiza lo contrario ya que obliga a que lo "continúe" pero a partir de la siguiente iteración.

Por ejemplo, el siguiente código:

```
for(int x = 0;x<150; x++){
   if(x%11 != 0)
      continue;
   System.out.println(x+" es multiplo de 11");
}
```

imprime en pantalla todos aquellos números múltiplos de 11 que se encuentran entre 0 y 150; esto se realiza obteniendo el residuo de dividir `x` entre 11 (`x%11`) y comparándolo con `0;` si hay residuo, se ejecuta la sentencia `continue`, que obliga a omitir todas las instrucciones que están después de esta sentencia, que en este caso es la instrucción `System.out.println(x+" es multiplo de 11");`, y comienza la siguiente iteración.

EJEMPLO 4.5 ESCRIBIR UN PROGRAMA QUE IMPLEMENTE EL MÉTODO DE BISECCIÓN Y RESUELVA LA ECUACION DEL EJEMPLO 2.1.

- Análisis y solución del problema

Se identifica una clase llamada Biseccion, que tendrá como atributos a, b, c, y tolerancia, así como un atributo que llevará el conteo de las iteraciones. Así mismo se identifican dos métodos: uno que implemente la función a resolver, y otro que implemente el método de bisección.

-Diagrama de clase

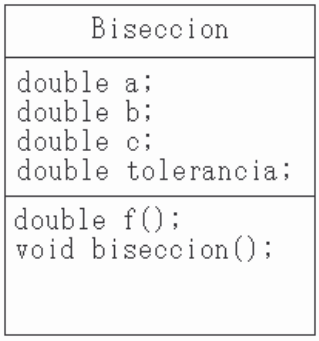


FIGURA 4.8 Diagrama de clase del ejemplo 4.5.

-Código

```
class Biseccion{
   double a,b,c,tolerancia;
   int i;
   double f(double x){
      double res;
      res=Math.pow(x,4)+5.8*Math.pow(x,3)-22.4*Math.pow(x,2)-
          31.2*x+57.6;
      return res;
   }
   void biseccion(){
      do{
         this.c=(this.a+this.b)/2;
         if((this.f(this.a)*this.f(this.c))<0)
            this.b=this.c;
         else
            this.a=this.c;
         this.i++;
      }while(Math.abs(this.b-this.a)>this.tolerancia);
   }
   public static void main(String args[]){
      Biseccion bs = new Biseccion();
      bs.tolerancia=0.001;
      bs.a=1;
      bs.b=2;
      bs.i=0;
      bs.biseccion();
      System.out.println("El valor de la raíz es: "+bs.c+
                         " en la iteración: "+bs.i);
   }
}
```

-Explicación del código

Del diagrama de clase se obtiene la base del código

```
class Biseccion{
   double a,b,c,tolerancia;
   int i;

   double f(double x){
   }

   void biseccion(){
   }
}
```

Se implementa el polinomio en el método `f`, utilizando el método `pow` de la clase `Math`.

```
double f(double x){
   double res;
   res=Math.pow(x,4)+5.8*Math.pow(x,3)-22.4*Math.pow(x,2)-
          31.2*x+57.6;
   return res;
}
```

Se implementa el algoritmo de bisección.

```
void biseccion(){
   do{
      this.c=(this.a+this.b)/2;
      if((this.f(this.a)*this.f(this.c))<0)
         this.b=this.c;
      else
         this.a=this.c;
      this.i++;
   }while(Math.abs(this.b-this.a)>this.tolerancia);
}
```

Como se puede observar, se ha introducido un nuevo término, la palabra ***this***, la cual permite utilizar dentro de los métodos, los atributos del objeto que mandó llamar al método.

Se agrega un método *main*, para asignar las condiciones iniciales del problema.

```
public static void main(String args[]){
   Biseccion bs = new Biseccion();
   bs.tolerancia=0.001;
   bs.a=1;
   bs.b=2;
   bs.i=0;
   bs.biseccion();
   System.out.println("El valor de la raíz es: "+bs.c+
                      " en la iteracion: "+bs.i);
}
```

EJEMPLO 4.6 ELABORAR UN PROGRAMA EN JAVA QUE CALCULE LOS PRIMEROS *N* NÚMEROS DE LA SERIE DE FIBONACCI.

- Análisis y solución del problema

Los números de la serie de Fibonacci se generan sumando los últimos dos números ya obtenidos en la serie, de esta forma se puede pensar en definir dos clases, la primera

realizará la suma de dos números nombrada SumaDosNumeros, y la otra desplegará en pantalla la serie, la cual se llamará Fibonacci.

Como se van a emplear dos clases es necesario describir cuál será la relación entre ellas, por lo que se elaborará su diagrama de clases empleando la teoría del diseño de jerarquía de clases.

-Diagrama de clases

Definición de atributos y métodos para cada clase.

La clase SumaDosNumeros, manejará dos variables (nombradas `a` y `b`) que harán las veces de sumandos, y una variable más que almacenará la suma de dichos números (nombrada `suma`); con la finalidad de que se puedan manipular valores muy grandes, estas tres variables se definirán de tipo ***double***. Esta clase además tendrá un método que realizará la suma.

La clase Fibonacci, manejará cuatro variables, tres para realizar la suma (`a, b` y `suma`), y la cuarta que indicará el número de elementos a calcular (`n`). Así mismo contendrá un método que implementará la impresión de la serie en pantalla, evaluando si el valor de `n` dado es válido o no.

1. *Análisis*

Se puede observar que las clases SumaDosNumeros y Fibonacci tienen atributos en común, los cuales son: `a, b` y `suma`.

2. *Agrupación*

La relación que existe entre estas dos clases es la de clase-subclase, es decir una clase hereda a otra clase; la clase o superclase es SumaDosNumeros y la subclase es Fibonacci, esto es debido a que la clase Fibonacci incluye todos los atributos de la superclase. El atributo `n` estará definido en la clase Fibonacci.

Finalmente el diagrama de clases de este ejemplo se muestra en la figura 4.9

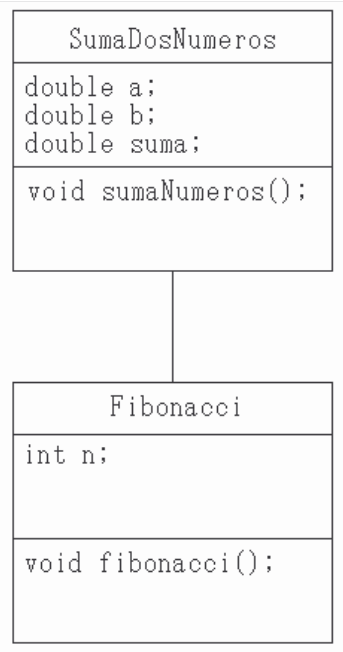


FIGURA 4.9 Diagrama de clase del ejemplo 4.6.

-Código clase SumaDosNumeros

```
class SumaDosNumeros{
   double a,b,suma;

   void sumaNumeros(){
      this.suma=this.a+this.b;
   }
}
```

-Explicación del código

Se define una clase llamada SumaDosNumeros con tres atributos y un método.

```
class SumaDosNumeros{
   double a,b,suma;

   void sumaNumeros(){
   }
}
```

Como se debe realizar la suma, al utilizar la referencia ***this***, indica que el objeto creado a partir de esta clase o subclases, cuando mande llamar este método, realizará la suma utilizando sus mismos atributos.

```
this.suma=this.a+this.b;
```

En esta línea se indica que utilizando los atributos `a` y `b` del objeto que la mandó llamar, haga la suma y lo almacene en su atributo `suma`.

-Código clase Fibonacci

```
class Fibonacci extends SumaDosNumeros{
   int n;

   void fibonacci(){
      if(this.n<1)
         System.out.println("No se indicó un número válido");
      else if(this.n==1)
         System.out.println("0");
      else if(this.n==2)
         System.out.println("0 1");
      else{
         System.out.print("0 1");
         this.a=0;
         this.b=1;
         this.sumaNumeros();
         for(int i=3;i<=this.n;i++){
            System.out.print(" "+(int)this.suma);
            this.a=this.b;
            this.b=this.suma;
            this.sumaNumeros();
         }
         System.out.println();
      }
   }

   public static void main(String args[]){
      Fibonacci f = new Fibonacci();
      f.n=Integer.parseInt(args[0]);
      f.fibonacci();
   }

}
```

-Explicación del código

Como se puede observar en el diagrama de clases de este ejercicio, la clase Fibonacci es una clase heredada de la clase SumaDosNumeros ya que los números de la serie de Fibonacci se generan sumando los últimos dos números ya obtenidos en la serie. En Java

para indicar esta herencia se emplea la palabra ***extends*** cuando se define la clase heredada o subclase.

```
class Fibonacci extends SumaDosNumeros{
   int temporal,n;

   void fibonacci(){
   }
}
```

El problema requiere de entrada el número de términos de la serie que se desean calcular. Dicho número puede estar en los siguientes casos: que sea menor que uno, que sea uno, que sea dos, o que sea mayor a dos. Para cada caso se requiere ejecutar un código diferente:

Cuando es menor que uno, deberá imprimir que el número no es correcto.
Cuando es uno, deberá imprimir un cero.
Cuando son dos, deberá imprimir un cero y un uno.
Cuando sean más, se deberá imprimir el cero, el uno, y los números de la serie.

```
if(this.n<1)
   System.out.println("No se indicó un número válido");
else if(this.n==1)
   System.out.println("0");
else if(this.n==2)
   System.out.println("0 1");
else{
   System.out.print("0 1");
   this.a=0;
   this.b=1;
   this.sumaNumeros();
   for(int i=3;i<=this.n;i++){
      System.out.print(" "+(int)this.suma);
      this.a=this.b;
      this.b=this.suma;
      this.sumaNumeros();
   }
   System.out.println();
}
```

Al definir a la clase Fibonacci como una clase heredada de SumaDosNumeros, la primera puede utilizar como propios los atributos y métodos de la clase SumaDosNumeros, por ello es que se emplea la palabra ***this***. Es importante destacar que el método suma, realiza la suma de dos variables tipo ***double***, por lo que cuando se manda a imprimir, hay que realizar un ***cast*** explícito, para que sólo se imprima la parte entera:`(int)this.suma.`

Para terminar la definición de la clase Fibonacci sólo hace falta agregar un método ***main*** el cual será el punto de entrada al programa.

```
public static void main(String args[]){
   Fibonacci f = new Fibonacci();
   f.n=Integer.parseInt(args[0]);
   f.fibonacci();
}
```

En el método `main`, es necesario crear un objeto de la clase, y asignar el número de elementos a calcular en el atributo `n`. Este valor se dará en la línea de comandos cuando se invoque la ejecución del programa con el siguiente formato:

java Fibonacci númeroDeElementos

De esta forma, el número de elementos a calcular estará en `args[0]`ya que el número de parámetro empieza en cero, después del nombre de la clase. Cuando se lee un valor desde la línea de comandos éste será siempre de tipo cadena. Para este ejemplo, el valor de `n` debe ser entero, por lo que se utiliza el método `parseInt()` de la clase `Integer`, para convertir la cadena a un valor numérico; esto se hace en la línea:

```
f.n=Integer.parseInt(args[0]);
```

EJEMPLO 4.7 ELABORAR UN PROGRAMA QUE VERIFIQUE SI UNA FECHA DADA (DD-MM-AAAA) EXISTE O NO.

-Análisis y solución del problema

Se identifica una clase llamada VerificaFecha, la cual tendrá como atributos día(d), mes(m), y año(a). Contendrá métodos que verifiquen si el día existe para determinado mes y si el año es bisiesto.

-Diagrama de clase

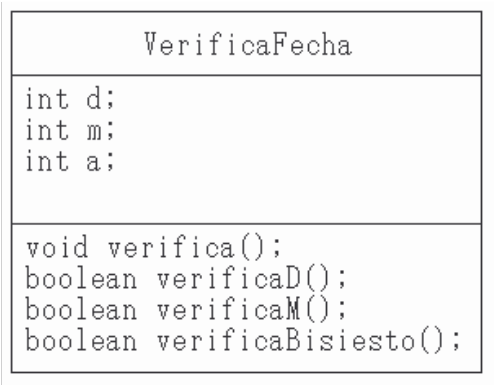


FIGURA 4.10 Diagrama de clase del ejemplo 4.7.

-Código

```
class VerificaFecha{
   int d,m,a;

   void verifica(){
      if(this.verificaM())
         System.out.println("Fecha valida");
      else
         System.out.println("Fecha incorrecta");
   }
   boolean verificaD(){
      switch(this.m){
         case 1:
         case 3:
         case 5:
         case 7:
         case 8:
         case 10:
         case 12:
            if((this.d>0)&&(this.d<32))
               return true;
         case 4:
         case 6:
         case 9:
         case 11:
            if((this.d>0)&&(this.d<31))
               return true;
         case 2:
            if(this.verificaBisiesto()){
               if((this.d>0)&&(this.d<30))
                  return true;
               else
                  return false;
            }
            else{
               if((this.d>0)&&(this.d<29))
                  return true;
               else
                  return false;
            }
         default:
            return false;
      }
   }
```

```
   boolean verificaM(){
      if((this.m>0)&&(this.m<13))
         return this.verificaD();
      else
         return false;
   }

   boolean verificaBisiesto(){
      if(((this.a%4==0)&&(this.a%100!=0))||(this.a%400==0))
         return true;
      else
         return false;
   }

   public static void main(String args[]){
      VerificaFecha vf = new VerificaFecha();
      vf.d=Integer.parseInt(args[0]);
      vf.m=Integer.parseInt(args[1]);
      vf.a=Integer.parseInt(args[2]);
      vf.verifica();
   }
}
```

-Explicación del código

Primeramente se escribe el código base de acuerdo al diagrama de clase del ejemplo.

```
 class VerificaFecha{
   int d,m,a;
   void verifica(){
   }
   boolean verificaD(){
   }
   boolean verificaM(){
   }
   boolean verificaBisiesto(){
   }
}
```

Ahora, para el método de verificaM(), se requiere revisar si el mes dado es válido.

```
boolean verificaM(){
   if((this.m>0)&&(this.m<13))
      return this.verificaD();
   else
      return false;
}
```

Para verificar si el año es bisiesto, el método verificaBisiesto() contiene:

```
boolean verificaBisiesto(){
   if(((this.a%4==0)&&(this.a%100!=0))||(this.a%400==0))
      return true;
   else
      return false;
}
```

La función que verifica el día, lo hace a partir del mes dado, por lo que si el mes es correcto, verifica el número de día; en el caso del mes de febrero revisa que el año sea bisiesto antes de verificar si es correcto el día.

```
boolean verificaD(){
   switch(this.m){
      case 1:
      case 3:
      case 5:
      case 7:
      case 8:
      case 10:
      case 12:
         if((this.d>0)&&(this.d<32))
            return true;
      case 4:
      case 6:
      case 9:
      case 11:
         if((this.d>0)&&(this.d<31))
            return true;
      case 2:
         if(this.verificaBisiesto()){
            if((this.d>0)&&(this.d<30))
               return true;
            else
               return false;
         }
         else{
            if((this.d>0)&&(this.d<29))
               return true;
            else
               return false;
         }
      default:
         return false;
   }
}
```

Por último, la función verifica(), muestra en pantalla si la fecha es correcta.

```
void verifica(){
   if(this.verificaM())
      System.out.println("Fecha valida");
   else
      System.out.println("Fecha incorrecta");
}
```

Ahora, utilizando la función `main`, se indicará desde la línea de comandos el valor de día, mes y año, por lo que utilizando `Integer.parseInt()`, se transformará cada uno de los valores dados y se asignarán a cada uno de los atributos.

```
public static void main(String args[]){
   VerificaFecha vf = new VerificaFecha();
   vf.d=Integer.parseInt(args[0]);
   vf.m=Integer.parseInt(args[1]);
   vf.a=Integer.parseInt(args[2]);
   vf.verifica();
}
```

Manejo de excepciones en Java

Una excepción en Java, se genera por errores al momento de ejecutarse una aplicación. Éstos suceden, por ejemplo, cuando se realiza una división entre cero, o se sobrepasa de los límites de un arreglo; existe una sentencia que permite lidiar con estos problemas.

Sentencia TRY-CATCH

Cuando hay un problema durante la ejecución del programa, éste lanza una excepción. Existen muchos tipos de excepciones, pero algunas de las más comunes son:

Excepción	Descripción
ArithmeticException	Se genera cuando ha ocurrido un error aritmético; por ejemplo una división entre cero.
ArrayIndexOutOfBoundsException	Se genera cuando se intenta acceder a un elemento de un arreglo fuera de los límites del mismo. Por ejemplo tratar de acceder al elemento 3 de arr[3].
NumberFormatException	Se genera cuando se trata de hacer una conversión de cadena a número, pero la cadena no tiene el formato apropiado.

Después de que ocurre una excepción, el programa detiene su ejecución, pero puede haber aplicaciones que requieren que no finalice el programa, sino que realice otra acción. Para lograr hacer esto, se debe utilizar la sentencia *try-catch*. Su sintaxis general es:

```
try
   proposición1
catch(expresión1)
   proposición2
catch(expresión2)
   proposición3
⋮
finally
   proposición_n
```

Donde la `proposición1` es todo el bloque de código que puede generar una o varias excepciones, y cada `catch` es el código que se deberá ejecutar en caso de encontrar una excepción igual a la que se encuentra dada por `expresión`; `expresión` es un objeto de tipo excepción.

Además de utilizar *try* y *catch*, se puede agregar después del último bloque de *catch* un bloque de finalización, y está indicado por la palabra *finally*. Este bloque se ejecuta independientemente de que se ejecute o no el bloque completo de *try*.

EJEMPLO 4.8 ELABORAR UN PROGRAMA QUE RECIBA DOS NÚMEROS COMO PARÁMETROS Y QUE REALICE LA MULTIPLICACIÓN ENTRE DICHOS NÚMEROS. SI NO SE INDICAN LOS PARÁMETROS DESDE LA LÍNEA DE COMANDOS, INDICAR CON UN MENSAJE LA FORMA EN QUE SE DEBE EJECUTAR EL PROGRAMA.

-Análisis y solución del problema

Se identifica una clase que realice la multiplicación de dos números que deberá recibir como parámetros desde línea de comandos.

-Diagrama de clase

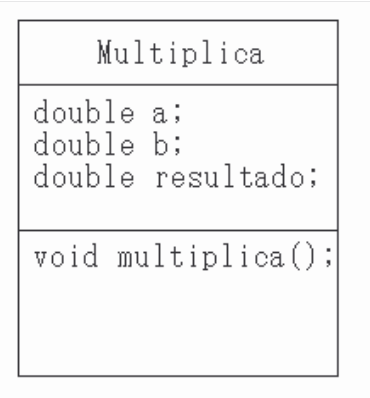


FIGURA 4.11 Diagrama de clase del ejemplo 4.8.

-Código

```
class Multiplica{
   double a,b,resultado;
   void multiplica(){
      this.resultado = this.a*this.b;
   }
   public static void main(String args[]){
      try{
         Multiplica m = new Multiplica();
         m.a = Double.parseDouble(args[0]);
         m.b = Double.parseDouble(args[1]);
         m.multiplica();
         System.out.println(m.a+" * "+m.b+" = "+m.resultado);
      }
      catch(ArrayIndexOutOfBoundsException aioobe){
         System.out.println("Faltaron parámetros");
         System.out.println("Sintaxis: java Multiplica num1”
                            +” num2");
      }
   }
}
```

-Explicación del código

El código base del programa es:

```
class Multiplica{
   double a,b,resultado;
   void multiplica(){
   }
}
```

Como se debe realizar la multiplicación se agrega dicha instrucción al método `multiplica()`:

```
this.resultado = this.a*this.b;
```

Ahora se agrega el método `main()`, asignando los valores obtenidos desde línea de comandos a las variables `a` y b.

```
public static void main(String args[]){
      Multiplica m = new Multiplica();
      m.a = Double.parseDouble(args[0]);
      m.b = Double.parseDouble(args[1]);
      m.multiplica();
      System.out.println(m.a+" * "+m.b+" = "+m.resultado);
}
```

Con esto se realiza la multiplicación de dos números. Si al invocar la ejecución del programa en la línea de comandos:

java Multiplica num1 num2

se omite uno o los dos números, el programa genera la excepción *ArrayIndexOutOfBound*s, dado que se trata de acceder a un elemento inexistente del arreglo, por lo que se utiliza la sentencia *try-catch*.

```
public static void main(String args[]){
   try{
      Multiplica m = new Multiplica();
      m.a = Double.parseDouble(args[0]);
      m.b = Double.parseDouble(args[1]);
      m.multiplica();
      System.out.println(m.a+" * "+m.b+" = "+m.resultado);
   }
   catch(ArrayIndexOutOfBoundsException aioobe){
      System.out.println("Faltaron parámetros");
      System.out.println("Sintaxis: java Multiplica num1
                  num2");
   }
}
```

El argumento de la sentencia `catch` debe ser el nombre de la excepción lanzada, en este caso debe atrapar un objeto de `ArrayIndexOutOfBounds`, que en el ejemplo se definió con el nombre `aioobe`.

EJEMPLO 4.9 MODIFICAR EL PROGRAMA ANTERIOR, PARA QUE CONSIDERE EL CASO EN QUE UNO DE LOS DOS NÚMEROS NO TENGA EL FORMATO NUMÉRICO (QUE CONTENGA LETRAS).

-Análisis y solución del problema

Teniendo el análisis del problema anterior, sólo se debe agregar la restricción que se deben escribir números, utilizando la excepción NumberFormatException.

-Diagrama de clase

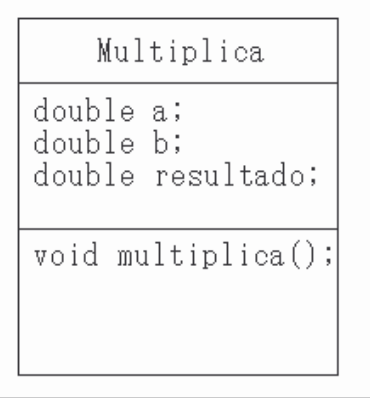


FIGURA 4.12 Diagrama de clase del ejemplo 4.9.

-Código

```
class Multiplica{
   double a,b,resultado;

   void multiplica(){
      this.resultado = this.a*this.b;
   }

   public static void main(String args[]){
      try{
         Multiplica m = new Multiplica();
         m.a = Double.parseDouble(args[0]);
         m.b = Double.parseDouble(args[1]);
         m.multiplica();
         System.out.println(m.a+" * "+m.b+" = "+m.resultado);
      }
      catch(ArrayIndexOutOfBoundsException aioobe){
         System.out.println("Faltaron parámetros");
         System.out.println("Sintaxis: java Multiplica num1”
                            +” num2");
      }
      catch(NumberFormatException nfe){
         System.out.println("Alguno de los dos parámetros no”
                            +” es un numero");
      }
   }
}
```

-Explicación del código

En el código del ejemplo anterior, se debe agregar la revisión de los formatos de los datos de entrada, no debiendo contener ningún otro carácter que no sea dígito. Esto se hace incluyendo otro bloque `catch`, el cual ahora debe atrapar un objeto la excepción `NumberFormatException` en `nfe`.

```
catch(NumberFormatException nfe){
   System.out.println("Alguno de los 2 parámetros no es un”
                      +" numero");
}
```

4.3 Tipos de clase

Hasta este momento se ha visto que una clase es definida por el usuario, describe los atributos y los métodos del objeto que se creará a partir de la misma. Los atributos definen el estado del objeto, es decir sus características, y los métodos definen el comportamiento de dicho objeto, es decir sus funciones. Pero antes de mencionar los modificadores que se pueden utilizar al momento de programar, es conveniente mencionar el concepto de paquete.

Paquete

Cuando se hacen sistemas más complejos muchas veces hay que agrupar la clases por carpetas o directorios; tal como se haría en una computadora para tener organizados todos los archivos. Este tipo de organización de clases, es lo que se conoce como paquete. Un paquete es un conjunto de clases que se encuentran relacionadas lógicamente entre sí y agrupadas por un mismo nombre.

EJEMPLO 4.10 IMPLEMENTAR UNA CLASE DENTRO DEL PAQUETE MISCLASES.SALUDOS, LA CUAL DEBERÁ DESPLEGAR EN PANTALLA UN SALUDO, MEDIANTE EL USO DE UN MÉTODO.

-Análisis y solución del problema

Se debe crear una clase dentro de un paquete, en este caso ***misClases.saludos***. Como se debe mandar un saludo, se creará otra clase llamada HolaMundo. No tendrá atributos, pero sí tendrá un método que despliegue el saludo.

-Diagrama de clase

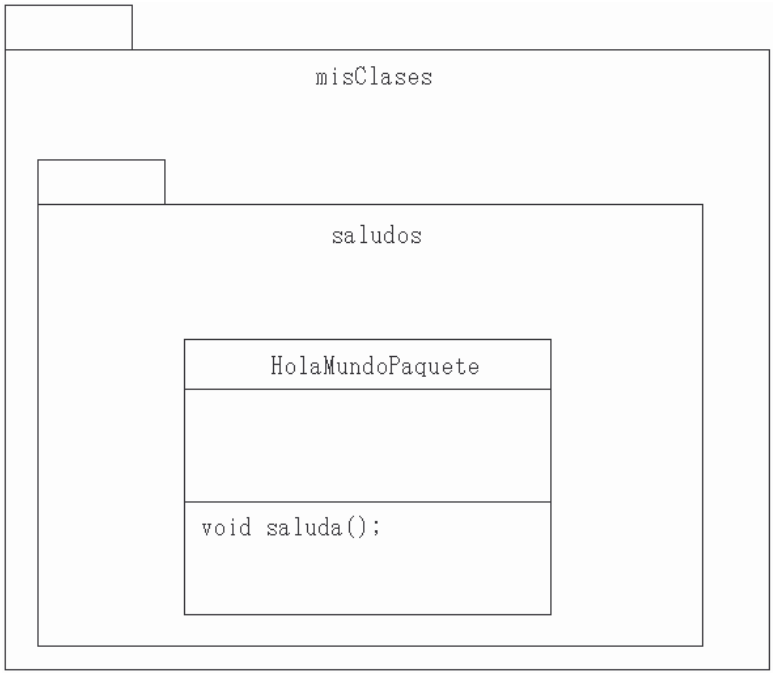


FIGURA 4.13 Diagrama de clase del ejemplo 4.10.

-Código

```
package misClases.saludos;

class HolaMundoPaquete{
   void saluda(){
      System.out.println("Hola Mundo");
      System.out.println("Desde un paquete");
   }
   public static void main(String args[]){
      HolaMundoPaquete hmp = new HolaMundoPaquete();
      hmp.saluda();
   }
}
```

-Explicación del código

Como se está especificando que la clase deberá estar contenida dentro del paquete *misClases.saludos*, se utiliza la sentencia *package*, seguida del nombre del paquete.

```
package misClases.saludos;
```

A continuación se define la clase con su método

```
class HolaMundoPaquete{
```

```
   void saluda(){
      System.out.println("Hola Mundo");
      System.out.println("Desde un paquete");
   }
}
```

Y se agrega un método `main` para hacer la prueba de la clase.

```
public static void main(String args[]){
   HolaMundoPaquete hmp = new HolaMundoPaquete();
   hmp.saluda();
}
```

La forma en que se compila esta clase es utilizando la opción *–d* para indicar el directorio o paquete donde se almacenará la clase; en este caso se utilizará el directorio donde se encuentra el archivo *.java*. Por lo que la línea de comandos para compilar el programa es:

```
javac –d . HolaMundoPaquete.java
```

Para ejecutarlo, se utiliza la siguiente instrucción:

```
java misClases.saludos.HolaMundoPaquete
```

en donde al nombre de la clase se le antepone el nombre del paquete unido por un punto.

4.3.1 Modificadores

Como se vio anteriormente, la base en código de una clase es:

```
class nombre_clase{
     //cuerpo de la clase
}
```

Pero en cuestiones de seguridad y estabilidad de un programa, existen modificadores que ayudan a volver un sistema más rígido. Entre estos modificadores están los modificadores *final* y *abstract* que definen ciertas restricciones de alteración. Existen otros modificadores que definen restricciones de visibilidad o acceso. Los modificadores se ponen antes de la palabra *class*.

- Modificador FINAL

Este modificador tiene diferentes efectos dependiendo de a qué elemento se le aplique, es decir a un atributo, a un método o a una clase.

Cuando se aplica este modificador a un atributo, bloquea el acceso a esa variable para cambiar su valor. Por ejemplo:

```
final int limiteSuperior = 6;
```

En este caso el atributo `limiteSuperior` mantendrá su valor de 6 hasta el término de la ejecución del programa.

Si se define una variable como `final` y luego se intenta ejecutar una asignación a dicho atributo, se generaría un error, por lo cual se le debe dar un valor inicial al momento de definirla como variable `final`.

Cuando se aplica este modificador a un método, lo que provoca es que el método no pueda ser modificado, es decir que una subclase no puede redefinir el método.

Cuando se aplica este modificador a una clase, es para indicar que hasta ahí termina la herencia, es decir, no se pueden generar subclases a partir de ésta. Adicionalmente, todos sus métodos se convierten automáticamente en final.

- Modificador ABSTRACT

Cuando una clase se define como abstracta, es porque no se crearán objetos a partir de ésta, debido a que sólo tiene como objetivo proporcionar los atributos y métodos que compartirán todas las subclases.

NOTA: Los modificadores *final* y *abstract*, se pueden utilizar también con los modificadores de acceso. Es decir:

```
modificadorAcceso tipoClase class NombreClase{
  //Cuerpo de la clase
}
```

Modificadores de acceso

Los modificadores de acceso se utilizan para ocultar la información de los atributos, esto se hace para evitar que la información del objeto sea modificada directamente desde otra clase o subclase. Por lo que si se desea modificar la información de los objetos, se deberán crear métodos que permitan realizar dichas modificaciones. En términos de seguridad de sistemas, el utilizar modificadores de visibilidad permiten crear aplicaciones más seguras.

El modificador de acceso también se puede utilizar para atributos y métodos.

- Modificador PUBLIC

Cuando un elemento es declarado público, esta característica permite que sea de fácil acceso desde cualquier otra clase o subclase que necesite utilizarlo.

- Modificador PRIVATE

Cuando un elemento es declarado privado, sólo se puede acceder por los métodos de la misma clase. Esto indica que no puede ser accedido por los métodos de cualquier otra clase o subclases.

- Modificador PROTECTED

Un miembro que es declarado protegido se comporta de la misma manera que el privado para otras clases, excepto para las clases del mismo paquete o sus subclases independientemente del paquete en el que se encuentran, para éstas se comporta como si fuera *public*.

En la figura 4.14 se muestra si se tiene o no acceso a un elemento (atributo o método) desde otras clases, de acuerdo con el modificador que tiene. Hay que hacer notar que incluso el no tener un modificador afecta su acceso.

	public	protected	*sin modificador*	private
Acceso dentro de la misma clase.	*SI*	*SI*	*SI*	*SI*
Acceso desde clases dentro del mismo paquete, no importando si hay o no herencia.	*SI*	*SI*	*SI*	*NO*
Acceso desde subclases declaradas dentro de otro paquete.	*SI*	*SI*	*NO*	*NO*
Acceso desde cualquier otra clase en otro paquete.	*SI*	*NO*	*NO*	*NO*

FIGURA 4.14 Tabla de modificadores de acceso.

EJEMPLO 4.11 IMPLEMENTAR UNA CLASE QUE TENGA UN ÚNICO ATRIBUTO DE TIPO ENTERO CON VISIBILIDAD ***PRIVATE***, Y DOS MÉTODOS ***PUBLIC***, UNO PARA ASIGNAR UN VALOR AL ATRIBUTO, Y OTRO PARA OBTENER DICHO VALOR. ESTA CLASE DEBERÁ ESTAR ALMACENADA EN EL PAQUETE MISCLASES.PRUEBAS. DE IGUAL FORMA IMPLEMENTAR OTRA CLASE QUE UTILIZARÁ LA CLASE DESCRITA ANTERIORMENTE PARA MANIPULAR UN VALOR ENTERO, ÉSTA DEBERÁ ESTAR EN EL PAQUETE MISCLASES.OTRASPRUEBAS.

-Análisis y solución del problema

Se implementarán dos clases, una se llamará Numero y otra CambiaValor. Como el enunciado describe claramente los atributos y métodos que deberá contener cada clase así como la pertenencia a un paquete, se pasa directamente a la elaboración del diagrama de clases correspondiente.

-Diagrama de clase

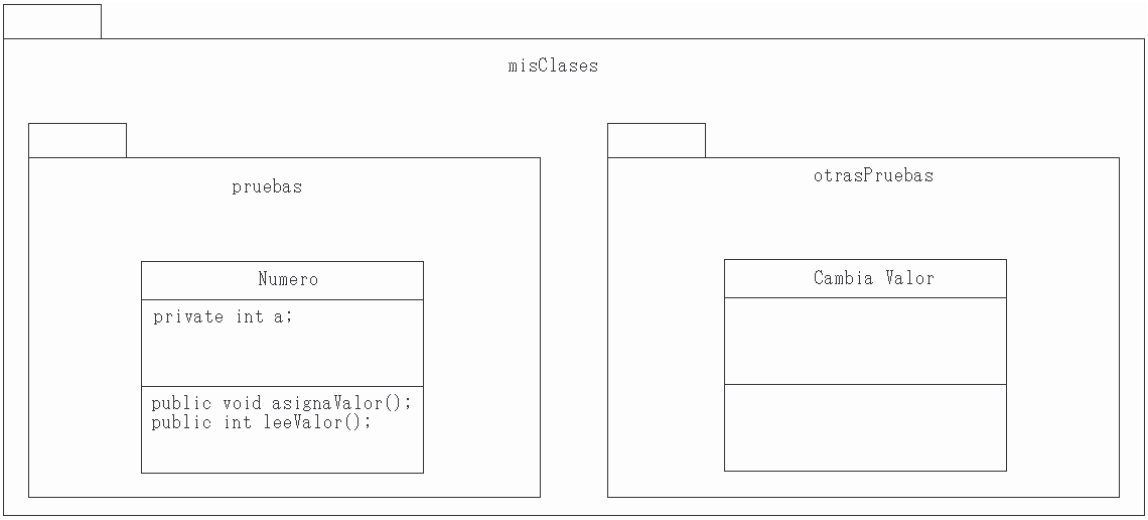


FIGURA 4.15 Diagrama de clase del ejemplo 4.11.

-Código de la clase Numero

```
package misClases.pruebas;

public class Numero{
   private int a;

   public void asignaValor(int x){
      this.a = x;
   }
   public int leeValor(){
      return this.a;
   }
}
```

-Explicación de código

Como la clase debe estar en el paquete `misClases.pruebas`, se debe declarar al inicio del programa:

```
package misClases.pruebas;
```

Ahora se define la estructura de la clase

```
public class Numero{
   private int a;

   public void asignaValor(){
   }
   public int leeValor(){
   }
}
```

Debido a que el atributo es `private`, los métodos deben ser `public`, de esta forma se puede acceder al atributo de la clase. El método para asignar un valor al atributo, deberá recibir un parámetro, por lo que el método queda de la siguiente forma.

```
public void asignaValor(int x){
   this.a = x;
}
```

Y el método para obtener el valor actual del atributo únicamente lo debe entregar

```
public int leeValor(){
   return this.a;
}
```

-Código de la clase CambiaValor

```
package misClases.otrasPruebas;

import misClases.pruebas.Numero;

public class CambiaValor{
   public static void main(String args[]){
      Numero n = new Numero();
      n.asignaValor(Integer.parseInt(args[0]));
      System.out.println("Valor asignado a la variable:
"+n.leeValor());
   }
}
```

-Explicación de código

Esta clase deberá estar contenida en el paquete `misClases.otrasPruebas`.

```
package misClases.otrasPruebas;
```

Como en esta clase se utilizarán los métodos y atributos de la clase `Numero`, se debe importar para poder hacer uso de ella, debido a que se encuentra en otro paquete.

```
import misClases.pruebas.Numero;
```

Se escribe el código base de la clase

```
public class CambiaValor{
}
```

Dentro del método `main`, se definie un objeto de tipo `Numero`, y utilizando un parámetro obtenido desde la línea de comandos, se asignará el valor al atributo del objeto y se obtendrá el valor para desplegarlo en pantalla.

```
public static void main(String args[]){
   Numero n = new Numero();
   n.asignaValor(Integer.parseInt(args[0]));
   System.out.println("Valor asignado a la variable: "
                      +n.leeValor());
}
```

EJEMPLO 4.12 MODIFICAR LA CLASE NUMERO DEL EJEMPLO 4.11, HACIENDO QUE LOS MÉTODOS SEAN *PROTECTED*. CON BASE EN LA TABLA DE LA FIGURA 4.14 HAGA LAS CORRECCIONES NECESARIAS PARA QUE LA CLASE CAMBIAVALOR PUEDA HACER USO DE LA CLASE NUMERO.

-Análisis y solución del problema

Dentro de la clase Numero, los métodos se deben cambiar de *public* a *protected*. Ahora, después de hacer esta modificación y volviendo a compilar ambas clases, cuando se trata de compilar CambiaValor enviará un mensaje indicando que como los métodos son *protected* no se puede hacer uso de ellos en la clase. Analizando la tabla de la 4.14, se observa que cuando una clase se encuentra en otro paquete, pero es subclase de la clase a utilizar, se pueden utilizar los elementos definidos como *protected*, por lo que la siguiente modificación se hará dentro de la clase CambiaValor, donde se definirá esta clase como subclase de Numero usando la palabra *extends*. Por último se deberá crear un objeto en esta subclase.

-Código de la clase Numero

```
package misClases.pruebas;

public class Numero{
   private int a;

   protected void asignaValor(int x){
      this.a = x;
```

```
   }
   protected int leeValor(){
      return this.a;
   }
}
```

-Código de la clase CambiaValor

```
package misClases.otrasPruebas;

import misClases.pruebas.Numero;

public class CambiaValor extends Numero{
   public static void main(String args[]){
      CambiaValor cv = new CambiaValor();
      cv.asignaValor(Integer.parseInt(args[0]));
      System.out.println("Valor asignado a la variable: "
                         +cv.leeValor());
   }
}
```

4.3.2 Métodos constructores

Un constructor, es un método que permite crear objetos a partir de una clase. Cuando se construye un objeto, éste contiene sólo una copia de los atributos, de los métodos no crea copia ya que todos los objetos creados por la misma clase los comparten.

Para crear un objeto se escribe la siguiente sentencia:

NombreClase objeto = new NombreClase( );

Esta sentencia crea un nuevo objeto de la clase NombreClase. Cuando se define una clase, por omisión se define su método constructor NombreClase( ) a menos que se defina explícitamente.

EJEMPLO 4.13 IMPLEMENTAR UNA CLASE LLAMADA PERSONA, QUE ALMACENE SU NOMBRE Y EDAD.

-Análisis y solución del problema

Por la sencillez del enunciado, se identifica una clase llamada Persona, que tendrá como atributos nombre y edad.

-Diagrama de clase

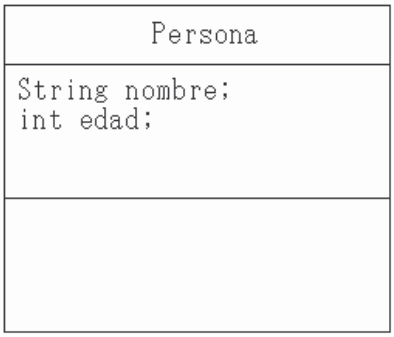


FIGURA 4.16 Diagrama de clase del ejemplo 4.13.

-Código

```
public class Persona{
   String nombre;
   int edad;

   public static void main(String args[]){
      Persona p = new Persona();

      System.out.println("Persona");
      System.out.println("nombre: "+p.nombre);
      System.out.println("edad: "+p.edad+"\n");
   }
}
```

-Explicación del código

El código base del programa es:

```
public class Persona{
   String nombre;
   int edad;
}
```

Se agrega el método `main`, creando un objeto, sin inicializar los valores de los atributos, para observar los valores con los que se inicia por omisión.

```
public static void main(String args[]){
   Persona p = new Persona();

   System.out.println("Persona");
   System.out.println("nombre: "+p.nombre);
   System.out.println("edad: "+p.edad+"\n");
}
```

También se pueden escribir métodos constructores, para inicializar automáticamente los objetos, o por cualquiera otra razón; esto se hace implementando una sobrecarga de métodos, es decir, que se pueden crear varios métodos con el mismo nombre, pero deben diferir en el número y/o tipo de parámetros..

Referencia THIS

Otro concepto muy importante en la programación orientada a objetos es la palabra reservada *this*, la cual permite hacer operaciones con los atributos del objeto que invocó al método pero sin ser explícitos.

Declaración de un arreglo en Java

A diferencia del lenguaje C, en Java se tiene cierta flexibilidad al momento de declarar arreglos, debido a que no es necesario indicar el número de elementos incluidos en el arreglo. Se tienen dos formas de declarar un arreglo, éstas son:

tipoDato nombreArreglo[];

tipoDato[] nombreArreglo;

Como se puede observar, es indistinto el uso de los corchetes, ya sea después del tipo de dato o después del nombre del arreglo. También cuando se va a crear un objeto de tipo arreglo, se pueden utilizar cualquiera de las formas anteriores, y se puede o no, indicar el número de elementos, como se muestra a continuación.

tipoDato nombreArreglo[] = new tipoDato[];

tipoDato[] nombreArreglo = new tipoDato[numeroElementos];

EJEMPLO 4.14 ELABORE UNA CLASE QUE REALICE LA SUMA DE UN ARREGLO UNIDIMENSIONAL CON 5 ELEMENTOS.

- Análisis y solución del problema

Se utilizará una clase, la cual contendrá dos variables de tipo double, un arreglo *arr*, y la variable *suma*.
Se utilizará un método para realizar la suma de los 5 elementos.

- Diagrama de clase

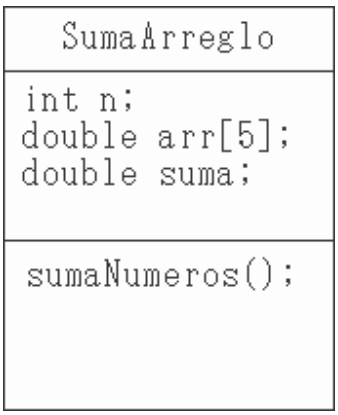


FIGURA 4.17 Diagrama de clase del ejemplo 4.14.

- Código

```
public class SumaArreglo{
   double arr[] = new double[5];
   double suma;
   public void sumaArr(){
      this.suma=0;
      for(int i=0;i<5;i++){
         this.suma=this.suma+this.arr[i];
      }
      System.out.println("El resultado de sumar el"+
                         " arreglo es: "+this.suma);
   }
   public static void main(String args[]){
      SumaArreglo sa = new SumaArreglo();
      sa.arr[0]=1;
      sa.arr[1]=2;
      sa.arr[2]=3;
      sa.arr[3]=4;
      sa.arr[4]=5;
      sa.sumaArr();
   }
}
```

- Explicación del código

El código base a partir del diagrama de clase es:

```
public class SumaArreglo{
   double arr[] = new double[5];
   double suma;
   public void sumaArr(){
   }
}
```

Ahora se implementa la suma del arreglo, utilizando la sentencia de control de flujo *for*, y la referencia this, para indicar que se utilizarán las variables del objeto que invoca al método. Así mismo, se deberá agregar en el método la sentencia de impresión con el resultado de la suma.

```
public void sumaArr(){
   this.suma=0;
   for(int i=0;i<5;i++){
      this.suma=this.suma+this.arr[i];
   }
   System.out.println("El resultado de sumar el"+
                      " arreglo es: "+this.suma);
}
```

Con lo que sólo falta agregar el punto de entrada al programa, es decir, el método ***main***, donde se creará un objeto de la clase, y se darán valores iniciales al arreglo con los 5 valores a sumar, y después de mandará llamar al método del objeto.

```
public static void main(String args[]){
   SumaArreglo sa = new SumaArreglo();
   sa.arr[0]=1;
   sa.arr[1]=2;
   sa.arr[2]=3;
   sa.arr[3]=4;
   sa.arr[4]=5;
   sa.sumaArr();
}
```

4.3.3 Interfaces

Una interfaz es similar a una clase en cuanto a que contiene atributos y métodos, pero a diferencia de una clase una interfaz no crea nuevas clases, agrega características a una clase; es decir, aquellas características que no obtiene por su herencia o por los atributos o métodos propios.

La sintaxis básica para definir una interfaz es muy similar a la de una clase:

```
public interface NombreInterfaz{
   //Cuerpo de la interfaz
}
```

Donde:

El modificador de acceso `public` indica que la interfaz puede ser utilizada por cualquier clase de cualquier paquete. En caso de omitir el modificador `public`, las clases sólo podrán ser accesibles para las clases del mismo paquete.

El cuerpo de la interfaz puede incluir declaración de constantes y declaración de métodos. Pero los métodos no podrán tener cuerpo. Por lo que cuando se utiliza una interfaz dentro de una clase, la forma de emplear los métodos definidos en la interfaz deberá ser a través de una redefinición de los métodos en la propia clase.

De igual forma que una clase puede heredar atributos y métodos de otra, las interfaces también pueden generar subinterfaces, para ello se utiliza la palabra ***extends***.

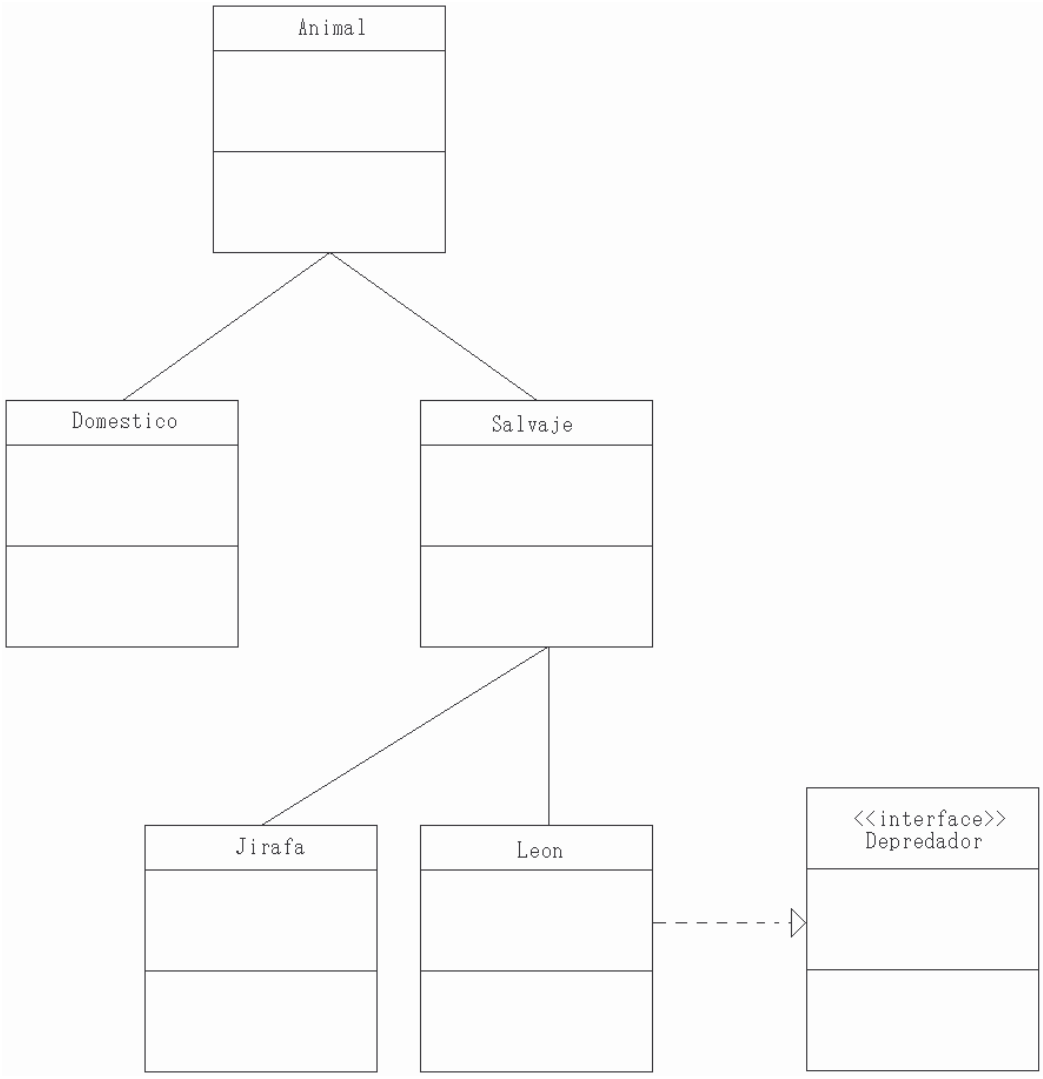


FIGURA 4.18 Diagrama de clases donde se indica la implementación de una interfaz.

Si una clase va a utilizar una interfaz, al definir esta clase se debe incluir la palabra ***implements*** de la siguiente forma:

```
modificador class NombreClase extends NombreClasePadre
implements NombreInterfaz{
   //Cuerpo clase
}
```

Una ayuda en la compresión de las interfaces se puede ver en el diagrama de clases de la figura 4.18, donde se puede observar que existe una clase llamada Leon, la cual es un Animal Salvaje y, en este caso, también es un Depredador, pero como no se puede realizar una herencia de dos padres distintos, se utiliza una interfaz, la cual permita heredar atributos y métodos desde otra "clase".

EJEMPLO 4.15 CON BASE EN EL DIAGRAMA DE CLASES DE LA FIGURA 4.17, OBTENGA EL CÓDIGO BASE DE LA CLASE LEON SIN ATRIBUTOS NI MÉTODOS.

-Análisis y solución del problema

Con base en el diagrama de clases, la clase Leon es subclase de la clase Salvaje, pero utiliza la interfaz Depredador para obtener atributos y métodos no heredados por la clase Salvaje.

-Código

```
public class Leon extends Salvaje implements Depredador{
}
```

-Explicación del código

Cuando en una clase se "heredarán" atributos de más de una clase, se utiliza la palabra *implements*, la cual permite acceder a atributos o métodos extras a la clase y que no fueron heredadas por la clase padre.

Ejercicios Propuestos

1) Investigar la precedencia de los tipos de datos primitivos: char, double, int, float, etc. Además indicar cuándo se debe hacer un *cast* explícito.

2) Investigar el rango de valores de los tipos de datos primitivos.

3) Investigar cuántas y cuáles son las palabras reservadas utilizadas en Java.

4) Elaborar un programa en Java que muestre en pantalla los primeros ***n*** número enteros. Es decir, que si se ejecuta:

```
java NombreClase 6
```

El resultado sea:

```
1 2 3 4 5 6
```

5) Implementar una clase que realice la multiplicación entre dos vectores de dimensión tres y obtenga un escalar. Es decir:

$$\begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

6) Implementar una clase que realice la siguiente multiplicación

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

7) Implementar una clase que obtenga la matriz inversa de una matriz de 3x3

8) Implementar una clase que realice el método de eliminación Gaussiana para sistemas de ecuaciones lineales. Resuelva el ejemplo

9) Implementar una clase que realice el método de Gauss-Jordan para sistemas de ecuaciones lineales.

10) Utilizando la clase ejemplo 4.13, implementar una clase llamada Empleado, proponer cuatro atributos propios de Empleado.

11) Investigar cómo se hace una sobrecarga de constructores, y modificar la clase del inciso 10, de tal forma que se pueda inicializar un objeto de la clase, al pasar parámetros desde la línea de comandos.

12) Implementar una clase que realice una operación matemática simple, al ejecutar la clase se debe indicar los operandos y el tipo de operación a realizar; por ejemplo:

```
java NombreClase 5 + 3
```

Y la clase deberá devolver el resultado; en este caso, el resultado es 8

13) Modificar la clase anterior del inciso 12, y utilizando el manejo de excepciones, que indique cuándo hay una división entre cero.

14) Implementar dentro de un paquete, una clase por cada operación matemática, es decir una clase para Suma, Resta, Multiplicación y otra para División. Los atributos de estas clases deberán ser *protected*.

CAPÍTULO 5

MÉTODOS NUMÉRICOS PARA SOLUCIÓN DE SISTEMAS Y ECUACIONES AVANZADAS

Introducción

En el capítulo dos se describieron los métodos numéricos que resuelven ecuaciones algebraicas y sistemas de ecuaciones, además de utilizar interpolación para obtener una aproximación de un valor intermedio entre dos puntos. En este capítulo se revisarán métodos numéricos que resuelven problemas matemáticos más complejos.

5.1 Derivación e integración numérica.

5.1.1 Derivación numérica

La derivada de una función, por definición, es la pendiente de la recta tangente a un punto en la curva. Usando el concepto de límite, la derivada se define como:

$$\frac{d}{dx} f(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

Como se puede observar, el término al que se le obtiene el límite viene siendo la pendiente de una recta que pasa por dos puntos sobre la curva.

Por lo tanto, cuando se utilizan los métodos numéricos para encontrar la derivada en un punto de una función, se recurre a utilizar un par de puntos, para obtener el valor de la pendiente de una recta que pasa por dichos puntos, y que se puede aproximar al valor de la derivada en el punto deseado.

Derivación por Serie de Taylor

La fórmula de la serie de Taylor es:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Donde *a*, es el punto respecto al cual se desea obtener el desarrollo de la serie.

Suponiendo que se desea obtener la derivada en el punto x_i, se obtendrá el desarrollo de la serie de Taylor en el punto $(x_i + \Delta x)$, respecto al punto x_i:

$$f(x_i + \Delta x) = f(x_i) + f'(x_i)(\Delta x) + \frac{f''(x_i)(\Delta x)^2}{2!} + \frac{f'''(x_i)(\Delta x)^3}{3!} + \ldots$$

El desarrollo de la serie en el punto $(x_i - \Delta x)$, respecto de x_i:

$$f(x_i - \Delta x) = f(x_i) - f'(x_i)(\Delta x) + \frac{f''(x_i)(\Delta x)^2}{2!} - \frac{f'''(x_i)(\Delta x)^3}{3!} + \ldots$$

Para obtener la aproximación de la derivada en un punto, existen tres formas de obtenerlo, por diferencias hacia adelante, hacia atrás, y por diferencias centrales.

La aproximación por diferencias hacia adelante, se obtiene del desarrollo de la serie para el punto $(x_i + \Delta x)$, utilizando únicamente los dos primeros términos, es decir:

$$f(x_i + \Delta x) = f(x_i) + f'(x_i)(\Delta x)$$

$$f(x_i + \Delta x) - f(x_i) = f'(x_i)(\Delta x)$$

$$f'(x_i) = f'_i = \frac{f(x_i + \Delta x) - f(x_i)}{(\Delta x)}$$

Como se puede observar, este resultado concuerda con la definición de la pendiente de una recta que pasa por dos puntos; en este caso, un punto es $(x_i + \Delta x)$, y el otro es x_i.

Para las diferencias divididas hacia atrás, ahora se utiliza el desarrollo de Taylor para el punto $(x_i - \Delta x)$, utilizando únicamente los dos primero términos:

$$f(x_i - \Delta x) = f(x_i) - f'(x_i)(\Delta x)$$

$$f'(x_i)(\Delta x) = f(x_i) - f(x_i - \Delta x)$$

$$f'(x_i) = \frac{f(x_i) - f(x_i - \Delta x)}{(\Delta x)}$$

Para las diferencias centrales, se resta $f(x_i - \Delta x)$ a $f(x_i + \Delta x)$, tomando sólo los dos primeros términos de ambas expresiones, teniendo:

$$f(x_i+\Delta x)-f(x_i-\Delta x)=f_i+f'_i(\Delta x)-(f_i-f'_i(\Delta x))$$
$$f(x_i+\Delta x)-f(x_i-\Delta x)=f_i+f'_i(\Delta x)-f_i+f'_i(\Delta x)$$
$$f(x_i+\Delta x)-f(x_i-\Delta x)=2f'_i(\Delta x)$$

$$f'_i=\frac{f(x_i+\Delta x)-f(x_i-\Delta x)}{2(\Delta x)}$$

De estos cálculos, se obtienen las ecuaciones para:

Diferencias hacia delante

$$f'_i=\frac{f(x_i+\Delta x)-f(x_i)}{(\Delta x)}=\frac{f_{i+1}-f_i}{\Delta x}$$

Diferencias hacia atrás

$$f'_i=\frac{f(x_i)-f(x_i-\Delta x)}{(\Delta x)}=\frac{f_i-f_{i-1}}{\Delta x}$$

Diferencias centrales

$$f'_i=\frac{f(x_i+\Delta x)-f(x_i-\Delta x)}{2(\Delta x)}=\frac{f_{i+1}-f_{i-1}}{2(\Delta x)}$$

Para obtener la derivada de segundo orden, se utilizan los primeros 3 términos de la serie de Taylor para $f(x_i+\Delta x)$ y $f(x_i-\Delta x)$, sumando ambas ecuaciones se obtiene:

$$f(x_i+\Delta x)+f(x_i-\Delta x)=f_i+f'_i(\Delta x)+\frac{f''_i(\Delta x)^2}{2!}+f_i-f'_i(\Delta x)+\frac{f''_i(\Delta x)^2}{2!}$$
$$f(x_i+\Delta x)+f(x_i-\Delta x)=2f_i+f''_i(\Delta x)^2$$

$$f''_i=\frac{f(x_i+\Delta x)-2f_i+f(x_i-\Delta x)}{(\Delta x)^2}$$

EJEMPLO 5.1 UTILIZANDO LAS TRES FÓRMULAS PARA OBTENER NUMÉRICAMENTE LA DERIVADA, OBTENER LA DERIVADA DE $f(x)=x^3-8x^2+15x+3$ EN $x=1.5$; UTILIZAR UN Δx TANTO DE 0.1 COMO DE 0.001, COMPARE RESULTADOS.

-Solución

Primero se obtienen los valores de la función utilizando el Δx de 0.1:

x	f(x)
1.4	11.064
1.5	10.875
1.6	10.616

Diferencias hacia adelante con Δx de 0.1

$$f'_i=\frac{f(x_i+\Delta x)-f(x_i)}{(\Delta x)}=\frac{10.616-10.875}{(0.1)}=\frac{-0.259}{0.1}=-2.59$$

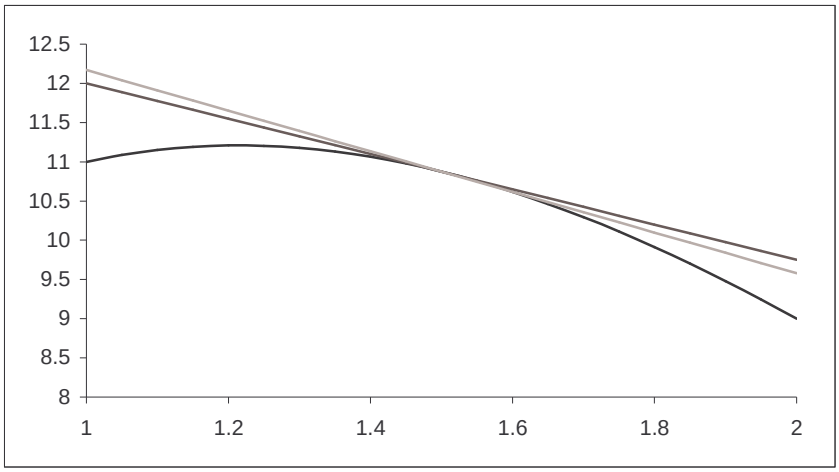


FIGURA 5.1 Gráfica de la recta obtenida por Diferencias hacia adelante con Δx de 0.1, del ejemplo 5.1, comparada con la recta tangente sobre el mismo punto.

Diferencias hacia atrás con Δx de 0.1

$$f'_i=\frac{f(x_i)-f(x_i-\Delta x)}{(\Delta x)}=\frac{10.875-11.064}{(0.1)}=\frac{-0.189}{0.1}=-1.89$$

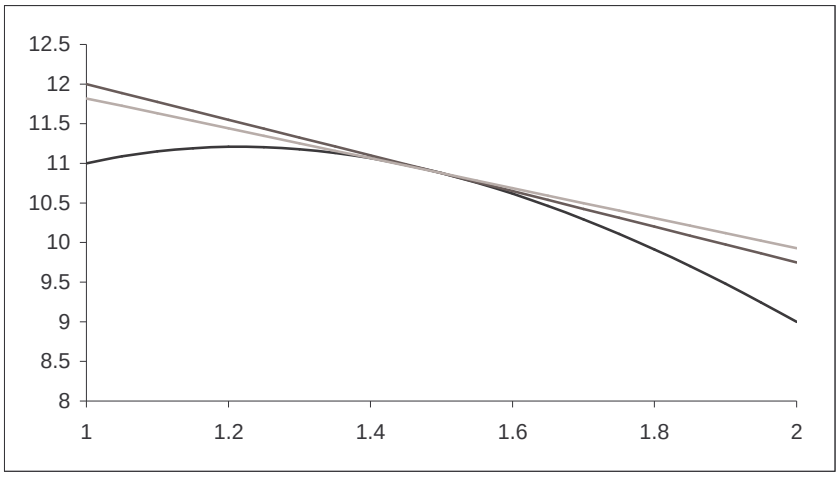


FIGURA 5.2 Gráfica de la recta obtenida por Diferencias hacia atrás con Δx de 0.1, del ejemplo 5.1, comparada con la recta tangente sobre el mismo punto.

Diferencias centrales con Δx de 0.1

$$f'_i = \frac{f(x_i + \Delta x) - f(x_i - \Delta x)}{2(\Delta x)} = \frac{10.616 - 11.064}{2(0.1)} = \frac{-0.448}{0.2} = -2.24$$

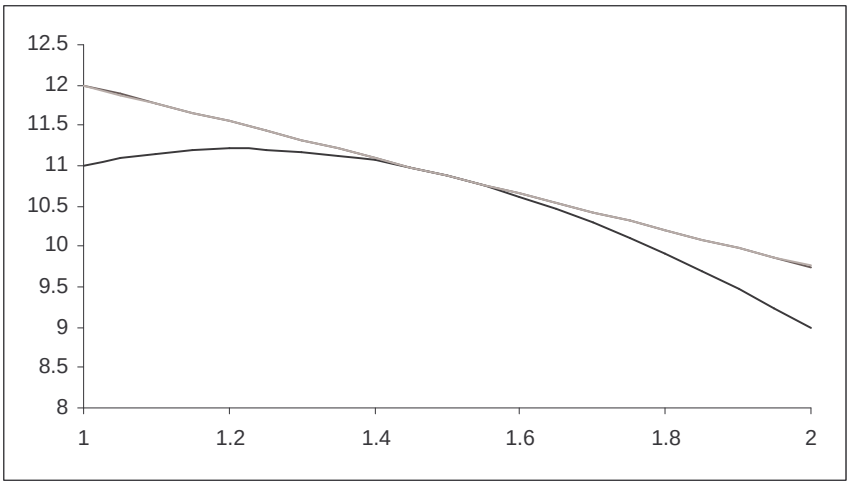


FIGURA 5.3 Gráfica de la recta obtenida por Diferencias centrales con Δx de 0.1, del ejemplo 5.1, comparada con la recta tangente sobre el mismo punto.

Ahora se calculan los valores de la función con Δx de 0.001:

x	f(x)
1.499	10.8772465
1.5	10.875
1.501	10.8727465

Diferencias hacia adelante con Δx de 0.001

$$f'_i = \frac{f(x_i + \Delta x) - f(x_i)}{(\Delta x)} = \frac{10.8727465 - 10.875}{(0.001)} = \frac{-0.0022535}{0.001} = -2.2535$$

Diferencias hacia atrás con Δx de 0.001

$$f'_i = \frac{f(x_i) - f(x_i - \Delta x)}{(\Delta x)} = \frac{10.875 - 10.8772465}{(0.001)} = \frac{-0.0022465}{0.001} = -2.2465$$

Diferencias centrales con Δx de 0.001

$$f'_i = \frac{f(x_i + \Delta x) - f(x_i - \Delta x)}{2(\Delta x)} = \frac{10.8727465 - 10.8772465}{2(0.001)} = \frac{-0.0045}{0.002} = -2.25$$

Obteniendo la derivada de la función y evaluando en el punto 1.5, se observa que la derivada en 1.5 es -2.25. Por lo que se puede concluir que a menor valor de Δx, menor es el error cometido al aproximarse.

El error que se comete por Diferencias centrales, no es tan grande. Esto se puede observar en la siguiente tabla:

Δx = 0.1			Δx = 0.001		
	ε absoluto	ε relativo		ε absoluto	ε relativo
-2.59	0.34	15.11%	-2.2535	0.0035	0.15%
-1.89	0.36	16%	-2.2465	0.0035	0.15%
-2.24	0.01	0.44%	-2.25	0	0%

EJEMPLO 5.2 UTILIZANDO LAS TRES FÓRMULAS PARA OBTENER NUMÉRICAMENTE LA DERIVADA, OBTENER LA DERIVADA DE $f(x) = e^{x^2}\cos(x^3)$ EN $x = 0.5$; UTILIZAR UN Δx DE 0.001.

-Solución

Primero se obtienen los valores que se utilizarán para el método:

x	f(x)
0.499	1.27285427
0.5	1.27400702
0.501	1.27516216

Diferencias hacia adelante con Δx de 0.001

$$f'_i = \frac{f(x_i + \Delta x) - f(x_i)}{(\Delta x)} = \frac{1.27516216 - 1.27400702}{(0.001)} = \frac{0.00115514}{0.001} = 1.15514$$

Diferencias hacia atrás con Δx de 0.001

$$f'_i = \frac{f(x_i) - f(x_i - \Delta x)}{(\Delta x)} = \frac{1.27400702 - 1.27285427}{(0.001)} = \frac{0.00115275}{0.001} = 1.15275$$

Diferencias centrales con Δx de 0.001

$$f'_i = \frac{f(x_i + \Delta x) - f(x_i - \Delta x)}{2(\Delta x)} = \frac{1.27516216 - 1.27285427}{2(0.001)} = \frac{0.00230789}{0.002} = 1.153945$$

EJEMPLO 5.3 IMPLEMENTAR UN PROGRAMAORIENTADO A OBJETOS QUE, UTILIZANDO LAS TRES APROXIMACIONES A LA DERIVADA DE PRIMER ORDEN, OBTENGA LA DERIVADA DE

$$f(x) = x^3 - 8x^2 + 15x + 3 \text{ EN } x = 1.5.$$

UTILICE UN Δx DE 0.001. LA CLASE QUE CALCULE LOS VALORES DE LA FUNCIÓN DEBERÁ ESTAR EN OTRO PROGRAMA.

-Análisis y solución del problema

Se identifican dos clases, una donde se implementarán las fórmulas de las aproximaciones, la cual se nombrará *DerivacionNumerica*; y otra donde se tendrá la función, y se creará un objeto de la clase *DerivacionNumerica*.
En la clase *DerivacionNumerica* se deberán definir los atributos necesarios para almacenar la información alrededor del punto del cual se desea obtener la derivada. Y un método por cada fórmula.

En la clase donde se definirá la función, que se denominará *EjemploCincoTres*, se deberán definir dos atributos por la información que se requiere, o sea, el punto donde se desea la derivada, y el incremento para obtener los otros dos puntos. Y dos métodos, uno para evaluar la función en los puntos dados, y otro para crear un objeto de la clase *DerivacionNumerica* y desplegar la derivada en pantalla, como se muestra en la figura 5.4.

-Diagrama de clase

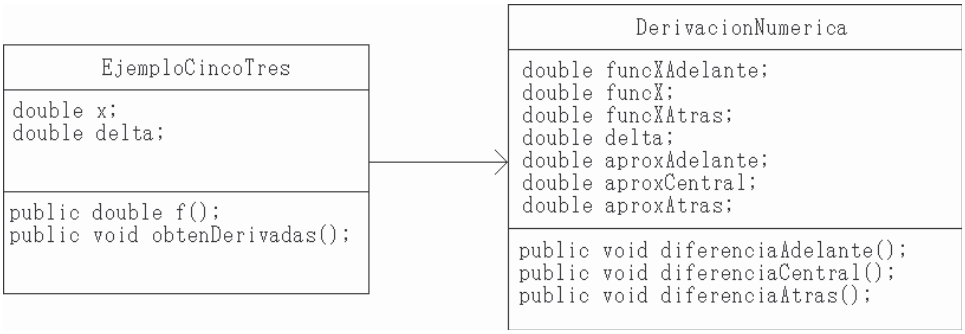


FIGURA 5.4 Diagrama de clase del ejemplo

-Código `DerivacionNumerica`

```
public class DerivacionNumerica{
   double funcXAdelante, funcX, funcXAtras, delta,
          aproxAdelante, aproxAtras, aproxCentral;

   public void diferenciaAdelante(){
      this.aproxAdelante=(this.funcXAdelante –
                          this.funcX)/this.delta;
   }

   public void diferenciaAtras(){
      this.aproxAtras=(this.funcX –
                       this.funcXAtras)/this.delta;
   }

   public void diferenciaCentral(){
      this.aproxCentral=(this.funcXAdelante –
                         this.funcXAtras)/(2*this.delta);
   }
}
```

-Explicación del código

Del diagrama de clase, se obtiene el código base del programa

```
public class DerivacionNumerica{
   double funcXAdelante, funcX, funcXAtras, delta,
          aproxAdelante, aproxAtras, aproxCentral;
```

```
   public void diferenciaAdelante(){
   }

   public void diferenciaAtras(){
   }

   public void diferenciaCentral(){
   }
}
```

Ahora se escriben las fórmulas, dentro del método correspondiente

```
public void diferenciaAdelante(){
   this.aproxAdelante=(this.funcXAdelante –
                       this.funcX)/this.delta;
}

public void diferenciaAtras(){
   this.aproxAtras=(this.funcX –
                    this.funcXAtras)/this.delta;
}

public void diferenciaCentral(){
   this.aproxCentral=(this.funcXAdelante –
                      this.funcXAtras)/(2*this.delta);
}
```

-Código `EjemploCincoTres`

```
public class EjemploCincoTres{
   double x,delta;

   public double funcion(double x){
      double res;
      res = Math.pow(x,3)-(8*Math.pow(x,2))+(15*x)+3;
      return res;
   }


   public void obtenDerivadas(){
      DerivacionNumerica dn = new DerivacionNumerica();
      dn.delta = this.delta;
      dn.funcXAdelante = this.funcion(this.x+this.delta);
```

```
      dn.funcX = this.funcion(this.x);
      dn.funcXAtras = this.funcion(this.x-this.delta);
      dn.diferenciaAdelante();
      dn.diferenciaAtras();
      dn.diferenciaCentral();

      System.out.println("La derivada en el punto: "+
                         this.x+"\n");
      System.out.println("utilizando diferencias hacia "+
                        "adelante: "+dn.aproxAdelante);
      System.out.println("utilizando diferencias hacia "+
                        "atras:"+dn.aproxAtras);
      System.out.println("utilizando diferencias"+
                        " centrales: "+dn.aproxCentral);
   }

   public static void main(String args[]){
      EjemploCincoTres ect = new EjemploCincoTres();
      ect.x = Double.parseDouble(args[0]);
      ect.delta = Double.parseDouble(args[1]);
      ect.obtenDerivadas();
   }
}
```

-Explicación del código

Del diagrama de clase, el código base del programa es:

```
public class EjemploCincoTres{
   double x,delta;

   public double funcion(){
   }

   public void obtenDerivadas(){
   }
}
```

Dentro del método `funcion()`, se implementa la función a resolver

```
public double funcion(double x){
   double res;
   res = Math.pow(x,3)-(8*Math.pow(x,2))+(15*x)+3;
   return res;
}
```

Dentro del método `obtenDerivadas()`, primeramente se crea un objeto de la clase `DerivacionNumerica`.

```
DerivacionNumerica dn = new DerivacionNumerica();
```

Después se asignan valores iniciales a los atributos del objeto, copiando el valor de `delta` de la clase, en el objeto creado, así como la asignación del valor de la función en el punto, y en los puntos a `delta` de distancia del punto central.

```
dn.delta = this.delta;
dn.funcXAdelante = this.funcion(this.x+this.delta);
dn.funcX = this.funcion(this.x);
dn.funcXAtras = this.funcion(this.x-this.delta);
```

Luego se mandan llamar las funciones del objeto `dn`.

```
dn.diferenciaAdelante();
dn.diferenciaAtras();
dn.diferenciaCentral();
```

Por último se muestran en pantalla los valores obtenidos

```
 System.out.println("La derivada en el punto: "+
                    this.x+"\n");
 System.out.println("utilizando diferencias hacia "+
                    "adelante: "+dn.aproxAdelante);
 System.out.println("utilizando diferencias hacia "+
                    "atras:"+dn.aproxAtras);
 System.out.println("utilizando diferencias"+
                    " centrales: "+dn.aproxCentral);
```

En virtud de que se requiere un punto de entrada a la clase, se define el método `main`, el cual recibirá desde línea de comandos los valores de `x` y `delta`.

```
public static void main(String args[]){
   EjemploCincoTres ect = new EjemploCincoTres();
   ect.x = Double.parseDouble(args[0]);
   ect.delta = Double.parseDouble(args[1]);
   ect.obtenDerivadas();
}
```

5.1.2 Integración numérica

Por definición, la integral de una función es el área bajo la curva. Por lo que, cuando se desea obtener numéricamente el valor aproximado de la integral de una función en un intervalo dado, se puede hacer sumando áreas por subintervalos. Existen varios métodos de integración numérica; en esta sección se describirán la Regla de Simpson 1/3 y la Regla de Simpson 3/8.

Regla de Simpson $\frac{1}{3}$

Este método aproxima el valor de la integral de una función obteniendo el valor del área bajo una función cuadrática sobre dos intervalos del mismo ancho. Por lo que suponiendo que la función a integrar es una cuadrática, la función es de la forma:

$$f(x) = Ax^2 + Bx + C$$

Por simplicidad en los cálculos, se supondrá que el punto medio del intervalo a integrar sea $(0, y_{i+1})$, además el intervalo será simétrico respecto a este punto, por lo que los dos puntos que definirán los extremos del intervalo serán $(-\Delta x, y_i)$ y $(\Delta x, y_{i+2})$.

Sustituyendo estos 3 puntos en la ecuación, genera el siguiente sistema de ecuaciones:

$$\begin{aligned} y_i &= A(\Delta x)^2 - B(\Delta x) + C \\ y_{i+1} &= C \\ y_{i+2} &= A(\Delta x)^2 + B(\Delta x) + C \end{aligned}$$

Cuya solución es:

$$\begin{aligned} A &= \frac{y_i - 2y_{i+1} + y_{i+2}}{2(\Delta x)^2} \\ B &= \frac{y_{i+2} - y_i}{2\Delta x} \\ C &= y_{i+1} \end{aligned}$$

Por otro lado, integrando en el intervalo $(-\Delta x, \Delta x)$, se tiene que:

$$\int_{-\Delta x}^{\Delta x} f(x)dx = \int_{-\Delta x}^{\Delta x} (Ax^2 + Bx + C)dx = \left[\frac{Ax^3}{3} + \frac{Bx^2}{2} + Cx\right]_{-\Delta x}^{\Delta x}$$
$$= \left[\frac{A(\Delta x)^3}{3} + \frac{B(\Delta x)^2}{2} + C(\Delta x)\right] - \left[\frac{A(-\Delta x)^3}{3} + \frac{B(-\Delta x)^2}{2} + C(-\Delta x)\right]$$
$$= \frac{2A(\Delta x)^3}{3} + 2C(\Delta x)$$

Sustituyendo los valores de las constantes A, B y C, en la solución de la integral, se tiene que:

$$\int_{-\Delta x}^{\Delta x} f(x)dx = \frac{2A(\Delta x)^3}{3} + 2C(\Delta x)$$
$$= \frac{2\left[\frac{y_i - 2y_{i+1} + y_{i+2}}{2(\Delta x)^2}\right](\Delta x)^3}{3} + 2[y_{i+1}](\Delta x)$$
$$= \frac{\Delta x}{3}[y_i + 4y_{i+1} + y_{i+2}]$$

Con base en este razonamiento de sumar el área calculada por 3 puntos de la curva, si se tienen 5 puntos, se dividen en 2 grupos de 3 puntos, donde tendrían como punto en común el punto medio. De esta forma se sumarían 2 áreas. Utilizando la fórmula calculada y los cinco puntos, se tiene:

$$A_{0,1,2} + A_{2,3,4} = \frac{\Delta x}{3}[y_0 + 4y_1 + y_2] + \frac{\Delta x}{3}[y_2 + 4y_3 + y_4]$$
$$= \frac{\Delta x}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + y_4]$$

Suponiendo 7 puntos:

$$A_{0,1,2} + A_{2,3,4} + A_{4,5,6} = \frac{\Delta x}{3}[y_0 + 4y_1 + y_2] + \frac{\Delta x}{3}[y_2 + 4y_3 + y_4] + \frac{\Delta x}{3}[y_4 + 4y_5 + y_6]$$
$$= \frac{\Delta x}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + 4y_5 + y_6]$$

Analizando las ecuaciones obtenidas para 3, 5 y 7 puntos, se obtiene la fórmula general:

$$S_N = \frac{\Delta x}{3}\left(y_0 + 4\sum_{i=1,3,5,\ldots}^{N-1} y_i + 2\sum_{i=2,4,6,\ldots}^{N-2} y_i + y_N\right)$$

Regla de Simpson $\frac{3}{8}$

Para esta regla, en lugar de utilizar un polinomio de segundo orden, ahora se utiliza uno de tercero. Su formula general es:

$$S_{\frac{3}{8}N} = \frac{3\Delta x}{8}\left(y_0 + 2\sum_{i=3,6,9,..}^{N-1} y_i + 3\sum_{i\neq \text{múltiplos de } 3}^{N-2} y_i + y_N \right)$$

EJEMPLO 5.4 UTILIZANDO LA REGLA DE SIMPSON DE $\frac{1}{3}$, OBTENER LA INTEGRAL DE $f(x) = 3x^2 - 2x - 8$ EN EL INTERVALO $(0,10)$. DIVIDIR EL INTERVALO PARA 3, 5 Y 7 PUNTOS. COMPARAR RESULTADOS.

-Solución

Para 3 puntos, el Δx es de 5; se calculan los valores de la función:

x	f(x)
0	-8
5	57
10	272

De la fórmula general:

$$S_3 = \frac{\Delta x}{3}(y_0 + 4y_1 + y_2) = \frac{5}{3}(-8 + 4(57) + 272) = \frac{5(492)}{3} = 820$$

Para 5 puntos, el Δx es de 2.5; se obtienen los valores de la función:

x	f(x)
0	-8
2.5	5.75
5	57
7.5	145.75
10	272

De la fórmula general:

$$S_5 = \frac{\Delta x}{3}(y_0 + 4(y_1 + y_3) + 2(y_2) + y_5) = \frac{2.5}{3}(-8 + 4(5.75 + 145.75) + 2(57) + 272) = 820$$

Para 7 puntos, el Δx es de 1.667, se obtienen los valores de la función:

x	f(x)
0	-8
1.667	-3
3.333	18.667
5	57
6.667	112
8.333	183.667
10	272

De la fórmula general:

$$S_7 = \frac{\Delta x}{3}\left(y_0 + 4\left(y_1 + y_3 + y_5\right) + 2\left(y_2 + y_4\right) + y_6\right)$$
$$= \frac{1.667}{3}\left(-8 + 4\left(-3 + 57 + 183.667\right) + 2\left(18.667 + 112\right) + 272\right) = 820$$

Como se puede observar en este ejemplo, el resultado no varía independientemente del número de puntos que se utilicen, esto es debido a que la función dentro del intervalo se comporta como si fuera una recta.

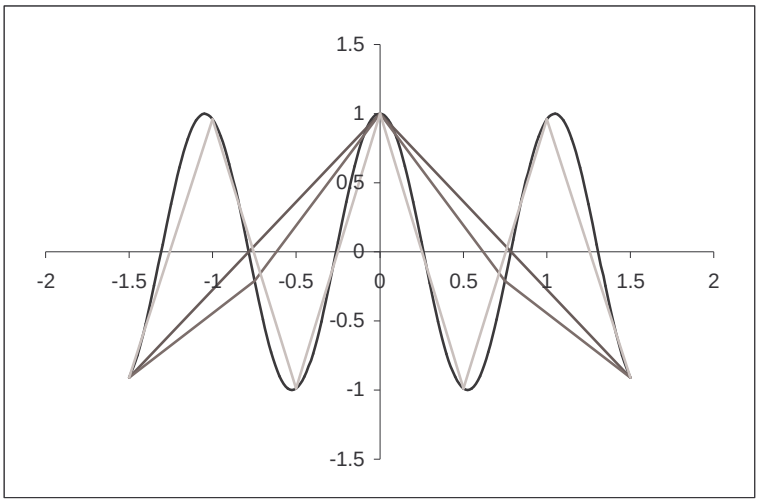


FIGURA 5.5 Gráfica de la función a integrar del ejemplo 5.4, junto con las líneas del área que se calcula dependiendo del número de puntos a utilizar.

EJEMPLO 5.5 UTILIZANDO LA REGLA DE SIMPSON DE $\frac{1}{3}$, OBTENER LA INTEGRAL DE $f(x) = \cos(6x)$, EN EL INTERVALO $(-1.5,1.5)$. DIVIDA EL INTERVALO PARA 3, 5 Y 7 PUNTOS. COMPARAR RESULTADOS.

-Solución

Para 3 puntos, el Δx es de 1.5; se obtienen los valores de la función:

x	f(x)
-1.5	-0.91113026
0	1
1.5	-0.91113026

De la fórmula general:

$$S_3 = \frac{\Delta x}{3}(y_0 + 4y_1 + y_2) = \frac{1.5}{3}(-0.91113026 + 4(1) - 0.91113026) = 1.0889$$

Para 5 puntos, el Δx es de 0.75; se obtienen los valores de la función:

x	f(x)
-1.5	-0.91113026
-0.75	-0.2107958
0	1
0.75	-0.2107958
1.5	-0.91113026

De la fórmula general:

$$S_5 = \frac{\Delta x}{3}(y_0 + 4(y_1 + y_3) + 2(y_2) + y_5)$$
$$= \frac{0.75}{3}(-0.91113026 + 4(-02107958 - 02107958) + 2(1) - 0.91113026) = -0.37715$$

Para 7 puntos, el Δx es de 0.5; se obtienen los valores de la función:

x	f(x)
-1.5	-0.91113026
-1	0.96017029
-0.5	-0.9899925
0	1
0.5	-0.9899925
1	0.96017029
1.5	-0.91113026

Resolviendo por fórmula general:

$$S_7 = 0.9832$$

Como se puede observar, los resultados arrojados en esta ocasión, son distintos. Resolviendo analíticamente la integral y sustituyendo los valores de los límites, el área bajo la curva es de 0.137372828. Por lo que para acercarse al valor real se deben utilizar más puntos. Empleando el método con 85 puntos, el resultado se aproxima mucho ya que el valor que arroja es 0.13737445.

EJEMPLO 5.6 IMPLEMENTAR, EN UN PROGRAMA ORIENTADO A OBJETOS, LA REGLA DE SIMPSON DE $\frac{1}{3}$. EN OTRO PROGRAMA, OBTENER LA INTEGRAL DE

$$f(x) = 3x^2 - 2x - 8 \text{ EN EL INTERVALO } (0,10).$$

EL NÚMERO DE PUNTOS DEBERÁ SER INDICADO AL EJECUTAR EL PROGRAMA.

-Análisis y solución del problema

Se identifican dos clases, en una clase nombrada ***SimpsonUnTercio***, se deberá implementar el método de la integración por Simpson. En otra clase, nombrada ***EjemploCincoSeis*** se implementará la función a integrar.

Para la clase ***SimpsonUnTercio***, se necesitarán como atributos, un arreglo que almacene los valores de la función en los puntos que se utilizarán para integrar, el número de puntos en los que se dividirá el intervalo, el valor de ***h***, las sumas de los elementos pares e impares, y el valor de la integral calculada. Como métodos, se implementará un método para cada una de las sumas, y otro método para obtener el valor de la integral.

En la clase ***EjemploCincoSeis***, se requieren como atributos los límites del intervalo, así como el número de puntos en los que se subdividirá el intervalo. Como métodos la función a integrar, y un método que cree un objeto de ***SimpsonUnTercio***, y usando ese objeto obtenga la integral de la función.

-Diagrama de clases

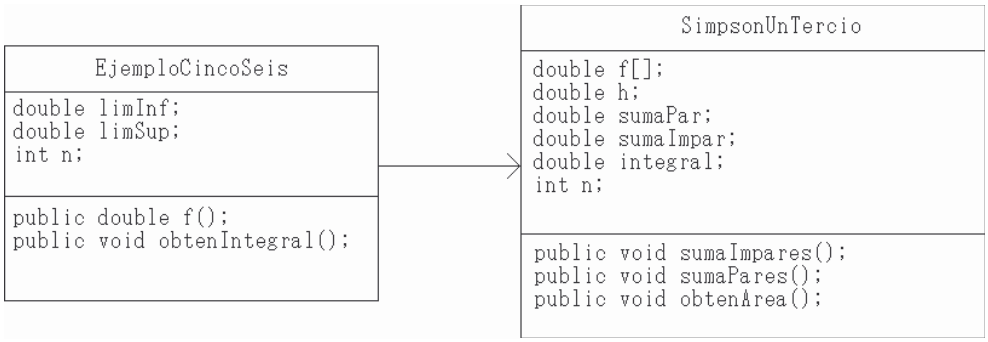


FIGURA 5.6 Diagrama de clases del ejemplo 5.6

-Código SimpsonUnTercio

```
public class SimpsonUnTercio{
   double fx[];
   double h, sumaPar, sumaImpar, integral;
   int n;
   public SimpsonUnTercio(int partes){
      this.fx = new double[partes];
      this.n = partes;
   }
   public void sumaImpares(){
      for(int i=1;i<(this.n-1);i++){
         if(i%2==0)
            continue;
         this.sumaImpar=this.sumaImpar+this.fx[i];
      }
   }
   public void sumaPares(){
      for(int i=2;i<(this.n-2);i++){
         if(i%2!=0)
            continue;
         this.sumaPar=this.sumaPar+this.fx[i];
      }
   }
   public void obtenArea(){
      this.sumaImpares();
      this.sumaPares();
      this.integral = (this.h/3)*(this.fx[0]+
                     (4*this.sumaImpar)+(2*this.sumaPar)+
                      this.fx[this.n-1]);
   }
}
```

-Explicación del código

Del diagrama de clases, el código base de SimpsonUnTercio es:

```
public class SimpsonUnTercio{
   double fx[];
   double h, sumaPar, sumaImpar, integral;
   int n;

   public void sumaImpares(){
   }
   public void sumaPares(){
   }
   public void obtenArea(){
   }
}
```

Se implementa la sumatoria para los elementos impares, utilizando el entero *n* como el tope para el ciclo.

```
public void sumaImpares(){
   for(int i=1;i<(this.n-1);i++){
      if(i%2==0)
         continue;
      this.sumaImpar=this.sumaImpar+this.fx[i];
   }
}
```

De igual forma, cambiando el ciclo ahora para hacer la suma de los pares

```
public void sumaPares(){
   for(int i=2;i<(this.n-2);i++){
      if(i%2!=0)
         continue;
      this.sumaPar=this.sumaPar+this.fx[i];
   }
}
```

Ahora, para el método `obtenArea()`, se debe mandar llamar los métodos anteriores, y luego aplicar la fórmula para la regla de Simpson de $\frac{1}{3}$

```
public void obtenArea(){
   this.sumaImpares();
   this.sumaPares();
   this.integral = (this.h/3) * (this.fx[0] +
                   (4*this.sumaImpar) + (2*this.sumaPar) +
                   this.fx[this.n-1]);
}
```

Como este método está condicionado al número de puntos que se utilizarán, se debe agregar un método constructor, que inicialice el arreglo con los elementos necesarios, mismos que definirá el usuario.

```
public SimpsonUnTercio(int partes){
   this.fx = new double[partes];
   this.n = partes;
}
```

-Código `EjemploCincoSeis`

```
public class EjemploCincoSeis{
   double limInf, limSup;
   int n;
```

```
   public double f(double x){
      double res;
      res = (3*Math.pow(x,2))-(2*x)-8;
      return res;
   }

   public void obtenIntegral(){
      SimpsonUnTercio sut = new SimpsonUnTercio(this.n);
      sut.h = (this.limSup-this.limInf)/(this.n-1);
      for(int i = 0;i<n;i++){
         sut.fx[i]=this.f(this.limInf+(i*sut.h));
      }

      sut.obtenArea();

      System.out.println("La integral en el intervalo: ("+
                         this.limInf+","+this.limSup+")\n");
      System.out.println(sut.integral);
   }

   public static void main(String args[]){
      EjemploCincoSeis ecs = new EjemploCincoSeis();
      ecs.limInf = Double.parseDouble(args[0]);
      ecs.limSup = Double.parseDouble(args[1]);
      ecs.n = Integer.parseInt(args[2]);
      ecs.obtenIntegral();
   }
}
```

-Explicación del código

Con base en el diagrama de clases se obtiene el código base del programa

```
public class EjemploCincoSeis{
   double limInf, limSup;
   int n;

   public double f(){
   }

   public void obtenIntegral(){
   }
}
```

Se implementa la función a resolver

```
public double f(double x){
```

```
   double res;
   res = (3*Math.pow(x,2))-(2*x)-8;
   return res;
}
```

Para obtener el valor del área bajo la curva de la función en el intervalo, dentro del método que resolverá este problema se deberá crear un objeto de `SimpsonUnTercio,` y sabiendo que el constructor debe llevar como argumento el número de puntos, se crea de la siguiente manera.

```
SimpsonUnTercio sut = new SimpsonUnTercio(this.n);
```

Con base en el número de puntos, se calcula el valor de `h`.

```
sut.h = (this.limSup-this.limInf)/(this.n-1);
```

En un ciclo, se obtienen los valores de la función en los puntos que se utilizarán para integrarla.

```
for(int i = 0;i<n;i++){
   sut.fx[i]=this.f(this.limInf+(i*sut.h));
}
```

Una vez obtenidos dichos valores, se puede obtener el área bajo la curva, por lo que se manda llamar a la función del objeto `sut`.

```
sut.obtenArea();
```

Y luego se manda desplegar el resultado en pantalla

```
System.out.println("La integral en el intervalo:("+
                   this.limInf+","+this.limSup+")\n");
System.out.println(sut.integral);
```

Sólo falta crear un punto de entrada a la clase, es decir un método `main`.

```
public static void main(String args[]){
   EjemploCincoSeis ecs = new EjemploCincoSeis();
   ecs.limInf = Double.parseDouble(args[0]);
   ecs.limSup = Double.parseDouble(args[1]);
   ecs.n = Integer.parseInt(args[2]);
   ecs.obtenIntegral();
}
```

5.2 Solución numérica de ecuaciones y sistemas de ecuaciones diferenciales.

Como se comentó en el capítulo 2, cuando se modelan sistemas se pueden obtener sistemas de ecuaciones, o se puede obtener una función que pueda simular un sistema. Pero no siempre es así, por ejemplo cuando se modelan sistemas físicos, como pueden ser sistemas mecánicos donde se involucran derivadas, la función que modela el sistema es una ecuación diferencial. Existen diversos métodos numéricos que ayudan a obtener la solución de dichas ecuaciones.

5.2.1 Solución de ecuaciones diferenciales de primer orden

Antes de resolver ecuaciones diferenciales más complicadas, primero se resolverán ecuaciones diferenciales de primer orden.

- Método de Euler

A partir del significado geométrico de la derivada, que es la pendiente de la recta tangente en un punto sobre la curva, si se toma otro punto bastante cercano del que se tiene la derivada, la recta formada por dichos puntos utilizando como pendiente la derivada, se puede asumir que dicha recta se aproxima a la función original.

Con esta idea, se obtiene la ecuación de la recta tangente que pasa por el punto (x_0, y_0), que es donde se tiene el valor de la derivada. Primero, de la ecuación general de una recta:

$$y = mx + b$$

Sustituyendo el punto (x_0, y_0) en la ecuación general se tiene:

$$y_0 = mx_0 + b$$

Despejando b:

$$b = y_0 - mx_0$$

Sustituyendo en la ecuación general:

$$y = mx + y_0 - mx_0$$
$$y = m(x - x_0) + y_0$$

Pero se sabe que el valor de la pendiente es el valor de la derivada en dicho punto, obteniendo finalmente:

$$y = f'(x_0, y_0)(x - x_0) + y_0$$

Se mencionó que tomando dos puntos lo más cercanos posible, se puede aproximar a la curva $f(x)$, por lo que el siguiente valor se obtendría sumando un Δx a x_0, tomando ese incremento como h, se tiene:

$x_1 = x_0 + h$, este nuevo punto se sustituye en $y = f'(x_0, y_0)(x - x_0) + y_0$, y se obtiene:

$$y_1 = f'(x_0, y_0)((x_0 + h) - x_0) + y_0$$
$$y_1 = hf'(x_0, y_0) + y_0$$

Como se puede observar, este método se vuelve iterativo, por lo que las fórmulas generales son:

$$x_{n+1} = x_n + h$$
$$y_{n+1} = hf'(x_n, y_n) + y_n$$

Para que este método obtenga la mejor aproximación el valor de h deberá ser lo más pequeño posible.

El resultado de aplicar este método, es la obtención del valor aproximado de la función solución a la ecuación diferencial en un punto deseado, partiendo de las condiciones iniciales de la ecuación diferencial.

EJEMPLO 5.7 UTILIZANDO EL MÉTODO DE EULER, OBTENER LA SOLUCIÓN DE LA ECUACIÓN DIFERENCIAL

$$\frac{dy}{dx} = y' = 2x + y$$

CON CONDICION INICIAL DE $y(0) = 1$ EN EL PUNTO $x = 1$. UTILIZAR UN VALOR h DE 0.2 Y 0.1. COMPARAR RESULTADOS.

-Solución

Inicialmente $x_0 = 0$ y $y_0 = 1$, con base en $h = 0.2$, se empieza a iterar:

$$x_1 = x_0 + h = 0 + 0.2 = 0.2$$
$$y_1 = hf(x_0, y_0) + y_0 = h(2x_0 + y_0) + y_0 = (0.2)(2(0) + 1) + 1 = 0.2 + 1 = 1.2$$

$$x_2 = x_1 + h = 0.2 + 0.2 = 0.4$$
$$y_2 = hf(x_1, y_1) + y_1 = h(2x_1 + y_1) + y_1 = (0.2)(2(0.2) + 1.2) + 1.2 = 0.32 + 1.2 = 1.52$$

$$x_3 = x_2 + h = 0.4 + 0.2 = 0.6$$
$$y_3 = hf(x_2, y_2) + y_2 = h(2x_2 + y_2) + y_2 = (0.2)(2(0.4) + 1.52) + 1.52 = 0.464 + 1.52 = 1.984$$

$x_4 = x_3 + h = 0.6 + 0.2 = 0.8$
$y_4 = hf(x_3, y_3) + y_3 = h(2x_3 + y_3) + y_3 = (0.2)(2(0.6) + 1.984) + 1.984 = 0.6368 + 1.984 = 2.6208$

$x_5 = x_4 + h = 0.8 + 0.2 = 1.0$
$y_5 = hf(x_4, y_4) + y_4 = h(2x_4 + y_4) + y_4 = (0.2)(2(0.8) + 2.6208) + 2.6208 = 3.46496$

Inicialmente $x_0 = 0$ y $y_0 = 1$, con base en $h = 0.1$, se empieza a iterar:
$x_1 = x_0 + h = 0 + 0.1 = 0.1$
$y_1 = hf(x_0, y_0) + y_0 = h(2x_0 + y_0) + y_0 = (0.1)(2(0) + 1) + 1 = 0.1 + 1 = 1.1$

$x_2 = x_1 + h = 0.1 + 0.1 = 0.2$
$y_2 = hf(x_1, y_1) + y_1 = h(2x_1 + y_1) + y_1 = (0.1)(2(0.1) + 1.1) + 1.1 = 0.13 + 1.1 = 1.23$

$x_3 = x_2 + h = 0.2 + 0.1 = 0.3$
$y_3 = hf(x_2, y_2) + y_2 = h(2x_2 + y_2) + y_2 = (0.1)(2(0.2) + 1.23) + 1.23 = 0.163 + 1.23 = 1.393$

$x_4 = x_3 + h = 0.3 + 0.1 = 0.4$
$y_4 = hf(x_3, y_3) + y_3 = h(2x_3 + y_3) + y_3 = (0.1)(2(0.3) + 1.393) + 1.393 = 0.1993 + 1.393 = 1.5923$

$x_5 = x_4 + h = 0.4 + 0.1 = 0.5$
$y_5 = hf(x_4, y_4) + y_4 = h(2x_4 + y_4) + y_4 = (0.1)(2(0.4) + 1.5923) + 1.5923 = 1.83153$

$x_6 = x_5 + h = 0.5 + 0.1 = 0.6$
$y_6 = hf(x_5, y_5) + y_5 = h(2x_5 + y_5) + y_5 = (0.1)(2(0.5) + 1.83153) + 1.83153 = 2.114683$

$x_7 = x_6 + h = 0.6 + 0.1 = 0.7$
$y_7 = hf(x_6, y_6) + y_6 = h(2x_6 + y_6) + y_6 = (0.1)(2(0.6) + 2.114683) + 2.114683 = 2.4461513$

$x_8 = x_7 + h = 0.7 + 0.1 = 0.8$
$y_8 = hf(x_7, y_7) + y_7 = h(2x_7 + y_7) + y_7 = (0.1)(2(0.7) + 2.4461513) + 2.4461513 = 2.83076643$

$x_9 = x_8 + h = 0.8 + 0.1 = 0.9$
$y_9 = hf(x_8, y_8) + y_8 = h(2x_8 + y_8) + y_8 = (0.1)(2(0.8) + 2.83076643) + 2.83076643 = 3.27384307$

$x_{10} = x_9 + h = 0.9 + 0.1 = 1.0$
$y_{10} = hf(x_9, y_9) + y_9 = h(2x_9 + y_9) + y_9 = (0.1)(2(0.9) + 3.27394307) + 3.27394307 = 3.78122738$

Como se puede observar, si se resuelve analíticamente la función general que resuelve la ecuación diferencial es: $y_G = C_1 e^x + C_2 x + C_3$. De acuerdo a la condición inicial, se llega a la función solución: $y = 3e^x - 2x - 2$. Por lo tanto el valor de la función en el punto $x = 1$ es 4.15484549.

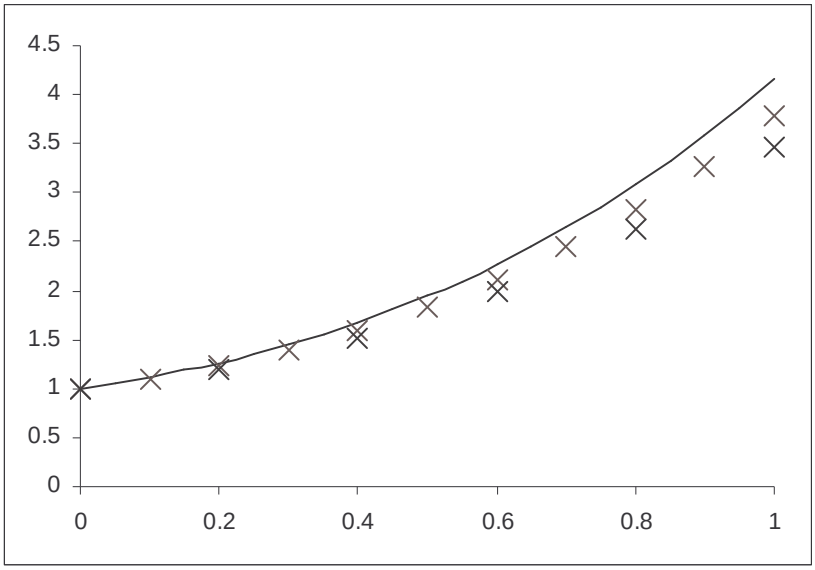


FIGURA 5.7 Gráfica de la función solución de la ecuación diferencial del ejemplo 5.5, comparando con los resultados arrojados por el método de Euler.

EJEMPLO 5.8 IMPLEMENTAR EN UNA CLASE, EL MÉTODO DE EULER. UTILIZANDO OTRA CLASE, OBTENER LA SOLUCIÓN DE LA ECUACIÓN DIFERENCIAL

$$\frac{dy}{dx} = y' = 2x + y$$

CON CONDICION INICIAL DE $y(0) = 1$ EN EL PUNTO $x = 1$. UTILIZAR EL VALOR h DE 0.01.

NOTA: UTILIZANDO LA CLASE *BIGDECIMAL* GENERAR UNA CLASE QUE PERMITA HACER REDONDEO.

-Análisis y solución del problema

Se crearán tres clases, cada una con una función en particular.

Una clase con la implementación del método de Euler, la cual deberá tener dos arreglos de dos elementos para almacenar tanto la iteración anterior como la nueva iteración, posteriormente se cambiarán los valores de la nueva iteración a la anterior; también un atributo que almacene el valor de *h*, y un objeto de la clase que se genere para hacer el redondeo. Como métodos se requiere de uno que atrase la iteración para poder calcular la siguiente, y otra donde se implemente la fórmula que se utiliza en cada iteración del método.

En la segunda clase, se implementará la ecuación diferencial a resolver y un método que obtenga la solución, haciendo uso del método de Euler.

La tercera clase, realizará el redondeo de los valores, para que se pueda obtener un valor exacto de *x* en cada iteración.

-Diagrama de clases

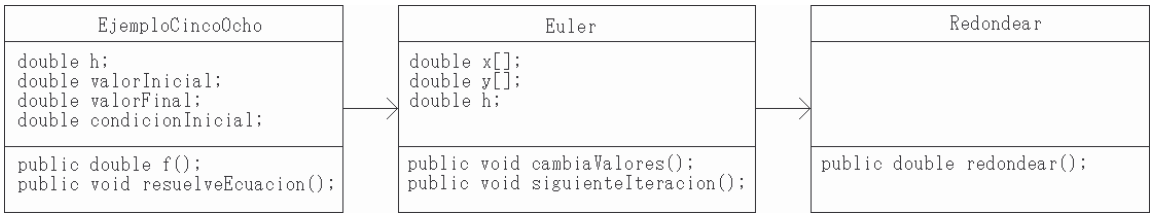


FIGURA 5.8 Gráfica de clases del ejemplo 5.8

-Código `Redondear`

```
import java.math.*;

public class Redondear{
   public double redondear(double num, int decimales){
      BigDecimal bd = new BigDecimal(Double.toString(num));
      bd = bd.setScale(decimales, BigDecimal.ROUND_HALF_UP);
      return bd.doubleValue();
   }
}
```

-Explicación del código

Como se va a utilizar la clase `BigDecimal`, que se encuentra en el paquete `java.math`, se debe importar dicho paquete.

```
import java.math.*;
```

Del diagrama de clases, el código base del programa es:

```
public class Redondear{
   public double redondear(){
   }
}
```

El método deberá recibir dos parámetros, el número que se redondeará, y a cuántos decimales se redondeará

```
public double redondear(double num, int decimales){
}
```

Dentro del método, lo primero que se debe hacer es crear un objeto de `BigDecimal`, mismo que permitirá hacer el redondeo del número.

```
BigDecimal bd = new BigDecimal(Double.toString(num));
```

Ahora, se modifica dicho número, y debido a que el método `setScale()` devuelve otro `BigDecimal`, se debe asignar al mismo objeto creado. El método recibe dos parámetros, los decimales a los que se redondeará, y el tipo de redondeo que en este caso se utilizará el `ROUND_HALF_UP`.

```
bd = bd.setScale(decimales, BigDecimal.ROUND_HALF_UP);
```

Luego se debe devolver el valor del número redondeado, con el método `doubleValue()`

```
return bd.doubleValue();
```

-Código `Euler`

```
public class Euler{
   double x[] = new double[2];
   double y[] = new double[2];
   double h;
   Redondear r = new Redondear();
   public void cambiaValores(){
      this.x[0] = this.x[1];
      this.y[0] = this.y[1];
   }
   public void siguienteIteracion(double fxy){
      this.x[1] = r.redondear(this.x[0]+this.h,2);
      this.y[1] = this.y[0]+(this.h*fxy);
   }

   public Euler(double x, double y,double h){
      this.x[1] = x;
      this.y[1] = y;
      this.h = h;
   }
}
```

-Explicación del código

Del diagrama de clases, se obtiene la base del programa

```
public class Euler{
   double x[] = new double[2];
   double y[] = new double[2];
```

```
   double h;
   public void cambiaValores(){
   }
   public void siguienteIteracion(){
   }
}
```

Como se requiere un objeto que permita redondear los números, se crea un objeto a partir de la clase `Redondear`

```
public class Euler{
   double x[] = new double[2];
   double y[] = new double[2];
   double h;
   Redondear r = new Redondear();

   public void cambiaValores(){
   }
   public void siguienteIteracion(){
   }
}
```

Ahora, el método `cambiaValores()`, se utilizará para colocar los valores x[1] y y[1] en x[0] y y[0], respectivamente. Esto con la finalidad de "atrasar" la iteración y poder calcular la siguiente.

```
public void cambiaValores(){
   this.x[0] = this.x[1];
   this.y[0] = this.y[1];
}
```

Ahora, el método `siguienteIteracion()`, deberá recibir como parámetro el valor de la función con los puntos de la iteración anterior. Y se implementan las fórmulas para realizar las iteraciones del método. Es en este método donde se utilizará el objeto `Redondear`, debido a que Java, al realizar las sumas, las hace aproximadas. Por ejemplo, si se suma 0.1 a 1, el resultado es 1.09999; por ello se requiere redondear.

```
public void siguienteIteracion(double fxy){
   this.x[1] = r.redondear(this.x[0]+this.h,2);
   this.y[1] = this.y[0]+(this.h*fxy);
}
```

En este caso el redondeo se hará a dos decimales, porque se utiliza un h de 0.01.

Ahora, al crear el objeto, es conveniente inicializarlo con los primeros valores, indicando que son la iteración actual, por lo mismo se almacenan en x[1] y y[1].

```
public Euler(double x, double y,double h){
   this.x[1] = x;
   this.y[1] = y;
   this.h = h;
}
```

-Código `EjemploCincoOcho`

```
public class EjemploCincoOcho{
   double h,valorInicial,valorFinal,condicionInicial;

   public double f(double x, double y){
      double res;
      res = (2*x)+y;
      return res;
   }

   public void resuelveEcuacion(){
      Euler e = new Euler(this.valorInicial,
                          this.condicionInicial,this.h);
      while(e.x[1]<this.valorFinal){
         e.cambiaValores();
         e.siguienteIteracion(this.f(e.x[0],e.y[0]));
      }

      System.out.println("El valor de la funcion solucion "+
                         "en el punto: "+e.x[1]);
      System.out.println("es: "+e.y[1]);
   }

   public static void main(String args[]){
      try{
         EjemploCincoOcho eco = new EjemploCincoOcho();
         eco.valorInicial = Double.parseDouble(args[0]);
         eco.valorFinal = Double.parseDouble(args[1]);
         eco.condicionInicial = Double.parseDouble(args[2]);
         eco.h = Double.parseDouble(args[3]);
         eco.resuelveEcuacion();
      }
      catch(ArrayIndexOutOfBoundsException aioobe){
         System.out.println("ERROR!!!");
         System.out.println("Forma de ejecucion:");
         System.out.println("java EjemploCincoOcho "+
                            "valorInicial valor Final "+
                            "condicionInicial h");
      }
```

```
   }
}
```

-Explicación del código

Del diagrama de clases, se obtiene el código base para el programa

```
public class EjemploCincoOcho{
   double h,valorInicial,valorFinal,condicionInicial;

   public double f(){
   }

   public void resuelveEcuacion(){
   }
}
```

Se implementa la función a resolver, en el método `f()`. Debido a que es una función que depende de dos variables, ambas las deberá recibir como parámetros.

```
public double f(double x, double y){
   double res;
   res = (2*x)+y;
   return res;
}
```

Dentro del método `resuelveEcuacion()`, se requiere crear un objeto de la clase `Euler`, y usando dicho objeto, se entra en un ciclo donde se cambian los valores para obtener la siguiente iteración, y esto se repetirá mientras la *x* de la iteración actual sea menor que valorFinal.

```
public void resuelveEcuacion(){
   Euler e = new Euler(this.valorInicial,
                       this.condicionInicial,this.h);
   while(e.x[1]<this.valorFinal){
      e.cambiaValores();
      e.siguienteIteracion(this.f(e.x[0],e.y[0]));
   }
}
```

Ahora se agregan las sentencias, dentro del mismo método, para desplegar en pantalla el resultado.

```
System.out.println("El valor de la funcion solucion "+
                   "en el punto: "+e.x[1]);
```

```
System.out.println("es: "+e.y[1]);
```

Por último se escribe el método `main`, de tal forma que se pueda ejecutar el programa. Debido a que requiere recibir cuatro parámetros desde línea de comandos, se utilizará la sentencia try-catch para indicar el orden en el que se deben acomodar los parámetros

```
public static void main(String args[]){
   try{
      EjemploCincoOcho eco = new EjemploCincoOcho();
      eco.valorInicial = Double.parseDouble(args[0]);
      eco.valorFinal = Double.parseDouble(args[1]);
      eco.condicionInicial = Double.parseDouble(args[2]);
      eco.h = Double.parseDouble(args[3]);
      eco.resuelveEcuacion();
   }
   catch(ArrayIndexOutOfBoundsException aioobe){
      System.out.println("ERROR!!!");
      System.out.println("Forma de ejecucion:");
      System.out.println("java EjemploCincoOcho "+
                         "valorInicial valor Final "+
                         "condicionInicial h");
   }
}
```

- Método de Runge-Kutta

Los matemáticos alemanes Runge y Kutta definieron un grupo de métodos de solución numérica de ecuaciones diferenciales más eficientes que los métodos de Euler y la serie de Taylor. Estos métodos se basan en el cálculo del promedio en el cambio de *y* cuando *x* avanza *h* unidades sobre la curva solución. El método más conocido es el de Runge-Kutta de cuatro pasos o RK4, éste es un método iterativo. El desarrollo de esta técnica es complicado algebraicamente, por lo que se describirá de forma muy concreta.

Primeramente se obtienen las constantes k_i, que son las estimaciones del cambio en *y*:

$$k_1 = f(x_n, y_n)$$
$$k_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right)$$
$$k_3 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right)$$
$$k_4 = f(x_n + h, y_n + hk_3)$$

Donde h es el incremento en la x_n

En cada iteración se deben calcular las constantes usando las ecuaciones anteriores y los siguientes puntos de x_n y y_n, usando las siguientes ecuaciones:

$$x_{n+1} = x_n + h$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

EJEMPLO 5.9 UTILIZANDO EL MÉTODO DE RUNGE-KUTTA DE CUARTO ORDEN, OBTENER LA SOLUCIÓN DE LA ECUACIÓN DIFERENCIAL

$$\frac{dy}{dx} = y' = 2x + y$$

CON CONDICION INICIAL DE $y(0) = 1$ EN EL PUNTO $x = 1$. UTILIZAR UN INCREMENTO h DE 0.2 Y 0.1. COMPARAR RESULTADOS.

-Solución

Se inicia con:

$x_0 = 0$

$y_0 = 1$

Iteración 1

Cálculo de las constantes *k*.

$$k_{11} = f(x_0, y_0) = 2x_0 + y_0 = 2(0) + 1 = 1$$

$$k_{21} = f\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}k_{11}\right) = 2\left(0 + \frac{0.2}{2}\right) + \left(1 + \frac{0.2}{2}(1)\right) = 1.3$$

$$k_{31} = f\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}k_{21}\right) = 2\left(0 + \frac{0.2}{2}\right) + \left(1 + \frac{0.2}{2}(1.3)\right) = 1.33$$

$$k_{41} = f(x_0 + h, y_0 + hk_{31}) = 2(0 + 0.2) + (1 + 1.33) = 1.666$$

Obtención de nuevos valores

$$x_1 = x_0 + h = 0 + 0.2 = 0.2$$

$$y_1 = y_0 + \frac{h}{6}(k_{11} + 2k_{21} + 2k_{31} + k_{41}) = 1 + \frac{0.2}{6}(1 + 2(1.3) + 2(1.33) + 1.666) = 1.2642$$

Iteración 2

Cálculo de las constantes k.

$$k_{12} = f(x_1, y_1) = 2x_1 + y_1 = 2(0.2) + 1.2642 = 1.6642$$
$$k_{22} = f\left(x_1 + \frac{h}{2}, y_1 + \frac{h}{2}k_{12}\right) = 2\left(0.2 + \frac{0.2}{2}\right) + \left(1.2642 + \frac{0.2}{2}(1.6642)\right) = 2.03062$$
$$k_{32} = f\left(x_1 + \frac{h}{2}, y_1 + \frac{h}{2}k_{22}\right) = 2\left(0.2 + \frac{0.2}{2}\right) + \left(1.2642 + \frac{0.2}{2}(2.03062)\right) = 2.067262$$
$$k_{42} = f(x_1 + h, y_1 + hk_{32}) = 2(0.2 + 0.2) + (1.2642 + 2.067262) = 2.4776524$$

Obtención de nuevos valores

$$x_2 = x_1 + h = 0.2 + 0.2 = 0.4$$
$$y_2 = y_1 + \frac{h}{6}(k_{12} + 2k_{22} + 2k_{32} + k_{42}) = 1.2642 + \frac{0.2}{6}(1.6642 + 2(2.0306) + 2(2.0672) + 2.4776) = 1.6754$$

Después de varias iteraciones, y ordenando todos los valores en una tabla:

x	y	k_1	k_2	k_3	k_4
0	1				
0.2	1.2642	1	1.3	1.33	1.666
0.4	1.67545	1.6642	2.0306	2.0673	2.47765
0.6	2.2663	2.47545	2.923	2.96775	3.469
0.8	3.0766	3.4663	4.013	4.0676	4.6798
1.0	4.15475	4.6766	5.3442	5.411	6.15876

Utilizando un incremento h de 0.1, y repitiendo todos estos pasos, quedan los siguientes valores:

x	y	k_1	k_2	k_3	k_4
0	1	1	1.15	1.1575	1.31575
0.1	1.1155	1.3155	1.4813	1.4896	1.6645
0.2	1.2642	1.6642	1.8474	1.8566	2.0499
0.3	1.4496	2.0496	2.252	2.2622	2.4758
0.4	1.6755	2.4755	2.6992	2.7104	2.9465
0.5	1.9462	2.9462	3.1934	3.2058	3.4667
0.6	2.26635	3.46635	3.7397	3.7533	4.0417
0.7	2.64125	4.04125	4.3433	4.3584	4.6771
0.8	3.0766	4.6766	5.0104	5.0271	5.3793
0.9	3.5788	5.3788	5.7477	5.7662	6.1554
1.0	4.1548	6.1548	6.5626	6.583	7.0131

Como se puede observar, con ambos valores de *h*, el resultado es casi el mismo, y comparándolo con el valor real de 4.15484549, se aproxima todavía más el resultado

obtenido con h = 0.1. Lo que quiere decir que entre más pequeño sea el incremento más aproximado el resultado será.

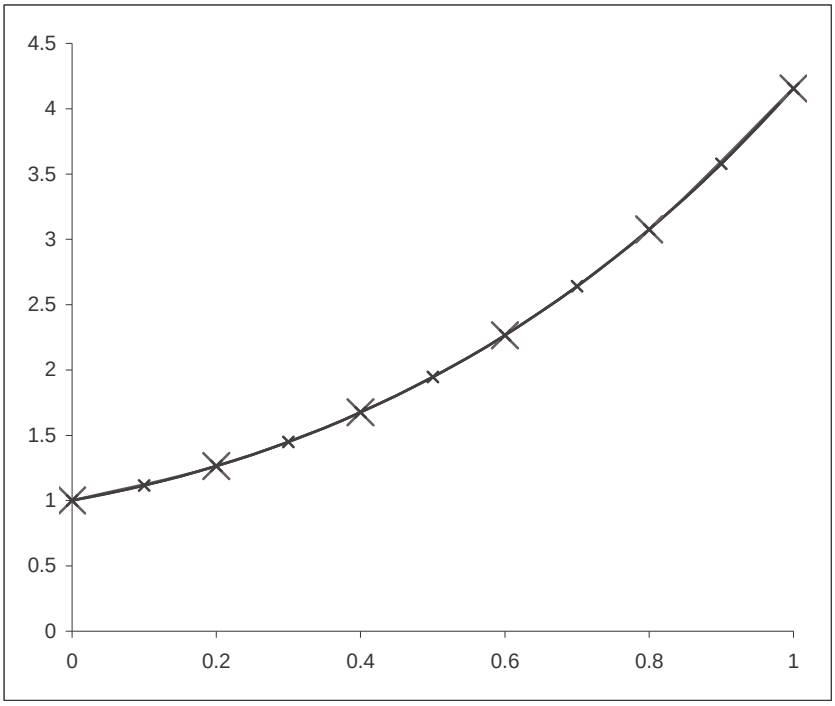


FIGURA 5.9 Gráfica de la función solución de la ecuación diferencial del ejemplo 5.6, comparando con los resultados arrojados por el método de Runge-Kutta.

Comparando el método de Euler con Runge-Kutta, se puede concluir que el método de Runge-Kutta es más aproximado.

5.2.2 Solución de ecuaciones diferenciales de orden *n*

Hasta este momento, sólo se ha mencionado la forma en que se resuelve una ecuación diferencial de primer orden. ¿Pero qué pasa cuando la ecuación diferencial es de orden mayor a uno?

La solución radica en transformar la ecuación diferencial en un sistema de ecuaciones, haciendo cambios de variable.

Suponiendo que se tiene la siguiente ecuación diferencial:

$$y^{(n)} = f\left(x, y, y^{|}, y^{||}, \ldots, y^{(n-1)}\right)$$

A partir de esta ecuación se genera un sistema de ***n-1*** ecuaciones, de la siguiente forma:

$$y' = y_1$$
$$y'_1 = y_2$$
$$y'_2 = y_3$$
$$\vdots$$
$$y'_{n-2} = y_{n-1}$$
$$y'_{n-1} = f(x, y, y_1, y_2, \ldots, y_{n-1})$$

Por lo que las condiciones iniciales quedan de la siguiente forma:

$$y(x_0) = y_0$$
$$y'(x_0) = y_1(x_0) = y'_0$$
$$y''(x_0) = y_2(x_0) = y''_0$$
$$\vdots$$
$$y^{(n-1)}(x_0) = y_{n-1}(x_0) = y^{(n-1)}{}_0$$

Y se procede a resolver este sistema por alguno de los métodos numéricos, en este caso Euler o Runge-Kutta.

EJEMPLO 5.10 UTILIZANDO EL MÉTODO DE EULER, OBTENER LA SOLUCIÓN DE LA ECUACIÓN DIFERENCIAL $y'''-4y''-5y'=0$ CON CONDICIONES INICIALES

$$y(0) = 13$$
$$y'(0) = 11$$
$$y''(0) = 79$$

EN EL PUNTO $x = 0.5$. UTILICE UN INCREMENTO h DE 0.1.

- Solución

Primero se transforma la ecuación diferencial en un sistema de ecuaciones de la siguiente forma:

$$y' = y_1$$
$$y'_1 = y'' = y_2$$
$$y'_2 = y''' = 4y''+5y' = 4y_2 + 5y_1$$

Es decir:

$$y' = y_1$$
$$y'_1 = y_2$$
$$y'_2 = 4y_2 + 5y_1$$

Las condiciones iniciales se reordenan de la siguiente manera:

$y(0) = y_0 = 13$
$y'(0) = y_1(0) = y_{10} = 11$
$y''(0) = y_2(0) = y_{20} = 79$

Comienza la solución utilizando el método de Euler.

Iteración 1:
$x_1 = x_0 + h = 0 + 0.1 = 0.1$
$y_1 = y_0 + hf_1(x_0, y_0, y_{10}, y_{20}) = 13 + 0.1(y_{10}) = 13 + 0.1(11) = 14.1$
$y_{11} = y_{10} + hf_2(x_0, y_0, y_{10}, y_{20}) = 11 + 0.1(y_{20}) = 11 + 0.1(79) = 18.9$
$y_{21} = y_{20} + hf_3(x_0, y_0, y_{10}, y_{20}) = 79 + 0.1(4y_{20} + 5y_{10}) = 79 + 0.1(4*79 + 5*11) = 116.1$

Iteración 2:
$x_2 = x_1 + h = 0.1 + 0.1 = 0.2$
$y_2 = y_1 + hf_1(x_1, y_1, y_{11}, y_{21}) = 14.1 + 0.1(y_{11}) = 14.1 + 0.1(18.9) = 15.99$
$y_{12} = y_{11} + hf_2(x_1, y_1, y_{11}, y_{21}) = 18.9 + 0.1(y_{21}) = 18.9 + 0.1(116.1) = 30.51$
$y_{22} = y_{21} + hf_3(x_1, y_1, y_{11}, y_{21}) = 116.1 + 0.1(4y_{21} + 5y_{11}) = 116.1 + 0.1(4*116.1 + 5*18.9) = 171.99$

Iteración 3:
$x_3 = x_2 + h = 0.2 + 0.1 = 0.3$
$y_3 = y_2 + hf_1(x_2, y_2, y_{12}, y_{22}) = 15.99 + 0.1(30.51) = 19.041$
$y_{13} = y_{12} + hf_2(x_2, y_2, y_{12}, y_{22}) = 30.51 + 0.1(171.99) = 47.709$
$y_{23} = y_{22} + hf_3(x_2, y_2, y_{12}, y_{22}) = 171.99 + 0.1(4*171.99 + 5*30.51) = 256.041$

Iteración 4:
$x_4 = x_3 + h = 0.3 + 0.1 = 0.4$
$y_4 = y_3 + hf_1(x_3, y_3, y_{13}, y_{23}) = 19.041 + 0.1(47.709) = 23.8119$
$y_{14} = y_{13} + hf_2(x_3, y_3, y_{13}, y_{23}) = 47.709 + 0.1(256.041) = 73.3131$
$y_{24} = y_{23} + hf_3(x_3, y_3, y_{13}, y_{23}) = 256.041 + 0.1(4*256.041 + 5*47.709) = 382.3119$

Iteración 5:
$x_5 = x_4 + h = 0.4 + 0.1 = 0.5$
$y_5 = y_4 + hf_1(x_4, y_4, y_{14}, y_{24}) = 23.8119 + 0.1(73.3131) = 31.14321$
$y_{15} = y_{14} + hf_2(x_4, y_4, y_{14}, y_{24}) = 73.3131 + 0.1(382.3119) = 111.54429$
$y_{25} = y_{24} + hf_3(x_4, y_4, y_{14}, y_{24}) = 382.3119 + 0.1(4*382.3119 + 5*73.3131) = 571.89321$

Al resolver numéricamente este problema, el resultado se obtiene en la *y* que no tiene 2 subíndices, que como se puede observar dio como resultado 31.14321. Este valor no es muy aproximado al valor real, que en este caso, tendría que ser $y(0.5) = 44.9736$.

Si se vuelve a resolver este método, pero utilizando una h de 0.001, después de 500 iteraciones, el valor encontrado es de 44.746, que ya es más aproximado al valor real.

Con esto se puede reafirmar, que entre más pequeño sea el incremento, se harán más iteraciones, pero el resultado será más aproximado.

5.2.3 Solución de sistemas de ecuaciones diferenciales

A diferencia de la resolución de una ecuación diferencial de orden mayor a uno, que se resuelve transformando en un sistema de ecuaciones; aquí se tiene un sistema de ecuaciones, donde todas las ecuaciones dependen de todas las variables, y se desea encontrar la solución de todas las funciones solución en un punto dado. Por ejemplo, un sistema de ecuaciones general sería:

$$y'_1 = f_1(x, y_1, y_2, \ldots, y_n)$$
$$y'_2 = f_2(x, y_1, y_2, \ldots, y_n)$$
$$\vdots$$
$$y'_n = f_n(x, y_1, y_2, \ldots, y_n)$$

Cuyas condiciones iniciales son:

$$y_1(x_0) = y_{10}$$
$$y_2(x_0) = y_{20}$$
$$\vdots$$
$$y_n(x_0) = y_{n0}$$

Como se puede observar el sistema es de orden uno. Y se procede a resolver por alguno de los métodos numéricos.

EJEMPLO 5.11 UTILIZANDO EL MÉTODO DE EULER, OBTENER LA SOLUCIÓN DEL SISTEMA DE ECUACIONES DIFERENCIALES

$$y'_1 = -0.5y_1$$
$$y'_2 = 4 - 0.3y_2 - 0.1y_1$$

CON CONDICIONES INICIALES

$$y_1(0) = 4$$
$$y_2(0) = 6$$

EN EL PUNTO $x = 2$. UTILICE UN INCREMENTO h DE 0.5.

- Solución

Se comienza a resolver por el método de Euler.

Iteración 1:

$$x_1 = x_0 + h = 0 + 0.5 = 0.5$$
$$y_{11} = y_{10} + hf_1(x_0, y_{10}, y_{20}) = 4 + 0.5(-0.5y_{10}) = 4 + 0.5(-0.5(4)) = 3$$
$$y_{21} = y_{20} + hf_2(x_0, y_{10}, y_{20}) = 6 + 0.5(4 - 0.3y_{20} - 0.1y_{10}) = 6 + 0.5(4 - 0.3(6) - 0.1(4)) = 6.9$$

Iteración 2:

$$x_2 = x_1 + h = 0.5 + 0.5 = 1$$
$$y_{12} = y_{11} + hf_1(x_1, y_{11}, y_{21}) = 3 + 0.5(-0.5y_{11}) = 3 + 0.5(-0.5(3)) = 2.25$$
$$y_{22} = y_{21} + hf_2(x_1, y_{11}, y_{21}) = 6 + 0.5(4 - 0.3y_{21} - 0.1y_{11}) = 6.9 + 0.5(4 - 0.3(6.9) - 0.1(3)) = 7.715$$

Iteración 3:

$$x_3 = x_2 + h = 1 + 0.5 = 1.5$$
$$y_{13} = y_{12} + hf_1(x_2, y_{12}, y_{22}) = 2.25 + 0.5(-0.5(2.25)) = 1.6875$$
$$y_{23} = y_{22} + hf_2(x_2, y_{12}, y_{22}) = 7.715 + 0.5(4 - 0.3(7.715) - 0.1(2.25)) = 8.44525$$

Iteración 4:

$$x_4 = x_3 + h = 1.5 + 0.5 = 2$$
$$y_{14} = y_{13} + hf_1(x_3, y_{13}, y_{23}) = 1.6875 + 0.5(-0.5(1.6875)) = 1.265625$$
$$y_{24} = y_{23} + hf_2(x_3, y_{13}, y_{23}) = 8.44525 + 0.5(4 - 0.3(8.44525) - 0.1(1.6875)) = 9.094088$$

5.3 Solución de ecuaciones en derivadas parciales.

Una ecuación en derivadas parciales es aquella ecuación en la que la función derivada, depende de dos o más variables, y su orden está determinado por el grado máximo de la derivada que se encuentre, por ejemplo:

$$\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = 0 \qquad Orden\_1$$
$$\frac{\partial u}{\partial y} + \frac{\partial^2 u}{\partial x^2} + \frac{\partial u}{\partial x} = 0 \qquad Orden\_2$$
$$\frac{\partial u}{\partial x} - \frac{\partial^2 u}{\partial x \partial y} + \frac{\partial u}{\partial y} = 0 \qquad Orden\_2$$
$$\frac{\partial^2 u}{\partial y^2} + \frac{\partial^3 u}{\partial x^2 \partial y} - \frac{\partial u}{\partial x} = 0 \qquad Orden\_3$$

Las ecuaciones en derivadas parciales de segundo orden se clasifican de tres formas de acuerdo a la siguiente ecuación general:

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + D = 0$$

Dependiendo de los valores de las constantes, su clasificación es:

$B^2 - 4AC < 0$	*Elíptica*
$B^2 - 4AC = 0$	*Parabólica*
$B^2 - 4AC > 0$	*Hiperbólica*

El objeto de reconocer el tipo de ecuación que se tiene, sirve para elegir un método para su solución. Además cada tipo de ecuación está asociado a un problema de ingeniería.

- Ecuaciones elípticas

Este tipo de ecuaciones están asociadas a problemas que tienen que ver con equilibrio, es decir que se está sujeto a condiciones de frontera dadas. Los ejemplos más comunes de esto son: distribuciones de temperatura sobre algún cuerpo, flujo de fluidos incompresibles; en términos generales, este tipo de ecuaciones son para encontrar potenciales.

- Ecuaciones parabólicas

Este tipo de ecuaciones están asociadas, a problemas que tienen que ver con propagación, ya que son problemas de transitorios en los que la solución está sujeta a condiciones iniciales y condiciones de frontera. Es decir, cualquier problema en el que el estado cambia con el tiempo.

- Ecuaciones hiperbólicas

Este tipo de ecuaciones, aunque trata con problemas de propagación, aparece una segunda derivada respecto del tiempo. Con esto, la solución consiste en diversos estados en los que el sistema está oscilando. Este tipo de problemas tiene que ver con vibraciones, es decir, ondas de un fluido, transmisión de señales, etc.

Solución de ecuaciones diferenciales parciales elípticas

En este apartado se analizará una solución en particular de las ecuaciones elípticas, utilizando una ecuación llamada ecuación de Laplace. Su representación matemática es:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Antes de empezar con el método para la resolución de este tipo de problemas, hay que hacer notar que al tener las segundas derivadas respecto a una sola variable, esta ecuación se puede tratar como un caso particular en el que se requiere obtener la segunda derivada de una función, por lo que se transforma cada término en una aproximación por Taylor de la segunda derivada, es decir:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{(i,j)} = \frac{u_{(i+1,j)} - 2u_{(i,j)} + u_{(i-1,j)}}{(\Delta x)^2}$$

$$\left.\frac{\partial^2 u}{\partial y^2}\right|_{(i,j)} = \frac{u_{(i,j+1)} - 2u_{(i,j)} + u_{(i,j-1)}}{(\Delta y)^2}$$

Donde los subíndices nos indican la variación de las variables, en este caso *i* es para indicar el valor de la variable *x* y *j* es para la *y*.

Sustituyendo esos valores en la ecuación general, se tiene que:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{u_{(i+1,j)} - 2u_{(i,j)} + u_{(i-1,j)}}{(\Delta x)^2} + \frac{u_{(i,j+1)} - 2u_{(i,j)} + u_{(i,j-1)}}{(\Delta y)^2} = 0$$

Con esta ecuación se pueden calcular, por ejemplo, las temperaturas de una placa.

EJEMPLO 5.12 SE TIENE UNA PLACA DE 25X20 [CM], Y SE MANTIENEN LOS 4 LADOS A DIFERENTES TEMPERATURAS. CALCULAR LAS TEMPERATURAS EN DIVERSOS PUNTOS DE LA PLACA, SEPARADOS 2.5 [CM] ENTRE SÍ, COMO SE MUESTRA EN LA SIGUIENTE FIGURA.

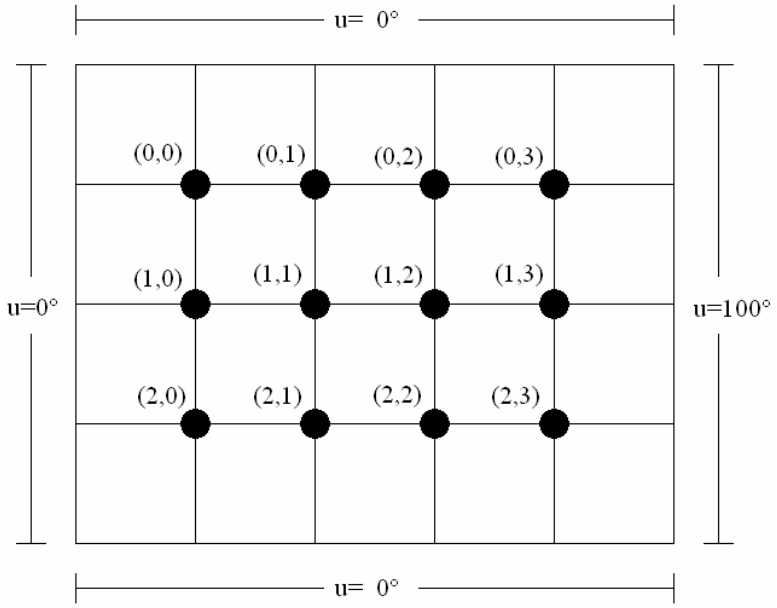


FIGURA 5.10 Gráfica del ejemplo 5.12

- Solución

De la ecuación de Laplace, utilizando las aproximaciones de Taylor, se tiene:

$$\frac{u_{(i+1,j)} - 2u_{(i,j)} + u_{(i-1,j)}}{(\Delta x)^2} + \frac{u_{(i,j+1)} - 2u_{(i,j)} + u_{(i,j-1)}}{(\Delta y)^2} = 0$$

Debido a que se tiene la misma separación entre muestras sobre *x* y *y*, se puede suponer que $\Delta x = \Delta y$. Por lo que la ecuación queda de la siguiente forma:

$$u_{(i+1,j)} + u_{(i-1,j)} + u_{(i,j+1)} + u_{(i,j-1)} - 4u_{(i,j)} = 0$$

De esta forma, para calcular la temperatura en el punto (0,0), se tiene que:

$$u_{(1,0)} + u_{(-1,0)} + u_{(0,1)} + u_{(0,-1)} - 4u_{(0,0)} = 0$$
$$u_{(1,0)} + 0 + u_{(0,1)} + 0 - 4u_{(0,0)} = 0$$

Para el punto (0,3):

$$u_{(1,3)} + u_{(-1,3)} + u_{(0,4)} + u_{(0,2)} - 4u_{(0,3)} = 0$$
$$u_{(1,3)} + 0 + 100 + u_{(0,2)} - 4u_{(0,3)} = 0$$

Para el punto (1,3):

$$u_{(2,3)} + u_{(0,3)} + u_{(1,4)} + u_{(1,2)} - 4u_{(1,3)} = 0$$
$$u_{(2,3)} + u_{(0,3)} + 100 + u_{(1,2)} - 4u_{(1,3)} = 0$$

Después de obtener las ecuaciones de todos los puntos, al ordenarlos queda un sistema de ecuaciones de 12 ecuaciones con 12 incógnitas de la siguiente forma:

u_{00}	u_{01}	u_{02}	u_{03}	u_{10}	u_{11}	u_{12}	u_{13}	u_{20}	u_{21}	u_{22}	u_{23}		b
-4	1			1								=	0
1	-4	1			1							=	0
	1	-4	1			1						=	0
		1	-4				1					=	-100
1				-4	1			1				=	0
	1			1	-4	1			1			=	0
		1			1	-4	1			1		=	0
			1			1	-4				1	=	-100
				1				-4	1			=	0
					1			1	-4	1		=	0
						1			1	-4	1	=	0
							1			1	-4	=	-100

Que, resuelto por Gauss-Jordan, la solución es:

u_{00}	u_{01}	u_{02}	u_{03}	u_{10}	u_{11}	u_{12}	u_{13}	u_{20}	u_{21}	u_{22}	u_{23}		b
1												=	3.346963
	1											=	8.708163
		1										=	19.460851
			1									=	43.131909
				1								=	4.679691
					1							=	12.024837
						1						=	26.003332
							1					=	53.066788
								1				=	3.346963
									1			=	8.708163
										1		=	19.460851
											1	=	43.131909

Y se puede observar que las u_{0i} y u_{2i} son idénticas, debido a la distribución del calor en la placa.

Ejercicios Propuestos

1) Utilizando la clase DerivacionNumerica del ejemplo 5.3, obtener la derivada de $f(x)=e^{\sin(x)}$ en el punto $x=15$, utilizando un Δx de 0.05.

2) Utilizando la clase DerivacionNumerica del ejemplo 5.3, obtener la derivada de $f(x)=\sin\left(e^{\cos\left(x^3\right)}\right)$ en el punto $x=1$, utilizando un Δx de 0.1, 0.05 y 0.01. Comparar resultados.

3) Implementar en una clase la derivación de segundo orden. Usando dicha clase, obtener el valor de la segunda derivada de $f(x)=x^5-6x^3+8x-3$ en el punto $x=1$ con un Δx de 0.1, 0.05 y 0.01. Comparar resultados.

4) Utilizando la clase SimpsonUnTercio del ejemplo 5.6, obtener la integral de $f(x)=12x^3+8x$ en el intervalo $[0,6]$ para 3 y 5 puntos. Comparar resultados.

5) Implementar en una clase el método de Simpson de 3/8.

6) Con la clase obtenida en el ejercicio anterior, calcular la integral del ejemplo 5.5 con 9, 11 y 13 puntos. Comparar resultados.

7) Usando la clase del ejercicio 5, calcular la integral del ejercicio 4, con 9 y 11 puntos. Comparar resultados.

8) Utilizando la clase de Euler del ejemplo 5.8, resolver la siguiente ecuación diferencial $y'=y-6e^{-2x}$ con condición inicial de $y(0)=5$, en el punto $x=1$. Utilizar un valor *h* de 0.1

9) Implementar en una clase el método de Runge-Kutta para solución de ecuaciones diferenciales de primer orden.

10) Con la clase del inciso 9, resolver la ecuación diferencial del inciso 8, con el mismo valor de *h*. Comparar resultados.

11) Con la clase del inciso 9, resolver la ecuación diferencial $y'=4y-4$, con condición inicial $y(0)=4$ en el punto $x=1$. Utilizar un valor *h* de 0.001.

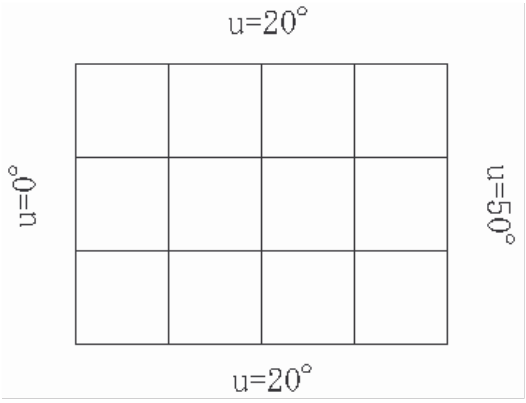
12) Implementar en una clase o modificar la clase Euler, para resolver por el método de Euler la ecuación diferencial de tercer orden $y'''+6y''+3y'-10y=10x+7$, con condiciones iniciales $y(0)=1 \quad y'(0)=10 \quad y''(0)=-43$ en el punto $x=1$. Utilizar un valor *h* de 0.01.

13) Implementar en una clase o modificar la clase del inciso 9, para resolver la ecuación diferencial $y'''-2y''-y'+2y=16x-18$, con condiciones iniciales $y(0)=0 \quad y'(0)=14 \quad y''(0)=14$ en el punto $x=1$. Utilizar un valor *h* de 0.01.

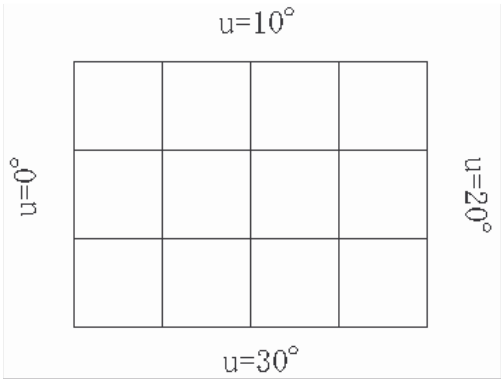
14) Utilizando el método de Euler y Runge-Kutta, resolver el siguiente sistema de ecuaciones $\begin{matrix} y'_1 = y_1 + y_2 \\ y'_2 = 2y_2 \\ y'_3 = y_3 \end{matrix}$, con condiciones iniciales $\begin{matrix} y_1(0)=2 \\ y_2(0)=1 \\ y_3(0)=1 \end{matrix}$ en el punto $x=1$. Utilizar un valor *h* de 0.001. Comparar resultados.

15) Utilizando el método de Euler y Runge-Kutta, resolver el siguiente sistema de ecuaciones $\begin{matrix} y'_1 = 2y_1 + y_3 \\ y'_2 = 2y_2 + \frac{1}{2}y_3 \\ y'_3 = 3y_3 \end{matrix}$, con condiciones iniciales $\begin{matrix} y_1(0)=3 \\ y_2(0)=3 \\ y_3(0)=2 \end{matrix}$ en el punto $x=1$. Utilizar un valor *h* de 0.001. Comparar resultados.

16) Resolver por el método de Lagrange para ecuaciones en derivadas parciales, el problema para la temperatura en una placa plana. Considerar $\Delta x = \Delta y = 1[cm]$



17) Resolver por el método de Lagrange para ecuaciones en derivadas parciales, el problema para la temperatura en una placa plana. Considerar $\Delta x = \Delta y = 10[cm]$



CAPÍTULO 6

PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

Introducción

La eficiencia de un programa radica en varios factores, uno de ellos es el uso adecuado del procesador, evitando tiempos muertos, y así aumentar la velocidad del programa que propicia una respuesta más rápida en entrega de resultados. En este capítulo se abordarán los temas que permitirán al programador aprovechar los recursos tanto del procesador como de la memoria para obtener un programa orientado a objetos eficiente. Asimismo se describirá cómo realizar las operaciones básicas de entrada y salida.

6.1 Multihilos

Antes de abordar el tema de hilos, es conveniente primero explicar la forma en que trabaja una CPU (Unidad Central de Procesamiento); esto con la finalidad de comprender el concepto de hilo.

Concepto de Proceso

En una computadora, cuando se ejecuta un programa o una aplicación en específico, el sistema operativo crea un proceso; este proceso es una copia del programa que se va a ejecutar la cual está almacena en la memoria principal. El proceso está conformado por dos partes: la memoria de programa y la memoria de datos.

En la actualidad, los sistemas operativos pueden atender varios procesos. Por ejemplo, si se desea ejecutar un procesador de textos, un explorador para páginas Web y una hoja de cálculo; por cada programa, se genera un proceso, dichos procesos competirán por entrar al procesador para ejecutarse.

La ejecución de los procesos que han ingresado al sistema puede ser de varias formas. Si los procesos sólo se ejecutan uno tras otro, es decir, que sólo puede entrar a ejecución un proceso y mientras no termine no se ejecuta otro, se le llama procesamiento por lotes (batch); de esta forma sólo se trabaja con una aplicación a la vez.

Existen los sistemas multitareas, los cuales asignan cierta cantidad de tiempo, a veces llamado tiempo de procesador, a los procesos para que puedan estar en ejecución; si no terminan de ejecutarse en ese tiempo tienen que esperar su turno para entrar nuevamente a ejecución; así es como el procesador atiende varias tareas o procesos de manera "simultánea" y el uso del procesador se vuelve más eficiente.

Bajo el esquema de un sistema multitareas, los procesos tienen tres diferentes estados mientras existan; por lo tanto, las etapas de la vida de un proceso son:

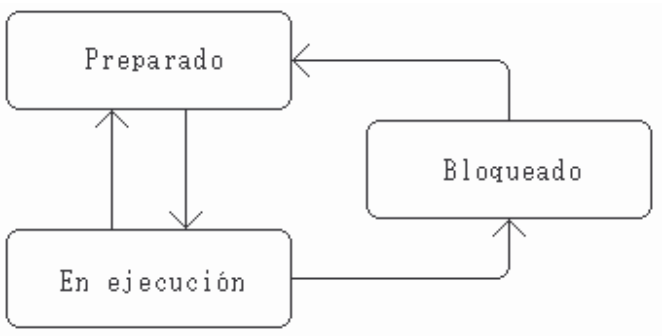


FIGURA 6.1 Diagrama de estados de los procesos

- Preparado: Este estado es cuando el proceso está listo para entrar al procesador a ejecutarse.

- En ejecución: El proceso se está ejecutando. Si el proceso no ha terminado de ejecutarse y ha terminado su tiempo dentro del procesador, pasa al estado Preparado. Si el proceso tiene un problema por el cual no puede seguirse ejecutando, pasa al estado Bloqueado.

- Bloqueado: En este estado, el proceso está en espera de que se atienda una operación que no requiere del procesador. En el momento en que se termina dicha operación, el proceso regresa al estado Preparado.

6.1.1 Hilos

Los hilos son "procesos ligeros", ya que se crean a partir de una aplicación. Así, por medio de hilos, una aplicación o programa puede fraccionarse en procesos ligeros cuya ejecución se hace "simultáneamente" aumentando su tiempo de ejecución.

Cuando se ejecuta un programa, se crea un proceso denominado primer hilo o hilo principal, del cual, se pueden generar más hilos; éstos compartirán el mismo espacio de direcciones, lo que significa que comparten las mismas variables globales, archivos abiertos, etc.

Dado que los hilos son creados por un proceso, y se comportan de la misma forma que un proceso, tienen los mismos estados: Preparado, En ejecución y Bloqueado.

Hilos en Java

En Java, los hilos se crean a través de la clase llamada *Thread*, de la cual se heredan métodos como *run()*, que contiene el código que ejecutará el hilo. Este método heredado se tiene que sobrescribir con instrucciones propias del hilo para que realice sus funciones. Esto se muestra en el siguiente ejemplo:

EJEMPLO 6.1 CREAR UNA CLASE QUE GENERE UN HILO; ÉSTE HARÁ UN CONTEO DEL 1 AL 10 EN ORDEN ASCENDENTE.

-Análisis y solución del problema

Se requiere una clase que sea subclase de *Thread*, la cual contendrá dentro del método *run()*, el código para el conteo del 1 al 10.

-Diagrama de clase

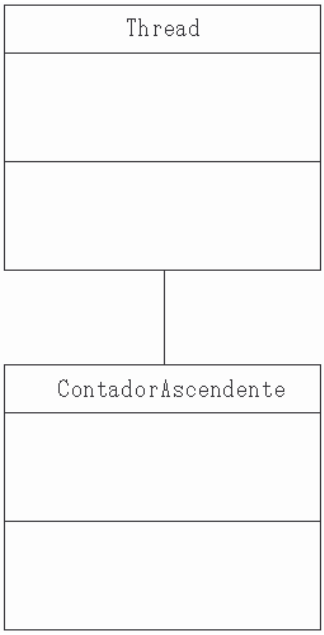


FIGURA 6.2 Diagrama de clase del ejemplo 6.1

Debido a que la clase Thread proporciona el método `run`() entre otros, en el diagrama de clase no es necesario redefinirlo dentro de `ContadorAscendente`.

-Código

```
public class ContadorAscendente extends Thread{
   public void run(){
```

```
      for(int i = 1;i<=10;i++){
         System.out.println(i);
      }
   }

   public static void main(String args[]){
      ContadorAscendente ca = new ContadorAscendente();
      ca.start();
   }
}
```

-Explicación del código

El código base del programa es:

```
public class ContadorAscendente extends Thread{
}
```

Hay que recodar que con la palabra `extends` se crea una subclase de la siguiente clase, en este caso, `ContadorAscendente` será una subclase de `Thread`.

Ahora se sobrescribe el método `run()`, utilizando un ciclo `for` para el conteo.

```
public void run(){
   for(int i = 1;i<=10;i++){
      System.out.println(i);
   }
}
```

En la función `main`, se crea un objeto de la clase, y con el método `start()`, se comienza la ejecución del hilo.

```
public static void main(String args[]){
   ContadorAscendente ca = new ContadorAscendente();
   ca.start();
}
```

6.1.2Multihilos

Existen aplicaciones que realizan varias tareas a la vez; si se considera que una tarea es aquella que sólo realiza una función específica dentro de una aplicación, entonces se puede referir a ella como un *hilo de ejecución*.

Es común que la implementación de las tareas se realice por medio de Multihilos, es decir, utilizar varios hilos simultáneamente coordinados por un proceso principal. En la sección

anterior se describió cómo crear un solo hilo de ejecución; ahora se mostrará, con un ejemplo, cómo crear dos hilos que trabajen a la vez.

EJEMPLO 6.2 CREAR UNA CLASE QUE GENERE DOS HILOS. ÉSTOS HARÁN UN CONTEO DEL 1 AL 10 EN ORDEN ASCENDENTE. UTILICE EL MÉTODO SLEEP( ) PARA DORMIR UN TIEMPO ALEATORIO LOS HILOS.

-Análisis y solución del problema

Se requiere una clase heredada de *Thread* que creará dos hilos, cada hilo ejecutará el mismo código; éste se especificará al sobrescribir el método *run( )*. Para distinguir un hilo del otro se les asignarán nombres, los cuales se mostrarán al momento de realizar el conteo. Debido a que esta clase manejará dos hilos "contadores", se nombrará ContadoresAscendentes.
-Diagrama de clases

FIGURA 6.3 Diagrama de clases del ejemplo 6.2

-Código

```
public class ContadoresAscendentes extends Thread{
   public ContadoresAscendentes(String nombre){
      this.setName(nombre);
      this.start();
   }
   public void run(){
      try{
         for(int i = 1;i<=10;i++){
            System.out.println(this.getName()+": "+i);
```

```
            sleep((long)(Math.random()*2000));
         }
      }
      catch(InterruptedException ie){
      }
   }
   public static void main(String args[]){
      ContadoresAscendentes ca1 = new
         ContadoresAscendentes("Hilo 1");
      ContadoresAscendentes ca2 = new
         ContadoresAscendentes("Hilo 2");
   }
}
```

-Explicación del código

El código base de la clase `ContadoresAscendentes` es:

```
public class ContadoresAscendentes extends Thread{
}
```

Para asignar nombres a los hilos que se generen se utilizará el método `setName()`. Este método se utilizará dentro de un constructor. De igual forma, dentro del constructor se indica la ejecución de los hilos.

```
public ContadoresAscendentes(String nombre){
   this.setName(nombre);
   this.start();
}
```

El método `run()`, queda prácticamente igual que en el ejemplo 6.1, pero al agregar el método `sleep()`, se debe declarar la excepción `InterruptedException` por lo que se utilizan las sentencias `try-catch`.

```
public void run(){
   try{
      for(int i = 1;i<=10;i++){
         System.out.println(this.getName()+": "+i);
         sleep((long)(Math.random()*2000));
      }
   }
   catch(InterruptedException ie){
   }
}
```

Como se puede observar, el método `sleep()`, tiene como argumento la función `random()` de la clase `Math`; como el método `sleep()` recibe el número de milisegundos que dormirá el hilo en formato `long`, se debe hacer un *cast* explícito, debido a que `random()` devuelve un número de tipo `double`. Debido a que la función `random()` devuelve un número entre 0 y 1, es necesario multiplicarlo por una constante para aumentar el tiempo en que se mandarán a dormir los hilos.

En el método `main`, se crean los objetos.

```
public static void main(String args[]){

   ContadoresAscendentes ca1 = new
      ContadoresAscendentes("Hilo 1");

   ContadoresAscendentes ca2 = new
      ContadoresAscendentes("Hilo 2");

}
```

Al momento de ejecutar esta clase, los hilos generados pueden ejecutarse de forma alternada. Esto se debe a que se encuentran tres procesos (el proceso principal y dos hilos) compitiendo por el procesador; como son atendidos por tiempos, sucede que el procesador puede no haber terminado con uno cuando se le vence el tiempo asignado y tiene que atender a otro.

EJEMPLO 6.3 CREAR UNA CLASE QUE INVOQUE A DOS CLASES: UNA QUE GENERE UN HILO CONTADOR ASCENDENTE Y OTRA QUE GENERE UN HILO CONTADOR DESCENDENTE.

NOTA: UTILICE LA CLASE DEL EJEMPLO 6.1

-Análisis y solución del problema

Se deben generar dos clases: ContadorAscendente y ContadorDescendente. Debido a que se utilizarán dos hilos contadores, en cada clase, al desplegar el conteo en pantalla agregar un indicador de la cuenta que está realizando.

Para la tercera clase se debe generar un objeto de cada una de las clases anteriores y ejecutar los hilos con la instrucción *start()*.

-Diagrama de clase

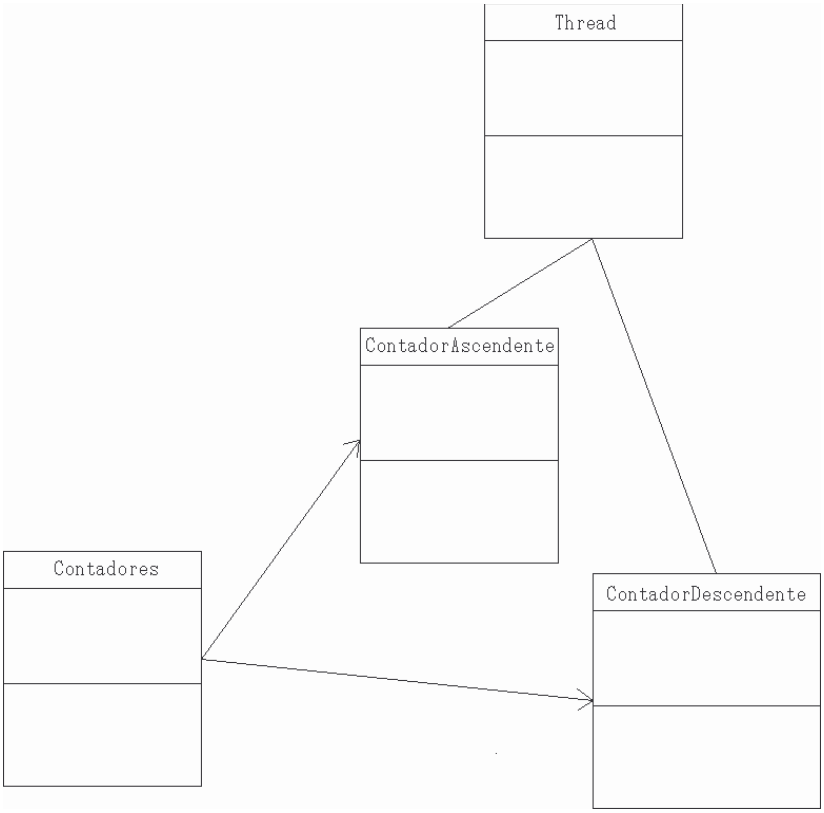


FIGURA 6.4 Diagrama de clases del ejemplo 6.3

-Código ContadorAscendente

```
public class ContadorAscendente extends Thread{
   public void run(){
      for(int i = 1;i<=10;i++){
         System.out.println("Asc: "+i);
      }
   }
}
```

-Explicación del código

Utilizando el código del ejemplo 6.1; debido a que sólo se agregará un indicador para distinguir que se trata del hilo ascendente, en el método *run()*, en la instrucción para imprimir en pantalla se agrega el identificador.
Como esta clase será utilizada por otra clase no es necesario el uso del método *main()*.

```
System.out.println("Asc: "+i);
```

-Código Contador Descendente

```
public class ContadorDescendente extends Thread{
   public void run(){
      for(int i = 10;i>=1;i--){
         System.out.println("Des: "+i);
      }
   }
```

-Explicación del código

De la misma manera que en la clase ContadorAscendente de este ejemplo, ahora se modifica el ciclo *for* para realizar el conteo descendente

```
for(int i = 10;i>=1;i--){
}
```

Pero en este caso se utilizará el indicador *Des*

```
System.out.println("Des: "+i);
```

-Código Contadores

```
public class Contadores{
   public static void main(String args[]){
      ContadorAscendente ca = new ContadorAscendente();
      ContadorDescendente cd = new ContadorDescendente();
      ca.start();
      cd.start();
   }
}
```

-Explicación del código

El código base para esta clase es:

```
public class Contadores{
}
```

Como esta clase únicamente utilizará objetos de otras clases sólo contendrá el método *main()*. Por lo que el código queda como:

```
public class Contadores{
   public static void main(String args[]){
   }
}
```

Ahora sólo falta crear los objetos de las clases anteriores y ejecutarlos con el método *start()*.

```
ContadorAscendente ca = new ContadorAscendente();
ContadorDescendente cd = new ContadorDescendente();
ca.start();
cd.start();
```

Como se puede observar, cuando se tiene una aplicación y se generan hilos para realizar distintas actividades, se puede obtener un programa que divida las operaciones matemáticas de algún cálculo en específico, de esta forma, si se cuenta con un sistema de procesamiento más robusto, se puede ejecutar más rápidamente la aplicación.

Existe toda una rama de estudio de algoritmos y programación de multitareas, donde se incluye procesamiento distribuido y paralelo para elaboración de sistemas de cómputo de alto desempeño. Es decir, obtener sistemas que sean más eficientes en uso de recursos de procesador, memoria y periféricos. Si es de su interés, el lector puede introducirse a esta rama ya con las bases expuestas en este tema.

6.2 Flujos de Datos

Otro aspecto importante, que se debe abordar para tener mayores herramientas de programación, es el ingresar información mientras se esté ejecutando un programa, o utilizar archivos para lectura o escritura.

Esto se logra utilizando un *flujo*, o en inglés *stream*. Un flujo sirve como intermediario entre la aplicación y el origen o el destino de los datos.

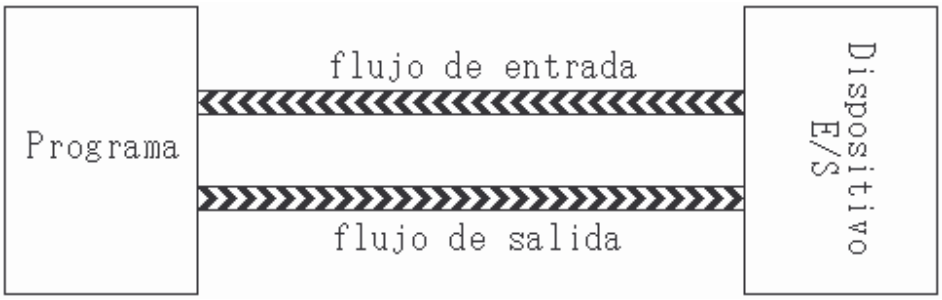


FIGURA 6.5 Esquema de un flujo de datos

Las cuatro clases base para los flujos para entrada o salida son: *InputStream*, *OutputStream*, *Reader* y *Writer*.

Flujo de salida

La clase *OutputStream* provee a sus subclases todas las características necesarias para crear un flujo de salida, es decir un flujo hacia el destino de los datos. El método más importante de esta clase es *write()*.

De igual forma, la clase *Writer* se utiliza para escribir información, y sus métodos son muy parecidos a los de la clase *OutputStream*.

Flujo de entrada

La clase *InputStream* provee a sus subclases todas las características necesarias para crear un flujo de entrada, es decir un flujo desde el origen de los datos que se requieren. El método más importante de esta clase es *read()*.

De igual forma, la clase *Reader* se utiliza para leer información, y sus métodos son muy parecidos a los de la clase *InputStream*.

Flujo estándar E/S

Un caso en particular de flujos de entrada y/o salida es el uso de la clase *System*, la cual contiene objetos tanto de la clase *InputStream*, como de la clase *PrintStream* que es subclase de *OutputStream*. Los objetos de dichas clases, dentro de la clase *System*, son:

- out – Objeto creado a partir de la clase PrintStream, el cual indica que el flujo de salida se debe tener hacia el dispositivo de salida definido por el entorno o por el usuario. Generalmente el dispositivo de salida es el monitor.
- in – Objeto creado a partir de la clase InputStream, el cual indica que el flujo de entrada se debe tener desde el dispositivo de entrada definido por el entorno o por el usuario. Generalmente el dispositivo de entrada es el teclado.
- err – Objeto creado a partir de la clase PrintStream, el cual indica que el flujo de salida, para los mensajes de error, se debe tener hacia el dispositivo de salida definido por el entorno o por el usuario. Generalmente el dispositivo de salida es el monitor.

Excepción IOException

Algo importante que hay que considerar cuando se utilizan estas clases, es que la mayoría de los métodos de *InputStream*, *OutputStream*, *Reader* y *Writer*; lanzan una excepción de entrada/salida. Por ello, es necesario utilizar la sentencia *try-catch* para declarar la excepción *IOException*.

EJEMPLO 6.4 CREAR UNA CLASE DONDE UTILIZANDO EL OBJETO *IN* DE LA CLASE *SYSTEM*, LEA DESDE TECLADO UNA LÍNEA, E IMPRIMA EN PANTALLA EL TEXTO LEÍDO.

-Análisis y solución del problema

Se identifica una clase, que se nombrará *LeerConSystem*, la cual utilizará el método *read()* del objeto *in* de la clase *System* para hacer la lectura. Dicho método leerá carácter por carácter, y obtendrá su equivalente numérico, es decir, devolverá un número entero por cada carácter leído. Esta clase contará con un atributo de tipo *char* para almacenar los caracteres leídos y un método que realice la lectura.

-Diagrama de clase

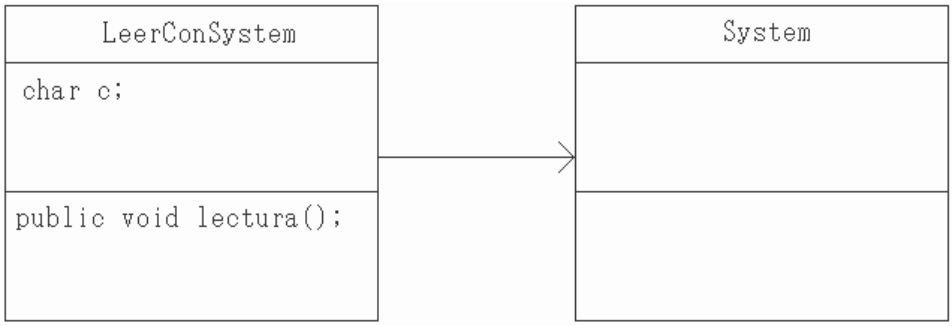


FIGURA 6.6 Diagrama de clase del ejemplo 6.4

-Código

```
import java.io.*;

public class LeerConSystem{
   char c;
   public void lectura(){
      try{
         while((this.c=(char)System.in.read())!='\n'){
            System.out.println(this.c);
         }
      }
      catch(IOException ioe){
      }
   }
   public static void main(String args[]){
      LeerConSystem lcs = new LeerConSystem();
      lcs.lectura();
   }
}
```

-Explicación del código

Cuando se requiere utilizar métodos de otras clases que se encuentran en algún paquete es necesario utilizar la palabra *import*, esta palabra sería el equivalente a un *#include*<> del lenguaje C. Debido a que se utilizará el método *read()* de la clase *InputStream*, este método lanza una excepción *IOException*, por lo que se debe importar el paquete *java.io.**.

```
import java.io.*;
```

Con esta sentencia, el esquema base de la clase es:

```
import java.io.*;

public class LeerConSystem{
   char c;
   public void lectura(){
   }
}
```

En el método `lectura()`, se debe utilizar la sentencia *try-catch* para "atrapar" la excepción IOException

```
public void lectura(){
   try{
   }
   catch(IOException ioe){
   }
}
```

Como se hará la lectura de una línea completa y el método *read()* lee el valor ASCII de cada carácter, se utilizará un ciclo *while* con la condición que mientras no encuentre un salto de línea estará haciendo la lectura de la línea. Y como el valor devuelto por el método es un entero, se debe hacer un *cast* explícito para que se convierta en carácter y se pueda almacenar en el atributo *c*.

```
public void lectura(){
   try{
      while((this.c=(char)System.in.read())!='\n'){
         System.out.println(this.c);
      }
   }
   catch(IOException ioe){
   }
}
```

Sólo queda agregar el método `main` para ejecutar la clase.

```
public static void main(String args[]){
```

```
   LeerConSystem lcs = new LeerConSystem();
   lcs.lectura();
}
```

Cuando se requiere hacer una lectura más dinámica, es decir leer líneas completas sin necesidad de ir carácter por carácter, se utiliza el método readLine() de la clase BufferedReader, la cual es una subclase de Reader.

EJEMPLO 6.5 IMPLEMENTAR UNA CLASE QUE REALICE LA LECTURA DESDE EL TECLADO UTILIZANDO LA CLASE *BUFFEREDREADER*. UTILIZAR EL MÉTODO *READLINE()*.

-Análisis y solución del problema

Se requiere una clase que tendrá como atributo una variable de tipo *String*, y un método de lectura.

-Diagrama de clase

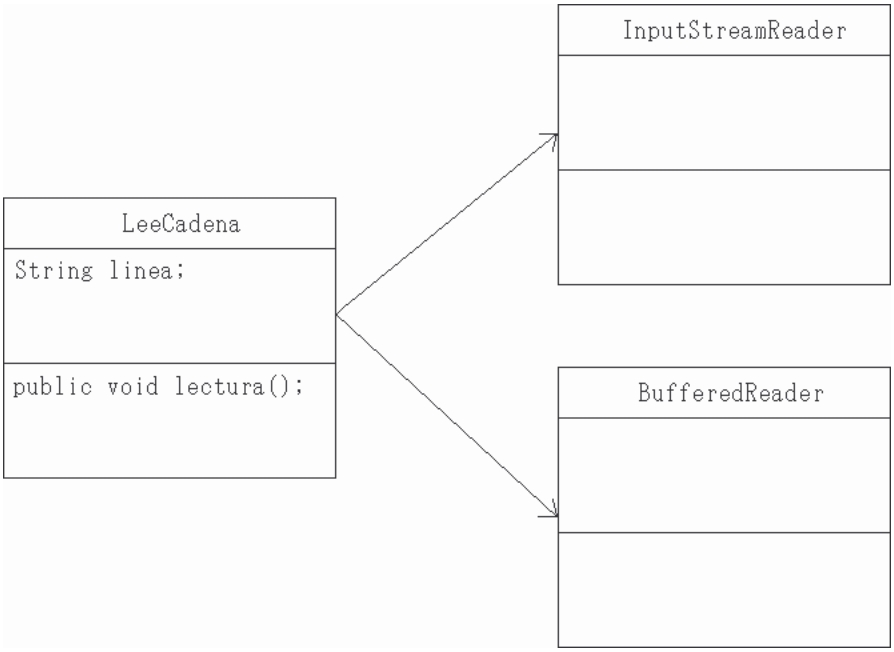


FIGURA 6.7 Diagrama de clase del ejemplo 6.5

-Código

```
import java.io.*;

public class LeeCadena{
   String linea;
```

```
   public void lectura(){
      try{
         InputStreamReader isr = new
            InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(isr);
         this.linea=br.readLine();
      }
      catch(IOException ioe){
      }
   }
   public static void main(String args[]){
      LeeCadena lc = new LeeCadena();
      lc.lectura();
      System.out.println("Linea leida: "+lc.linea);
   }
}
```

-Explicación del código

Como se utilizarán las clases de entrada y salida se debe importar el paquete *java.io*

```
import java.io.*;
```

Por lo que el código base del programa queda de la siguiente forma:

```
import java.io.*;
public class LeeCadena{
   String linea;
   public void lectura(){
   }
}
```

Dentro del método de `lecturaLinea()` como el método de `readLine()` lanza la excepción `IOException`, se debe declarar usando el `try-catch`

```
public void lectura(){
   try{
   }
   catch(IOException ioe){
   }
}
```

Para lograr hacer la lectura de la línea completa, desde el teclado, se debe utilizar un flujo de entrada; como en este caso es de lectura, se debe utilizar un objeto que genere el flujo de lectura de entrada, esto se logra utilizando la clase `InputStreamReader`.

```
public void lectura(){
```

```
   try{
      InputStreamReader isr = new
         InputStreamReader(System.in);
   }
   catch(IOException ioe){
   }
}
```

Una vez creado el objeto de `InputStreamReader` con el parámetro indicando desde donde es el flujo, en este caso `System.in`, ahora se crea un buffer para realizar dicha lectura utilizando un objeto de la clase `BufferedReader`.

```
public void lectura(){
   try{
      InputStreamReader isr = new
         InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
   }
   catch(IOException ioe){
   }
}
```

Ahora el programa está listo para hacer una lectura desde el flujo de entrada, por lo que en el atributo `linea`, se almacena la lectura hecha por el método `readLine()`, del objeto `br`.

```
public void lectura(){
   try{
      InputStreamReader isr = new
         InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      this.linea=br.readLine();
      System.out.println(linea);
   }
   catch(IOException ioe){
   }
}
```

Ahora sólo resta agregar un método `main` para ejecutar el programa.

```
public static void main(String args[]){
   LeeCadena lc = new LeeCadena();
   lc.lectura();
}
```

EJEMPLO 6.6 MODIFICAR LA CLASE BISECCION DEL EJEMPLO 4.5 PARA QUE LEA LOS VALORES DE *A*, *B* Y *TOLERANCIA*. LA ECUACIÓN A RESOLVER POR EL MÉTODO DE BISECCIÓN ES

$$f(x) = x^4 + 5.8x^3 - 22.4x^2 - 31.2x + 57.6$$

-Análisis y solución del problema

Con base en el código de la clase `Biseccion` del ejemplo 4.5, se suprimen las asignaciones de valor a los atributos. Para realizar la lectura de los valores, se debe crear un objeto de `LeeCadena`, y agregar las líneas necesarias para la asignación de dichos valores en los atributos.

-Diagrama de clases

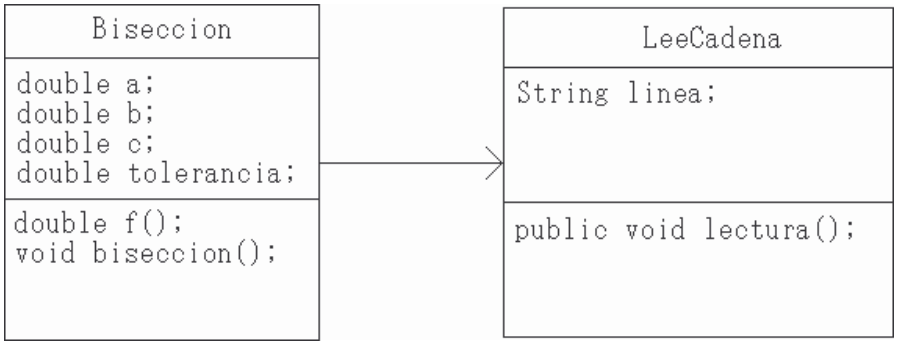


FIGURA 6.8 Diagrama de clases del ejemplo 6.6

-Código

```
class Biseccion{
   double a,b,c,tolerancia;
   int i;

   double f(double x){
      double res;
      res=Math.pow(x,4)+5.8*Math.pow(x,3)-22.4*Math.pow(x,2)-
          31.2*x+57.6;
      return res;
   }

   void biseccion(){
      do{
         this.c=(this.a+this.b)/2;
         if((this.f(this.a)*this.f(this.c))<0)
            this.b=this.c;
         else
            this.a=this.c;
```

```
         this.i++;
      }while(Math.abs(this.b-this.a)>this.tolerancia);
   }

   public static void main(String args[]){
      Biseccion bs = new Biseccion();
      LeeCadena lc = new LeeCadena();

      System.out.println("Dame el valor de a:");
      lc.lectura();
      bs.a=Double.parseDouble(lc.linea);

      System.out.println("Dame el valor de b:");
      lc.lectura();
      bs.b=Double.parseDouble(lc.linea);

      System.out.println("Dame el valor de la tolerancia:");
      lc.lectura();
      bs.tolerancia=Double.parseDouble(lc.linea);

      bs.i=0;
      bs.biseccion();
      System.out.println("El valor de la raíz es: "+bs.c+
                         " en la iteracion: "+bs.i);
   }
}
```

-Explicación del código

Dentro del método `main` se debe crear un objeto de la clase `LeeCadena`.

```
LeeCadena lc = new LeeCadena();
```

Debido a que el método `lectura` de la clase `LeeCadena` no devuelve un valor, sino que lo almacena en un atributo `String`, primero se debe hacer la lectura del dato, para después convertir dicho dato en un valor numérico usando `Double.parseDouble()`.

```
System.out.println("Dame el valor de a:");
lc.lectura();
bs.a=Double.parseDouble(lc.linea);
```

Manejo de archivos en Java

Visto el concepto de lo que es un flujo, y cómo trabaja, ahora se utilizarán los flujos para hacer la lectura o escritura de un archivo.

- Lectura de un archivo

Así como se utiliza un apuntador a archivo, es decir, un apuntador de tipo FILE en lenguaje C, en Java existe una clase que define un objeto de tipo File, el cual funciona de la misma forma que un apuntador, sólo que esto sirve únicamente para determinar el origen o destino del flujo de datos. Después de crear el objeto File, se debe crear el flujo de datos. De esta forma se puede leer o escribir.

EJEMPLO 6.7 IMPLEMENTAR UNA CLASE, EN LA CUAL SE GENERE UN FLUJO DE ENTRADA, Y SE HAGA LA LECTURA DE UN ARCHIVO, DESPLEGANDO EN PANTALLA EL CONTENIDO DE DICHO ARCHIVO. SE DEBE RECIBIR DENTRO DEL PROGRAMA EL NOMBRE DEL ARCHIVO A LEER.

-Análisis y solución del problema

Utilizando la clase de LeeCadena, se debe leer el nombre del archivo, y utilizando la cadena almacenada en el atributo del objeto se pasará como argumento del método de la clase LeeArchivo.

-Diagrama de clases

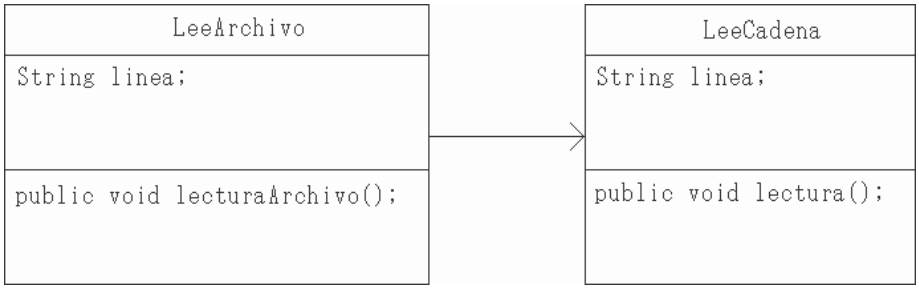


FIGURA 6.9 Diagrama de clases del ejemplo 6.7

-Código

```
import java.io.*;

public class LeeArchivo{
   String linea;
   public void lecturaArchivo(String archivo){
      try{
         File f = new File(archivo);
         FileInputStream fis = new FileInputStream(f);
         InputStreamReader isr = new InputStreamReader(fis);
         BufferedReader br = new BufferedReader(isr);
```

```
         while((this.linea=br.readLine())!=null){
            System.out.println(this.linea);
         }
         br.close();
         isr.close();
         fis.close();
      }
      catch(FileNotFoundException fnfe){
         System.out.println("No fue posible encontrar el"+
                            " archivo: "+archivo);
      }
      catch(IOException ioe){
         System.out.println(ioe.getMessage());
      }
   }

   public static void main(String args[]){
      LeeArchivo la = new LeeArchivo();
      LeeCadena lc = new LeeCadena();
      System.out.println("Dame el nombre del archivo"+
                         " a leer: ");
      lc.lectura();
      la.lecturaArchivo(lc.linea);
   }
}
```

-Explicación del código

Debido a que se utilizan clases creadas para entrada y salida de datos, se debe importar el paquete `java.io`.

```
import java.io.*;
```

Con base en el diagrama de clases, se obtiene el código base de la clase

```
public class LeeArchivo{
   String linea;
   public void lecturaArchivo(){
   }
}
```

El método deberá recibir como parámetro el nombre del archivo a leer

```
public void lecturaArchivo(String archivo){
}
```

Al utilizar las clases del paquete `java.io`, éstas pueden arrojar una excepción por lo que se deberá utilizar la sentencia `try-catch`.

```
public void lecturaArchivo(String archivo){
   try{
   }
}
```

A continuación se crea un objeto `File`, para indicar el origen de los datos.

```
File f = new File(archivo);
```

Ahora se genera un flujo de entrada desde archivo, creando un objeto de la clase `FileInputStream`

```
FileInputStream fis = new FileInputStream(f);
```

Ahora se crea un flujo de entrada, similar al creado en el método `lectura()` del ejemplo 6.4, sólo que ahora el argumento de `InputStreamReader` es el flujo desde archivo.

```
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);
```

Con el flujo creado, el método va quedando de la siguiente manera:

```
public void lecturaArchivo(String archivo){
   try{
      File f = new File(archivo);
      FileInputStream fis = new FileInputStream(f);
      InputStreamReader isr = new InputStreamReader(fis);
      BufferedReader br = new BufferedReader(isr);
   }
}
```

Para el ciclo de impresión, se indica que mientras la línea leída sea diferente de `null`, imprima la información leída

```
while((this.linea=br.readLine())!=null){
   System.out.println(this.linea);
}
```

Terminado de leer el archivo, se deben cerrar los flujos

```
br.close();
isr.close();
fis.close();
```

Con esto se termina el bloque de `try`; ahora para las sentencias `catch`, se puede observar en la documentación de las clases, que la clase `FileInputStream`, al crear un objeto, puede lanzar `FileNotFoundException`, lo cual indica que el archivo no existe o no es un archivo sino un directorio.

```
catch(FileNotFoundException fnfe){
   System.out.println("No fue posible encontrar el"+
                      " archivo: "+archivo);
}
```

De igual forma las clases utilizadas del paquete `java.io`, los métodos pueden arrojar una excepción de `IOException`, por lo que también se debe declarar.

```
catch(IOException ioe){
   System.out.println(ioe.getMessage());
}
```

Con esto, el método *lecturaArchivo()*, que de la siguiente forma:

```
public void lecturaArchivo(String archivo){
   try{
      File f = new File(archivo);
      FileInputStream fis = new FileInputStream(f);
      InputStreamReader isr = new InputStreamReader(fis);
      BufferedReader br = new BufferedReader(isr);
      while((this.linea=br.readLine())!=null){
         System.out.println(this.linea);
      }
      br.close();
      isr.close();
      fis.close();
   }
   catch(FileNotFoundException fnfe){
      System.out.println("No fue posible encontrar el"+
                         " archivo: "+archivo);
   }
   catch(IOException ioe){
      System.out.println(ioe.getMessage());
   }
}
```

Sólo falta el método `main`, en el cual se hará la lectura del nombre del archivo

```
public static void main(String args[]){
   LeeArchivo la = new LeeArchivo();
   LeeCadena lc = new LeeCadena();
   System.out.println("Dame el nombre del archivo a leer: ");
   lc.lectura();
   la.lecturaArchivo(lc.linea);
}
```

- Escritura a un archivo

Como se ha descrito, cuando se lee o escribe a un dispositivo de E/S, se hace por medio de flujos, por lo que cuando se hace la lectura de un archivo se define un objeto de tipo `File`, y dicho objeto sirve para indicar el origen de los datos del flujo, tal y como se trató en el ejemplo 6.4, en el cual el origen de los datos era el teclado. Por lo que ahora para hacer una escritura se creará un objeto de tipo `File`, el cual funcionará como el destino de los datos, y ahora se creará un flujo de salida.

Un flujo de salida se crea utilizando la clase *Writer* y *OutputStream*, las cuales permiten generar un flujo de salida a un destino dado.

EJEMPLO 6.8 IMPLEMENTAR UNA CLASE, EN LA CUAL SE GENERE UN FLUJO DE SALIDA, Y SE HAGA LA ESCRITURA A UN ARCHIVO, DE INFORMACION LEÍDA DESDE EL TECLADO. SE ESCRIBIRÁ DENTRO DEL ARCHIVO MIENTRAS EXISTAN CARACTERES EN LA LÍNEA QUE SE LEA. SE DEBE RECIBIR DENTRO DEL PROGRAMA EL NOMBRE DEL ARCHIVO A ESCRIBIR.

-Análisis y solución del problema

En este caso se volverá a utilizar la clase `LeeCadena`, para leer datos desde teclado, mismos que se irán guardando en un archivo, por medio de una segunda clase, que se nombrará `EscribeArchivo`. Para esta última clase sólo se utilizará un método que abrirá el archivo para escritura, y utilizando el objeto de `LeeCadena` mandará a impresión los datos.

-Diagrama de clases

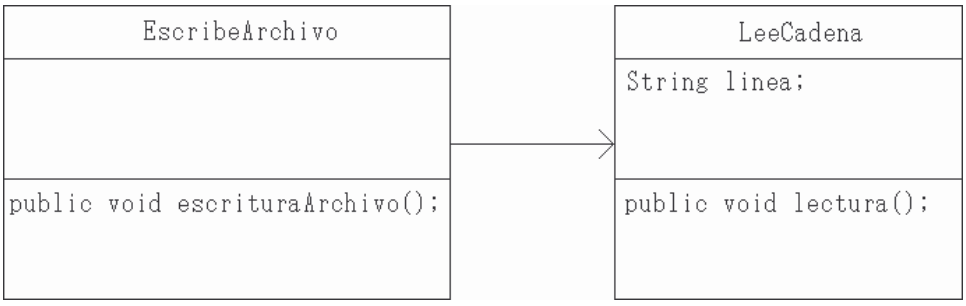


FIGURA 6.10 Diagrama de clases del ejemplo 6.8

-Código

```
import java.io.*;

public class EscribeArchivo{
   public void escrituraArchivo(String archivo){
```

```
      try{
        LeeCadena lc = new LeeCadena();

        File f = new File(archivo);
        FileOutputStream fos = new FileOutputStream(f);
        OutputStreamWriter osw = new OutputStreamWriter(fos);
        BufferedWriter bw = new BufferedWriter(osw);

        lc.lectura();
        while(lc.linea.compareTo("")!=0){
           bw.write(lc.linea+"\n");
           lc.lectura();
        }
        bw.close();
        osw.close();
        fos.close();
      }
      catch (IOException ioe){
      }
   }

   public static void main(String args[]){
      EscribeArchivo ea = new EscribeArchivo();
      LeeCadena lc = new LeeCadena();

      System.out.println("Nombre de archivo donde se"+
                         " escribira: ");
      lc.lectura();
      ea.escrituraArchivo(lc.linea);
   }
}
```

-Explicación del código

Como se está haciendo uso de clases para flujos de salida, se debe importar el paquete `java.io`.

```
import java.io.*;
```

Del diagrama de clases, se obtiene el código base del programa.

```
public class EscribeArchivo{
   public void escrituraArchivo(){
   }
}
```

El método de `escrituraArchivo()`, deberá recibir como parámetro el nombre del archivo a abrir y escribir la información dentro del archivo. Debido a que puede ocurrir un `IOException`, se debe declarar usando un `try-catch`

```
public class EscribeArchivo{
   public void escrituraArchivo(String archivo){
      try{
      }
      catch (IOException ioe){
      }
   }
}
```

Ahora se crea un objeto de la clase `LeeCadena`

```
LeeCadena lc = new LeeCadena();
```

A continuación se debe crear el objeto de tipo `File` y el flujo será de salida, por lo que de las clases utilizadas en el ejemplo 6.6, ahora se utilizan las contrarias que generan un flujo de salida.

```
File f = new File(archivo);
FileOutputStream fos = new FileOutputStream(f);
OutputStreamWriter osw = new OutputStreamWriter(fos);
BufferedWriter bw = new BufferedWriter(osw);
```

El flujo seguirá escribiendo dentro del archivo mientras haya letras escritas, en el momento en que no haya nada, se detendrá. Debido a que se está utilizando una cadena, no se puede comparar como si fuera un valor numérico o un carácter, se debe utilizar el método `compareTo()` definido en la clase `String`. Como entrando al ciclo no se ha hecho ninguna lectura, se debe hacer una primer lectura antes del ciclo.

```
lc.lectura();
while(lc.linea.compareTo("")!=0){
   bw.write(lc.linea+"\n");
   lc.lectura();
}
```

Después de haber hecho la escritura de toda la información, ahora se deben cerrar los flujos.

```
bw.close();
osw.close();
fos.close();
```

Ahora sólo falta agregar un método `main` que utilizando un objeto de `LeeCadena`, obtenga el nombre del archivo.

```
public static void main(String args[]){
   EscribeArchivo ea = new EscribeArchivo();
   LeeCadena lc = new LeeCadena();

   System.out.println("Nombre de archivo donde se"+
                      " escribira: ");
   lc.lectura();
   ea.escrituraArchivo(lc.linea);
}
```

Se sugiere al lector revisar otras clases de flujos de datos para realizar las operaciones de entrada y salida que se requieren en los diferentes desarrollos de aplicaciones.

Ejercicios Propuestos

1) Modificar la clase ContadoresAscendentes del ejemplo 6.2, de tal forma que se generen automáticamente, tantos hilos como se indiquen al ejecutar la clase.
   ```
   java ContadoresAscendentes 3
   ```
   Cada hilo deberá tener un nombre.

2) Implementar en una clase la creación de 8 hilos, los cuales deberán leer 3 líneas de texto desde teclado, al término de la lectura de las 3 líneas desplegar en pantalla la información leída, así como el nombre del hilo. ¿Qué sucede al término de la ejecución?

3) Investigar la forma de asignar prioridades a hilos. ¿Para qué sirve asignar prioridades?

4) Investigar la forma de utilizar las sentencias wait() y notify(), cuando se manipulan hilos, y cuándo se deben utilizar dichas sentencias.

5) Modificar la clase EscribeArchivo del ejemplo 6.7, de tal forma que se pueda seguir escribiendo en el mismo archivo, sin borrar la información ya contenida en él. Es decir que cuando se abra el archivo y se ubique al final del mismo, para continuar escribiendo

6) Implementar una clase donde, utilizando la clase del inciso 5 y la clase LeeArchivo, se copie la información de un archivo en otro archivo.

7) Implementar en una clase, la lectura de una matriz de cualquier dimensión, desde un archivo.

8) A la clase del inciso 7, agregar un método para que transponga la matriz leída y la escriba en otro archivo.

9) Resolver por medio de Eliminación Gaussiana una matriz cuadrada de cualquier dimensión, leída desde un archivo.

CONCLUSIONES

La información contenida en el presente documento ayuda y apoya al alumno en la asignatura correspondiente, ya que tratamos de explicar a detalle conceptos y temas que en algún momento resultan ser un tanto difíciles y complicadas de entender.

Nuestro objetivo principal al comenzar este trabajo fue el desarrollar un documento el cual apoyara de la mejor manera a la asignatura Programación avanzada y métodos numéricos, contribuyendo de esta manera en el aprendizaje de la misma; puesto que se carece de material didáctico enfocado completamente a ella decidimos realizar estas notas con la finalidad de que los alumnos que cursan esta asignatura tuvieran un material de apoyo, el cual abarcara lo mejor posible todo el contenido. Dicho objetivo lo hemos visto realizado, puesto que el contenido de este libro contiene las características antes mencionadas; hemos tratado de cubrir al máximo posible el temario de la asignatura para con ello facilitar al alumno la comprensión de la misma.

Con la información contenida en este documento estamos completamente seguros que la programación ya no será una dificultad ni motivo de preocupación, ya que uno de los aspectos más importantes que tomamos en cuenta al momento para la realización de este trabajo, fue la creciente dificultad de que se tiene generalmente al abstraer y analizar problemas, razón por la cual se complica el resolver los mismos. Por lo citado anteriormente tratamos de fomentar en el lector el realizar una serie de pasos ordenados que le permitan resolver los problemas de una manera fácil y sencilla.

En este trabajo también dejamos abierta la posibilidad para aprender otro lenguaje orientado a objetos, ya que la base es la misma y existen diversos lenguajes orientados a objetos que son muy similares.

Logramos cumplir nuestras metas de apoyo-enseñanza, ya que pretendemos que cuando el alumno lea el texto, entienda y se le facilite los temas contenidos en su temario, así como el buen desarrollo y entendimiento de cada uno de los capítulos.

CONCLUSIONES PARTICULARES

Durante el desarrollo del presente trabajo hemos aprendido a visualizar diferentes cosas que como alumnos muchas veces no nos damos cuenta que pasan a nuestro alrededor, cosas que son de vital importancia y que nosotros pasamos desapercibidos, por ejemplo, no vemos el esfuerzo que desempeñan algunas personas por plasmar en material didáctico sus conocimientos para así transmitir enseñanza, ya que ellos buscan la manera de que lo que van a mostrar al lector sea explícito, ilustrativo y claro, pues su objetivo primordial es hacer que quien lea, estudie y analice su trabajo y/o escrito pueda entender de la mejor manera el contenido de éste y lo pueda llevar entonces a la práctica. Situaciones comos éstas, que parecieran tan simples y sencillas son cuestiones a las que nos enfrentamos al redactar este documento, puesto que nos vimos en la necesidad de detallar la redacción de los textos, poniéndonos siempre del lado del lector, para que éste comprendiera lo mejor posible el contexto en el momento de consultarlos, pues aunque nosotros podíamos entender a la perfección nuestras palabras, el lector no hacía lo mismo, entonces buscamos cuál era la mejor manera de redactar nuestras ideas y pensamientos para así transmitir la información clara y entendible para quien lean nuestro textos; realizado esto, la redacción y descripción de nuestro trabajo obtuvo los resultados que se muestran en el contenido del mismo y logrando con ello la satisfacción de poder apoyar a lectores que requieran el uso de esa información.

Pero no sólo fue redacción, también recurrimos a estudiar e investigar la mejor manera de explicar conceptos y ejercicios numéricos, esto a nivel de desarrollo de ejercicios, y vimos al final que lo realizado cumplía satisfactoriamente nuestras expectativas en la realización de las mismas.

Viridiana del Carmen De Luna Bonilla

Virgilio Green Pérez

APÉNDICE A

APÉNDICE A

Compilación y ejecución de un programa en lenguaje Java

Existen diversos entornos de programación que incluyen hasta el editor de textos. En este apéndice lo que se explicará es la forma en que se compila y ejecuta un programa en lenguaje Java utilizando la línea de comandos, siendo más específico la línea de comandos de Windows.

Primeramente, para compilar y ejecutar un programa escrito en lenguaje Java, se debe guardar el programa en un archivo que deberá tener por nombre el mismo nombre de la clase pero con extensión *.java*; por ejemplo, si la clase se llama *HolaMundo*, el archivo se deberá llamar *HolaMundo.java*.

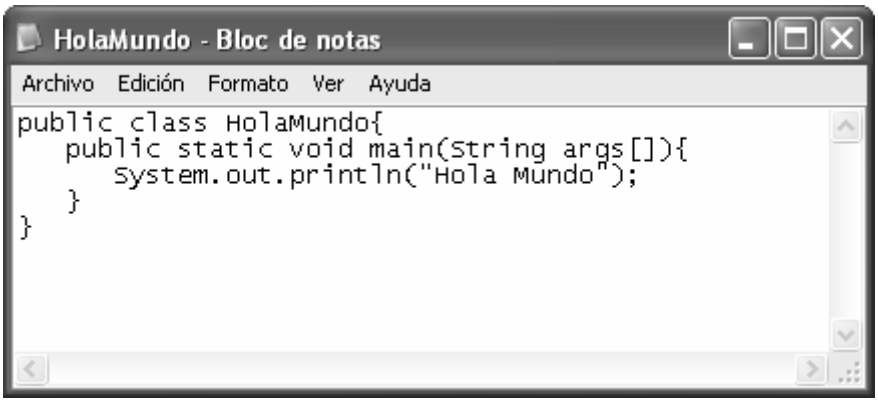


FIGURA A.1 Vista de la ventana de un bloc de notas con la clase HolaMundo.

Ahora, para compilar el programa, se debe abrir una ventana de MS-DOS, una de las formas en que se puede abrir esta ventana es desde Inicio → Ejecutar y utilizar el comando *cmd*.

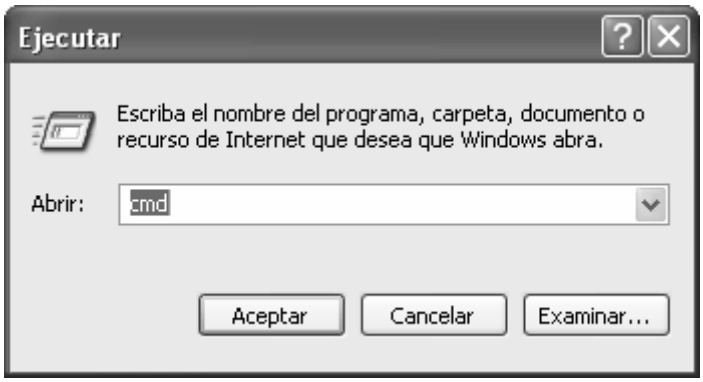


FIGURA A.2 Ventana Ejecutar.

Antes de compilar hay que verificar el valor de unas variables de entorno para la correcta compilación y ejecución del programa. En la pantalla de MS-DOS se debe introducir el comando set.

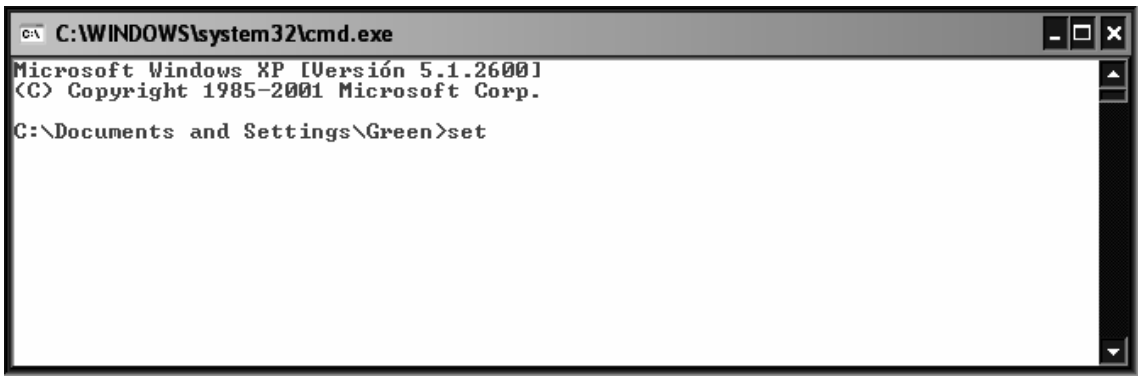


FIGURA A.3 Ventana MS-DOS.

Con este comando se visualizan todas las variables de entorno. Se debe verificar que en la variable `CLASSPATH` se tenga la ruta `C:\j2sdk1.4.1_01\jre\lib;.`; y en la variable `Path` se tenga la ruta `C:\j2sdk1.4.1_01\bin;.`.

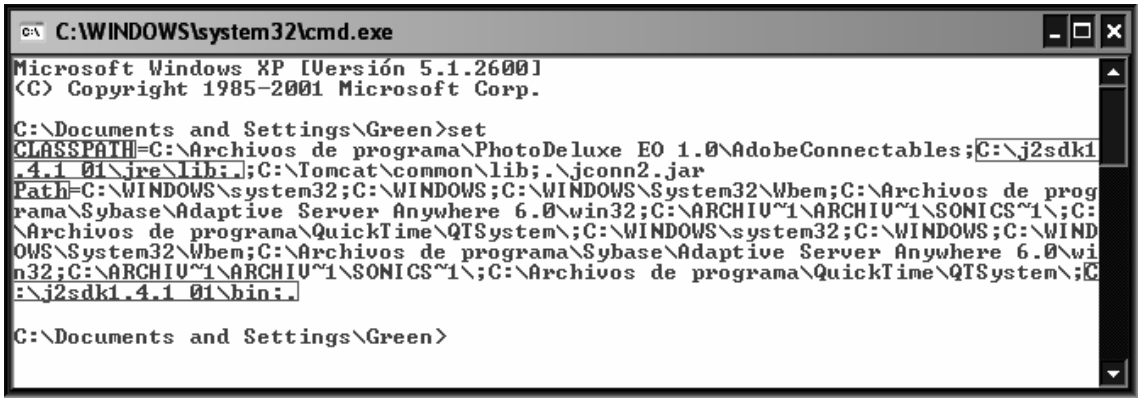


FIGURA A.4 Comando set.

NOTA: Es posible que en otra computadora no sea la ruta correcta, es decir, que no sea j2sdk1.4.1_01 sino que sea j2sdk1.5.2_02, o cualquier otra ruta; utilizando un explorador de Windows, se verifica dentro de la unidad C, el nombre de la carpeta que contiene los comandos de java. En este caso se tiene que es:

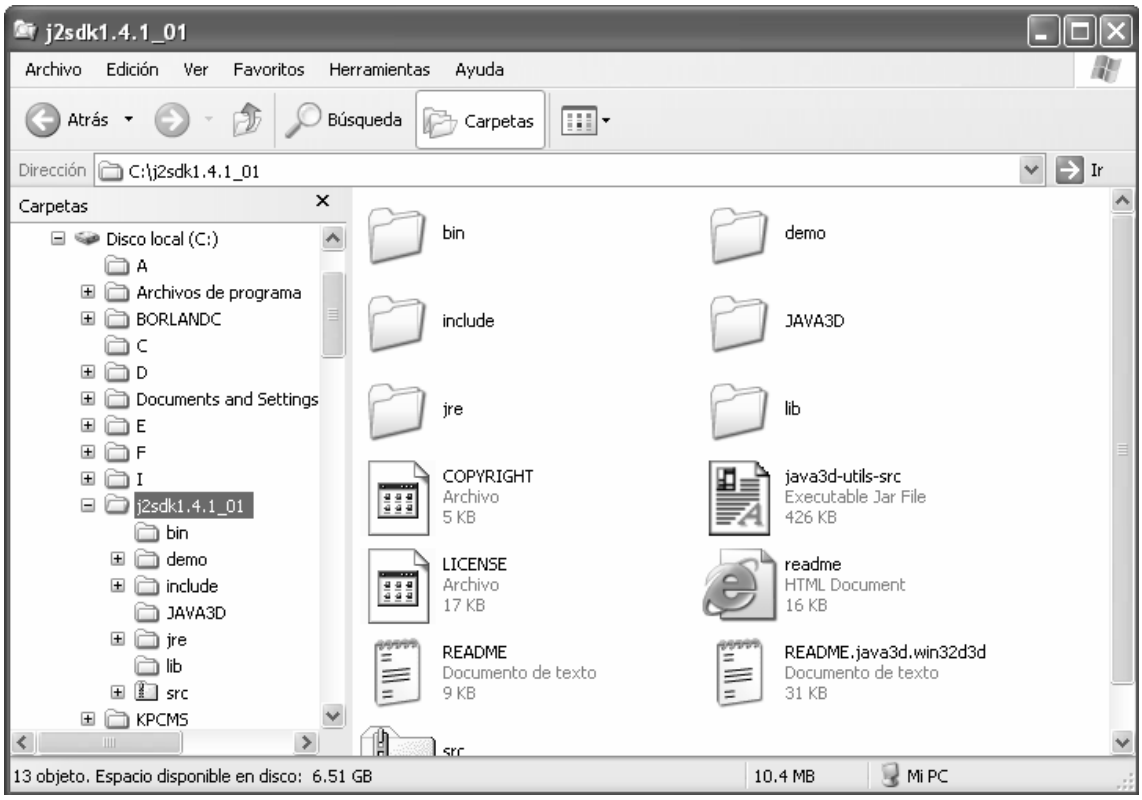


FIGURA A.5 Verificar ruta con un explorador de Windows.

Si no contiene dichos valores, en la misma línea de comandos se deberá escribir:

```
set CLASSPATH = %CLASSPATH%;C:\j2sdk1.4.1_01\jre\lib;.
set Path = %Path%;C:\j2sdk1.4.1_01\bin;.
```

Lo que se hace con %CLASSPATH% es mantener el contenido de CLASSPATH y agregar la ruta que se pone a continuación.

Una vez hecho esto, ahora se debe ubicar en el directorio donde se encuentran los archivos a compilar y ejecutar, en este caso la ruta de los archivos es D:\PAyMN

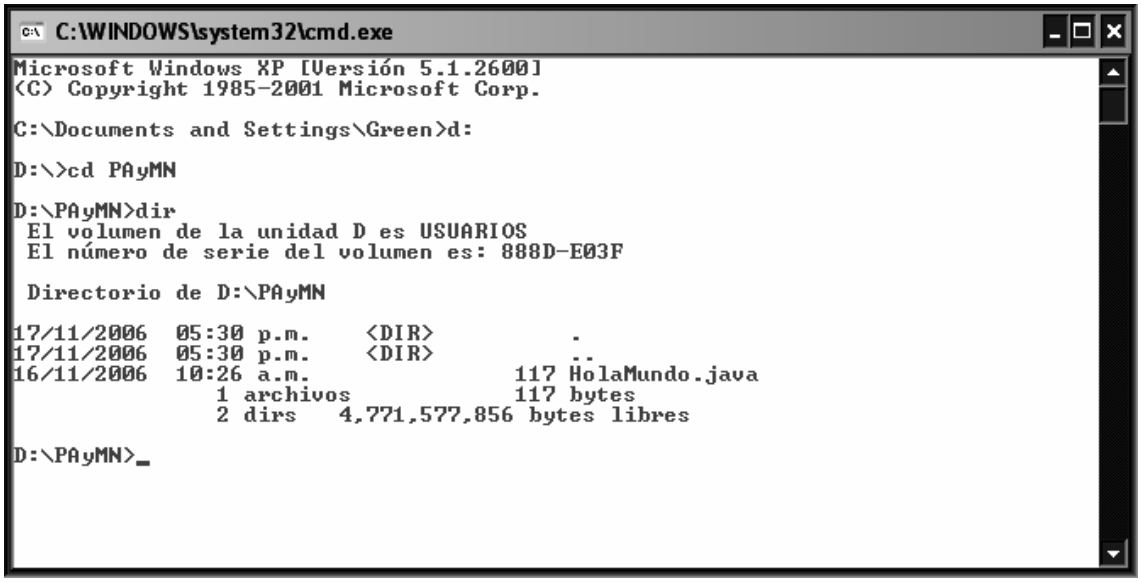


FIGURA A.6 Ubicarse en la ruta del archivo.

Para compilar el programa se debe utilizar el comando `javac`, que es el compilador del lenguaje Java, seguido por el nombre del archivo, en este caso `HolaMundo.java`. Esto genera un archivo con extensión .class

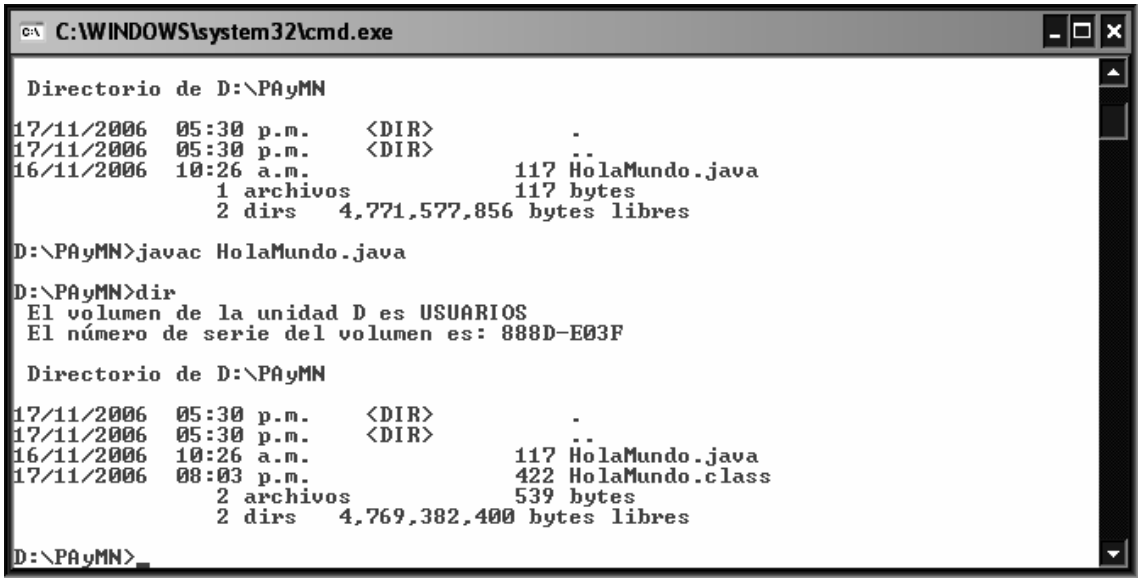


FIGURA A.7 Después de compilar el programa se genera el archivo *.class* en el mismo directorio.

Para ejecutar el archivo `.class`, sólo se debe mandar llamar a la máquina virtual de Java mediante el comando `java`, seguido del nombre de la clase, es decir `HolaMundo`. No es necesario poner el `.class`.

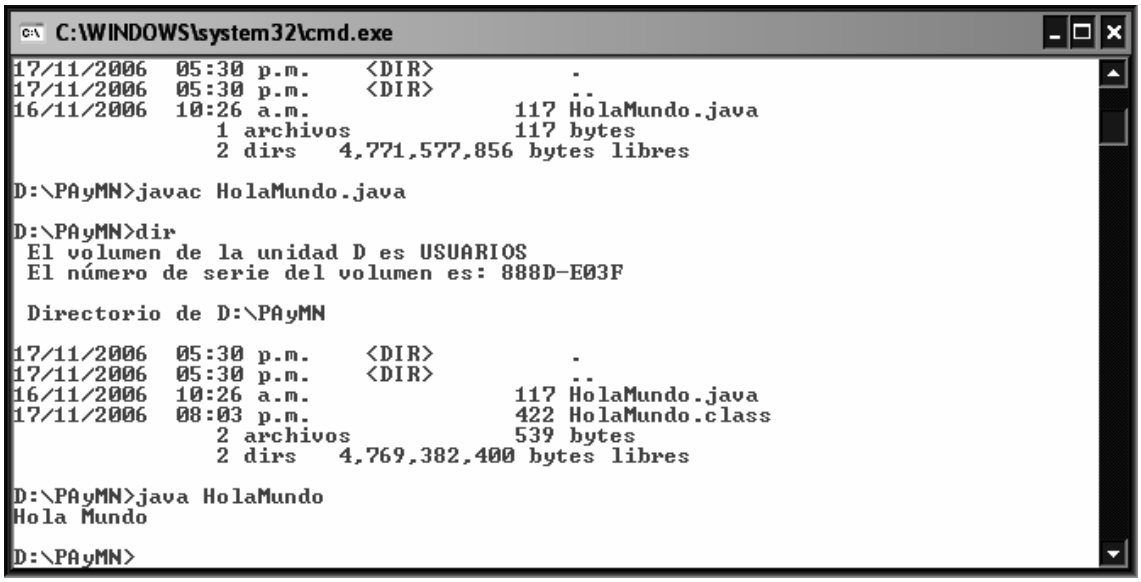


FIGURA A.8 Ejecución del programa.

Es común que un programa en Java invoque a métodos de otras clases. Si los métodos invocados no pertenecen a clases predefinas en Java, es importante considerar que el archivo *.class* perteneciente a la clase usada se encuentre en el mismo directorio del programa a ejecutar.

APÉNDICE B

ARREGLOS Y ESTRUCTURAS

Objetivos

El alumno conocerá y aplicará los conceptos de arreglo y estructura en la realización de programas que resuelvan problemas de tipo numérico.

Al final de esta práctica el alumno podrá:

1. *Manejar arreglos de varias dimensiones.*
2. *Manipular estructuras.*
3. *Realizar correctamente programas que utilicen arreglos y estructuras.*

Antecedentes

1. Manejar sentencias de control de flujo.
2. Entender el concepto de arreglo y estructura en la programación estructurada.

Introducción

Arreglo

Un arreglo es un conjunto de datos del mismo tipo, que están organizados secuencialmente en memoria principal, y que dichos datos se acceden a través del nombre del arreglo. Cada uno de los componentes del arreglo es llamado *elemento*, y cada elemento ocupa una dirección contigua en memoria.

Los arreglos pueden ser de una, dos o más dimensiones. La complejidad de su uso aumenta de acuerdo al número de dimensiones en que se haya definido el arreglo. Así un arreglo de una dimensión se puede ver como una lista lineal de datos, uno de dos dimensiones como una tabla, y uno de tres dimensiones como un conjunto de tablas.

Ejemplo un arreglo de una dimensión o unidimensional:

```
int  arr1[5];
```

En esta declaración se indica que a la variable *arr1* le asigne un espacio en memoria, de tal forma que se puedan almacenar 5 elementos de tipo *int*. Por ejemplo, si en un cierto equipo, el tipo *int* tiene un tamaño de 2 bytes, quiere decir que la dirección de cada elemento, suponiendo que empieza en FF00, sería:

arr1[0] → FF00
arr1[1] → FF02
arr1[2] → FF04
arr1[3] → FF06
arr1[4] → FF08

Con esto se puede observar que las direcciones son contiguas y dejan 2 bytes para almacenar correctamente a un *int*.

Cabe hacer notar que el arreglo emplea ***subíndices*** para hacer referencia a cada elemento de éste. Así, el primer elemento del arreglo está asociado con el subíndice *0*, el segundo con el *1* y así sucesivamente hasta el subíndice *n-1*, donde *n* es el número de elementos del arreglo.

Ejemplo de un arreglo de 2 dimensiones:

int arr2[3][2];

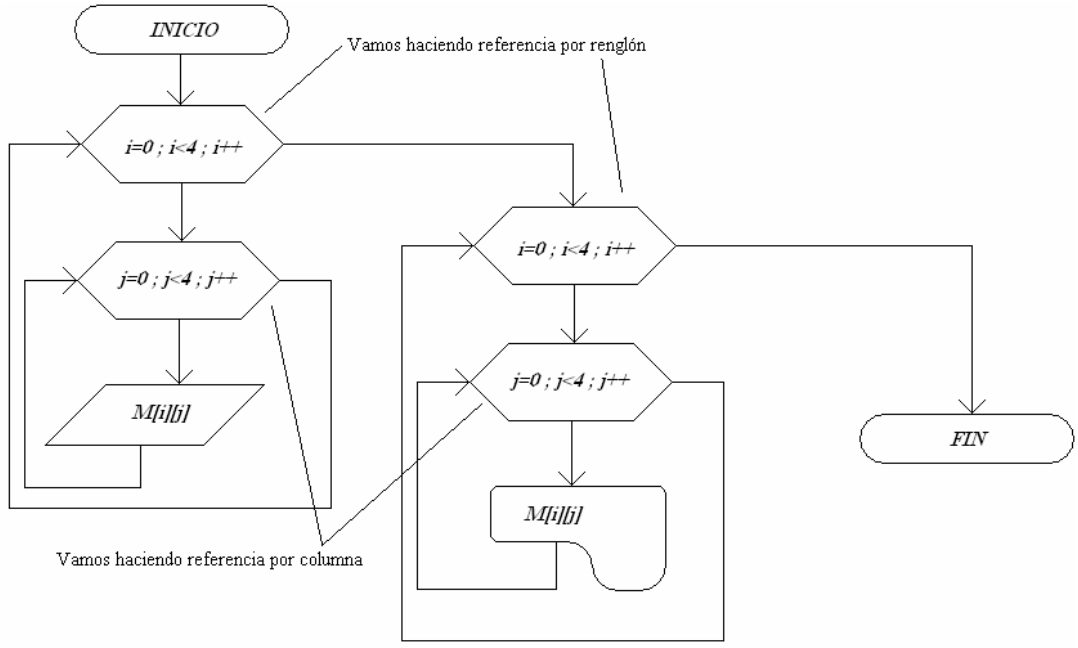
La dirección de cada elemento, suponiendo que empieza en FF0A, sería:

arr2[0][0] → FF0A
arr2[0][1] → FF0C
arr2[1][0] → FF0E
arr2[1][1] → FF10
arr2[2][0] → FF12
arr2[2][1] → FF14

Se puede observar que las localidades de memoria se asignan por renglones, es decir, se colocan los 2 elementos del renglón 0 de manera consecutiva, después los 2 elementos del renglón 1 y así sucesivamente.

Ejemplo utilizando arreglos

Ejemplo 1. Leer los elementos de una matriz cuadrada de orden 4 y desplegarlos en pantalla.



-Explicación de las variables

i: Variable de tipo entero, utilizada como variable de control de ciclos y como índice del arreglo.
j: Variable de tipo entero, utilizada como variable de control de ciclos y como índice del arreglo.
M: Arreglo bidimensional de tipo entero.

-Código

```
/* Ejemplo1: Programa que lee y despliega en pantalla una matriz
de 4 x 4 */

#include<stdio.h>

void main(){
  int M[4][4];
  int i,j;
  /* Ciclo para la lectura por renglón*/
  for(i=0;i<4;i++)
    for(j=0;j<4;j++){
      printf("Dame el elemento %d,%d: ",i,j);
      scanf("%d",&M[i][j]);
    }
  printf("\n");
  /*Ciclo para la escritura de la matriz en pantalla*/
  for(i=0;i<4;i++){
    for(j=0;j<4;j++)
      printf("%d\t",M[i][j]);
    printf("\n");/*nueva línea para imprimir el siguiente renglón*/
  }
}
```

Estructuras

Una estructura es un conjunto de datos a los que se hace referencia a través de un mismo nombre. A diferencia de un arreglo, una estructura puede tener elementos de distintos tipos, por ejemplo de tipo int, float, double, char, etc.

Declaración de una estructura

```
struct{
  tipo1 nom_var11, nom_var12,…;
  tipo2 nom_var21, nom_var22,…;
  .
  .
  tipon nom_varn1, nom_varn2,…;
}nombre_variable_estructura ;
```

Ejemplo de declaración de una estructura

```
struct {
  int i,j;
  float x,y;
}var_estructura;
```

En la sentencia anterior se define una variable de tipo *estruct* nombrada *var_estructura* compuesta de dos elementos de tipo *int* y dos elementos de tipo *float*.

Ejemplo utilizando estructuras

Ejemplo 2. Leer el nombre y edad de un alumno y almacenarlo en una estructura.

-Explicación de variables

alumno: Variable de tipo estructura con 2 variables.
- nombre: Arreglo unidimensional de tipo carácter.
- edad: Variable de tipo entero.

-Código

```
/* Ejemplo 2: Programa que lee el nombre y la edad del alumno y
los guarda en una estructura */

#include<stdio.h>

void main(){
/*Declaración de la estructura con el nombre alumno*/
 struct{
   char nombre[50];
   int edad;
 }alumno;

 printf("\n Introduce el nombre: ");
 /*Lee la cadena introducida utilizando el formato %s, y es
 almacenada en la variable nombre de la estructura alumno*/
 scanf("%s",alumno.nombre);
 printf("\n Introduce la edad: ");
 scanf("%i",&alumno.edad);
 printf("\n\nEl nombre tecleado es %s",alumno.nombre);
 printf("\nLa edad es %i",alumno.edad);
}
```

En este ejemplo se observa que para referirse a un elemento de la estructura se hace uso del operador punto (.), separando el nombre de la variable de tipo estructura y el nombre del elemento: alumno.nombre y alumno.edad.

Problemas Propuestos

1. Hacer el algoritmo y programa que almacene números en una matriz de 5 x 6. Imprimir la matriz, así como la suma de todos sus elementos.
2. Hacer el algoritmo y programa que lea una matriz de 5 x 5 y determine la posición [renglón, columna] del valor máximo en la matriz. En caso de que el valor máximo se repita, determinar todas las posiciones en las que se encuentra.
3. Hacer el algoritmo y programa que lea una matriz de 6 x 4. Calcular la suma de cada renglón y almacenarla en un vector; la suma de cada columna y almacenarla en otro vector; e imprimir ambos vectores.
4. Hacer el algoritmo y programa que lea una matriz de 3 x 5. Sumar las columnas e imprimir la columna que tuvo la máxima suma, así como ese resultado.
5. Hacer el algoritmo y programa que lea una matriz de 5 x 5 y que almacene la diagonal principal en un vector. Imprimir el vector resultante.
6. Hacer el algoritmo y programa que llene automáticamente una matriz de 20 x 20 con valores de cero, excepto en la diagonal principal, que deberán ser unos. Imprimir en pantalla.
7. Hacer el algoritmo y programa que almacene en un arreglo de 3 dimensiones, dos matrices cuadradas de 3 x 3, hacer la suma, y guardar la nueva matriz, dentro del mismo arreglo. Imprimir las tres matrices.
8. Hacer el algoritmo y programa que lea una matriz de 5 x 5, y que imprima la matriz transpuesta.
9. Hacer el algoritmo y programa que lea una matriz de 5 x 6 y que imprima cuántos de los números almacenados son ceros, cuántos son positivos y cuántos son negativos.
10. Hacer el algoritmo y programa que realice la multiplicación de dos matrices de 3 x 3 e imprima el resultado en pantalla.
11. Hacer el algoritmo y programa que pregunte por el orden de dos matrices y determine si se puede realizar la suma de la primera con la segunda, en caso de ser cierto, que obtenga los valores de dichas matrices, realice la suma e imprima el resultado en pantalla.
12. Hacer el algoritmo y programa que almacene en una estructura, un número complejo, y que obtenga su forma polar.
13. Hacer el algoritmo y programa que almacene en un arreglo de estructuras, dos números complejos, y que obtenga la multiplicación.
14. Hacer el algoritmo y programa que almacene en un arreglo de estructuras, los puntos extremos de dos rectas, y que determine la longitud de las mismas; además indicar si son perpendiculares.
15. Hacer el algoritmo y programa que almacene en un arreglo de estructuras, el nombre y calificación de varios alumnos, que obtenga el promedio de las calificaciones, y que diga el nombre del alumno con la calificación más alta. Si se repiten las calificaciones altas, deberá indicar todos los alumnos con esa calificación.

APUNTADORES

Objetivos

El alumno conocerá y aplicará el concepto de apuntador para la realización de programas de manejo eficiente de memoria.

Al final de esta práctica el alumno podrá:

4. *Acceder a localidades de memoria a través de apuntadores para el uso eficiente del almacenamiento primario.*

Antecedentes

3. Manejar sentencias de control de flujo en lenguaje C
4. Utilizar arreglos de varias dimensiones para almacenamiento de datos
5. Conocer la estructura de la memoria principal de una computadora

Introducción

Apuntador

Comúnmente se manejan variables que tienen el siguiente aspecto:

```
int i;
char c;
float f;
```

Cada una de estas variables tiene asignada una localidad de memoria, de tamaño suficiente para almacenar un valor del tipo especificado; es decir, a través del nombre de la variable y no de la dirección se almacenan datos o valores en la memoria.

Otro tipo de variables, que se utilizan en el lenguaje C, son las de tipo *apuntador* (*o puntero*). Su característica principal es que, a diferencia de las variables de uso común, éstas no almacenan datos como tal, sino que almacenan direcciones de memoria.

En la declaración de variables de tipo apuntador, se debe indicar el tipo de valor que se almacenará en la dirección contenida por dicha variable apuntador. Para especificar que se trata de un apuntador, al declararlo, se debe anteponer un asterisco al nombre de la variable. Ejemplos:

```
int *ip;        /* ip contendrá direcciones de memoria en donde se
                   almacenen valores del tipo entero */
char *cp;       /* cp contendrá direcciones de memoria en donde se
                   almacenen valores del tipo carácter */
float *fp;      /* fp contendrá direcciones de memoria en donde se
                   almacenen valores del tipo flotante */
```

Ejemplo 1

Programa A

```
#include<stdio.h>

main(){
   int a=0;
   a = 5;
   a = a * 2;
   printf("El valor de a es
%d",a);
}
```

Programa B

```
#include<stdio.h>

main(){
   int a=0,*ip=&a;
   *ip=5;
   *ip = *ip * 2;
   printf("El valor de a es
%d",*ip);
}
```

Estos dos códigos hacen exactamente lo mismo, la diferencia es que en el Programa A, sólo se emplea la variable *a*, y en el Programa B, se utiliza además un apuntador a través del cual se modifica el valor de la variable *a*.

Del Programa B, se puede observar que con la sentencia

int a=0, *ip=&a;

al tiempo que se declara el apuntador a enteros *ip,* se le da el valor inicial *&a*. El operador *&* indica "la dirección de"; así, el valor inicial de *ip* será la dirección de *a*.

En el manejo de apuntadores, además del operador *&*, existe el operador *, con el cual se accede a la información que está almacenada en la dirección contenida en la variable apuntador. Por esta razón, la siguiente línea de código

```
*ip=5;
```

asigna el valor de 5 a la localidad de memoria que apunta *ip*; equivalente a escribir *a = 5*, ya que *ip* tiene la dirección de *a*.

Ahora, si se desea mostrar la dirección en la cual está contenido el valor de la variable *a*, se puede agregar la siguiente sentencia:

printf("La dirección de a es %X",ip);

Con esto se despliega en pantalla la dirección de la variable *a* en hexadecimal (formato %X); como se puede observar no utilizamos el operador *, debido a que deseamos conocer la dirección de memoria.

Nombre de un arreglo como apuntador

Cuando se hace referencia a un elemento del arreglo, generalmente se usa el nombre de la variable y su posición dentro de éste:

```
vector[2] = 4;     /* al tercer elemento de vector se le asigna un valor de 4 */
```

Pero lo que hace C, en realidad, es utilizar el nombre del arreglo como un apuntador y al indicarle la posición del elemento, hace referencia a la dirección que obtiene de sumar la dirección de inicio del arreglo más la posición del elemento en cuestión. De la línea de código anterior, sería equivalente a:

*(vector +2) = 4;

Ejemplo 2

Programa que presenta el manejo de arreglos empleando el nombre del arreglo como apuntador.

-Explicación de variables

arr: Arreglo unidimensional de tipo entero
ip: Apuntador de tipo entero
i: Variable de tipo entero, utilizada como variable de control de ciclos y como índice del arreglo.

-Código

```
#include<stdio.h>
void main(){
   /*Se declara un arreglo de 5 elementos con valores iniciales*/
   int arr[5]={5,7,11,13,17};
   /*Se declara un apuntador a enteros*/
   int *ip;
   /*Se declara una variable entera que sirva como contador*/
   int i;
   /*Se asigna a ip, la dirección del primer elemento de arr*/
   ip=arr; /*Esta línea es equivalente a utilizar ip = &arr[0]*/
   printf("El valor de ip es %X\n",ip);
   /*Ciclo de impresión de los elementos de arr*/
   for(i=0;i<5;i++){
      printf("El valor de arr[%d] es: %d\n",i,arr[i]);
   }
   /*Ciclo de impresión de los elementos del arreglo utilizando el
     apuntador ip*/
   for(i=0;i<5;i++){
      printf("El valor de arr[%d] usando ip es: %d, dir:
              %X\n",i,*(ip+i),(ip+i));
   }
   /*Ciclo de impresión de los elementos del arreglo utilizando su
     nombre(arr) como apuntador*/
   for(i=0;i<5;i++){
      printf("El valor de arr[%d] usando solo arr es: %d, dir:
             %X\n",i,*(arr+i),(arr+i));
   }
}
```

Revisando el código, primeramente se hace la declaración de un arreglo de tipo *int* de 5 elementos, asignándole valores iniciales; así como de una variable de tipo apuntador a enteros.

```
int arr[5]={5,7,11,13,17};
int *ip;
```

Posteriormente se almacena el valor de la dirección inicial del arreglo, que corresponde al elemento cero de éste, en la variable de tipo apuntador.

```
ip=arr;
```

En el primer ciclo de impresión de los valores del arreglo, se continúa usando *arr[i]*; pero en el segundo ciclo se utiliza el apuntador *ip* para hacer referencia a los mismos elementos de *arr*, y esto se hace mediante el álgebra de apuntadores, donde sólo existen 2 operaciones: suma y resta; de esta forma un apuntador sólo puede operar con otro apuntador o con cualquier variable o constante entera.

En el último ciclo de impresión, se observa que la utilización del nombre del arreglo sigue las mismas reglas como si se estuviera utilizando apuntadores, porque se comporta de la misma forma.

Ejemplo 3

Programa que realiza la suma de dos polinomios de cualquier grado.

-Explicación de variables

polinom: Nuevo tipo de estructura, definida por el usuario.
grado: Variable de tipo entero que almacena el grado del polinomio a manipular.
coef: Variable de tipo apuntador que ayudará en el almacenamiento de los coeficientes del polinomio.

polA: Variable de tipo *polinom* que almacenará la información del polinomio A
polB: Variable de tipo *polinom* que almacenará la información del polinomio B
polRes: Variable de tipo *polinom* que almacenará la información del polinomio resultante de la suma de A y B.

i: Variable de tipo entero, utilizada como variable de control de ciclos
gradoMAX: Variable de tipo entero, utilizada para identificar el grado máximo entre A y B
polMAX: Variable de tipo entero, utilizada para indicar el polinomio de grado máximo.

-Código

```
#include<stdio.h>
#include<stdlib.h>

// Definición de un nuevo tipo de dato: polinom
typedef struct{
   int grado;
   int *coef;
}polinom;

// Función de lectura de polinomios
polinom LeePolinom(){
   polinom pol;
   int i = 0;
   pol.grado=0;
   pol.coef=NULL;

   printf("Indica el grado del polinomio: ");
   scanf("%d",&pol.grado);

   /* Asignación de memoria para el vector de coeficientes, según
      el grado del polinomio */
   pol.coef = (int *)malloc((pol.grado + 1) * sizeof(int));

   if(pol.coef != NULL){
      //Lectura de coeficientes, de mayor a menor grado
      for(i=pol.grado; i>=0;i--){
         printf("Dame el valor del coeficiente %d: ",i);
         scanf("%d",&pol.coef[i]);
      }
   }
   return pol;
}

// Función que suma A y B
polinom SumaPolinom(polinom polA, polinom polB){
   int gradoMAX, polMAX;
   int i=0;
   polinom polRes = {0,NULL};
   /* Condición que indica el grado que tendrá polRes */
   if(polA.grado>=polB.grado){
      gradoMAX = polA.grado;
      polMAX = 1;
   }
   else{
      gradoMAX = polB.grado;
      polMAX = 2;
   }
   polRes.grado = gradoMAX;
```

```
   //Asignación de memoria para polRes
   polRes.coef=(int *)malloc((gradoMAX + 1) * sizeof(int));

   //Si A fue de mayor grado
   if(polMAX == 1){
       /*Ciclo de copiado de los primeros coeficientes de A en
          Res */
      for(i=gradoMAX;i>polB.grado;i--){
         polRes.coef[i]=polA.coef[i];
      }
      //Suma de coeficientes restantes
      for(i=polB.grado;i>=0;i--){
         polRes.coef[i]=polA.coef[i]+polB.coef[i];
      }
   }

   //Si B fue de mayor grado
   else{
       /* Ciclo de copiado de los primeros coeficientes de B en
          Res */
      for(i=gradoMAX;i>polA.grado;i--){
         polRes.coef[i]=polB.coef[i];
      }
       //Suma de coeficientes restantes
      for(i=polA.grado;i>=0;i--){
         polRes.coef[i]=polB.coef[i]+polA.coef[i];
      }
   }
   return polRes;
}


// Función Principal

void main(){
   polinom polA, polB, polRes;
   int i=0;

   polA = LeePolinom();
   if(polA.coef !=NULL){
      polB = LeePolinom();
      if(polB.coef != NULL){
         polRes = SumaPolinom(polA,polB);
         if(polRes.coef != NULL){
            for(i=polRes.grado;i>=0;i--){
               printf("+(%d)x^%d ",polRes.coef[i],i);
            }
         }
      }
   }
```

```
   //Se libera la memoria asignada a los polinomios
   free(polA.coef);
   free(polB.coef);
   free(polRes.coef);
}
```

Descripción del ejemplo 3

En este programa, primeramente se define un nuevo tipo de dato, utilizando *typedef*; el nuevo tipo de dato nombrado *polinom* corresponde a una estructura que contiene dos elementos: *grado* y *coef*. En el primer elemento se almacenará el grado de un polinomio y en el segundo, los coeficientes de dicho polinomio, los cuales deberán ser enteros; como *coef* está definido como un apuntador a enteros, se utilizará para generar un vector de enteros (los coeficientes).

```
typedef struct{
   int grado;
   int *coef;
}polinom; /* nombre del nuevo tipo de datos */
```

Como el programa requiere leer dos polinomios de cualquier grado, se define una función que haga la lectura de un polinomio, la cual se invocará dos veces.

```
// Función de lectura de polinomios
polinom LeePolinom(){
   polinom pol;
   int i = 0;
   pol.grado=0;
   pol.coef=NULL;

   printf("Indica el grado del polinomio: ");
   scanf("%d",&pol.grado);

   /* Asignación de memoria para el vector de coeficientes, según
      el grado del polinomio */
   pol.coef = (int *)malloc((pol.grado + 1) * sizeof(int));

   if(pol.coef != NULL){
      //Lectura de coeficientes, de mayor a menor grado
      for(i=pol.grado; i>=0;i--){
         printf("Dame el valor del coeficiente %d: ",i);
         scanf("%d",&pol.coef[i]);
      }
   }
   return pol;
}
```

Como puede observarse, esta función define una variable *pol* de tipo *polinom* y le asigna valores iniciales a sus elementos: a *pol.grado* el valor de cero y a *pol.coef* la dirección nula (NULL) por ser del tipo apuntador. Después de conocer el grado del polinomio asignamos la memoria que requiere el apuntador *coef* para almacenar a todos los coeficientes, esto se hace empleando la función *malloc()*:

```
pol.coef = (int *)malloc((pol.grado + 1) * sizeof(int));
```

El apuntador *coef* ahora se puede utilizar como un arreglo, por ejemplo en la lectura de los coeficientes.

```
scanf("%d",&pol.coef[i]);
```

Además de la lectura de polinomios, se necesita otra función con la cual se hará la suma de ambos polinomios.

```
// Función que suma A y B
polinom SumaPolinom(polinom polA, polinom polB){
   int gradoMAX, polMAX;
   int i=0;
   polinom polRes = {0,NULL};
   /* Condición que indica el grado que tendrá polRes */
   if(polA.grado>=polB.grado){
      gradoMAX = polA.grado;
      polMAX = 1;
   }
   else{
      gradoMAX = polB.grado;
      polMAX = 2;
   }
   polRes.grado = gradoMAX;
   //Asignación de memoria para polRes
   polRes.coef=(int *)malloc((gradoMAX + 1) * sizeof(int));

   //Si A fue de mayor grado
   if(polMAX == 1){
       /*Ciclo de copiado de los primeros coeficientes de A en
          Res */
      for(i=gradoMAX;i>polB.grado;i--){
         polRes.coef[i]=polA.coef[i];
      }
      //Suma de coeficientes restantes
      for(i=polB.grado;i>=0;i--){
         polRes.coef[i]=polA.coef[i]+polB.coef[i];
      }
   }

   //Si B fue de mayor grado
   else{
       /* Ciclo de copiado de los primeros coeficientes de B en
          Res */
      for(i=gradoMAX;i>polA.grado;i--){
```

```
            polRes.coef[i]=polB.coef[i];
         }
          //Suma de coeficientes restantes
         for(i=polA.grado;i>=0;i--){
            polRes.coef[i]=polB.coef[i]+polA.coef[i];
         }
   }
   return polRes;
}
```

Esta función recibe como parámetros los dos polinomios a sumar: *polA* y *polB*; después define el polinomio resultante (el que contendrá el resultado de la suma), asignándole valores iniciales.

```
polinom polRes = {0,NULL};
```

Se asigna el grado al polinomio resultante *polRes* determinado por el grado mayor *gradoMax* de entre los polinomios *polA* y *polB.* Una vez conocido el grado de *polRes*, se asigna la memoria que requiere su elemento apuntador *coef* para almacenar a todos los coeficientes, esto se hace empleando la función *malloc()*:

```
polRes.coef=(int *)malloc((gradoMAX + 1) * sizeof(int));
```

Ahora, para hacer la suma se vacían los coeficientes de mayor grado, no incluidos en el polinomio de menor grado para después realizar la suma de coeficientes uno a uno:

```
//Suma de coeficientes para el caso de que A sea de mayor grado
      for(i=polB.grado;i>=0;i--){
         polRes.coef[i]=polA.coef[i]+polB.coef[i];
```

La función ***SumaPolinom,*** como es del tipo ***polinom,*** regresa el polinomio resultante:

```
return polRes;
```

Y por último, la función principal *main*, define a los polinomios, invoca dos veces a la función *LeePolinom* para leer cada uno de los polinomios a sumar, posteriormente invoca a la función *SumaPolinom* para que realice la suma de los polinomios y finalmente imprime el polinomio resultante.

Cabe hacer notar que en la función principal también se libera la memoria asignada a los polinomios, esto es conveniente hacer cuando se usa la función *malloc()*, para evitar que el sistema genere errores de manejo de memoria.

EJERCICIOS PROPUESTOS

1. Dadas las siguientes declaraciones:

```
int dato;
int *apuntador;
apuntador = &dato;
*apuntador = 5;
```

a) ¿Qué valor tiene apuntador?
b) ¿Cuál es el valor de dato?
c) ¿Cuál es el valor de *apuntador?

2. Del siguiente programa:

```
#include<stdio.h>
main(){
   char *apuntador;
   char variable;

   variable = 1;
   apuntador = &variable;

   printf("Valor almacenado en variable = %d\n",variable);
   printf("Valor almacenado en apuntador = %d\n",apuntador);
   *apuntador = 2;
   printf("Nuevo valor almacenado en variable = %d\n",variable);
   printf("Valor almacenado en apuntador = %d\n",apuntador);
}
```

a) ¿Cuál es la salida que produce?
b) Hacer el análisis del programa, es decir explicarlo de manera puntual.

3. Se tiene un programa con las siguientes declaraciones:
int num;
int *p;

Suponiendo que la dirección de *num* es 7753 y la de *p* 8364, determinar los siguientes valores:
a) `num` b) `p` c) `&num` d) `&p` e) `*p`

Después de ejecutar, de forma acumulativa, las siguientes sentencias:

```
A. num = 5; p=12
B. num = p
C. num = &p
D. p = &p
E. p = &num
F. *p = 10
```

Por ejemplo:
Resolviendo A

A. num=5, p=12, &num=7735, &p=8364,*p=dato almacenado en 12

4. Utilizando el nombre de un arreglo unidimensional como apuntador, implemente una función que lea los coeficientes de un polinomio de orden 4, y que con base en ellos, obtenga el valor de dicho polinomio, para una *X*, dada por el usuario.

5. Utilizando la aritmética de apuntadores, implemente una función que transponga una matriz cuadrada utilizando únicamente un apuntador.

6. Utilizando 2 apuntadores, intercambiar los valores de 2 arreglos de una dimensión con el mismo número de elementos.

7. Elabore un programa que defina una variable de tipo *int*, y un apuntador de tipo *double*. Haga que el apuntador obtenga la dirección de la variable tipo *int*. ¿Qué pasa? Imprima en pantalla el valor de la variable usando el apuntador, y usando en el *printf()* el modificador *%g*. ¿Qué pasa?

8. Modifique el programa anterior, ahora usando una variable *double* y un apuntador *int*. Conteste las mismas preguntas.

MANEJO DE ARCHIVOS

Objetivos

El alumno conocerá y aplicará el concepto de archivo para el almacenamiento y recuperación de datos persistentes.

Al final de esta práctica el alumno podrá:

1. *Utilizar las herramientas para el almacenamiento y recuperación de datos contenidos en archivos.*

Antecedentes

6. Manejar ciclos de repetición en lenguaje C.
7. Manipular sentencias de control de flujo en lenguaje C.
8. Haber aplicado el concepto de apuntador para el manejo de datos en memoria principal

Introducción

Archivos

Un archivo o fichero, es un conjunto de datos que se encuentran almacenados en memoria secundaria y se distingue a través de un nombre. A diferencia de los arreglos y de las estructuras de datos ***struct*** de C, los archivos mantienen sus datos sin importar si el programa que accede a ellos está en ejecución o ha terminado. El nombre de un archivo consta de un nombre en sí y una extensión, donde la extensión indica el tipo de archivo del que se trata, pudiendo ser de texto plano (txt) o de imágenes (jpg, bmp, etc) o de cualquier otro tipo de datos.

Acceso a archivos

En el lenguaje C, para manipular un archivo se requiere una variable apuntador, la cual irá conteniendo la dirección de almacenamiento del dato del archivo al que se accederá. Dicha variable se declara de tipo apuntador a *FILE* (archivo). Un ejemplo de la definición de un apuntador a un archivo es:

```
FILE *apArch;
```

Una vez definido dicho apuntador, hay que "abrir" el archivo para acceder a los datos que contenga o que se grabarán en él; esto se hace con la función *fopen()*, la cual recibe como parámetros el nombre del archivo y el modo en que va a ser empleado.

El modo, indica la acción que se va a realizar sobre el archivo. A continuación se presenta una tabla con los diferentes modos de acceso al archivo que se "abrirá".

Modo	Acción
“r”	Abre un archivo para lectura
“w”	Crea un archivo para escritura; si existe, borra y escribe nueva información en el archivo.
“a”	Abre o crea un archivo y añade datos al final del mismo
“rb”	Abre un archivo en modo binario para lectura
“wb”	Crea un archivo binario para escritura; si existe, borra y escribe nueva información en el archivo.
“ab”	Abre o crea un archivo binario y añade datos al final del mismo

El valor que entrega la función *fopen()* es la dirección de la primera localidad de memoria del archivo. Así, una sentencia de apertura de un archivo generalmente tiene el siguiente aspecto:

apArch = fopen(“datos.txt”, “w”);

donde ***apArch*** ha sido declarado como un apuntador a archivo.

Ejemplos de manejo de archivos

Lectura de un archivo

Ejemplo 1

Programa que lee el contenido de un archivo llamado *texto.txt* y lo despliega en pantalla. (Nota: para la correcta ejecución de este programa, debe existir un archivo de texto simple con el nombre “texto.txt” en donde se vaya a ejecutar este programa).

-Explicación de variables

f: Apuntador de tipo *FILE*, auxiliar en la lectura de un archivo.
c: Variable de tipo char, auxiliar en el almacenamiento y despliegue de los caracteres del archivo en pantalla.

-Código

```
#include<stdio.h>

void main(){
   FILE *f;
   char c;
   f=fopen(“texto.txt”,”r”);
   c=fgetc(f);
   while(c!=EOF){
```

```
        printf("%c",c);
        c=fgetc(f);
    }
    fclose(f);
}
```

Descripción del ejemplo 1

Con la instrucción ***FILE *f*** se define la variable de tipo apuntador a archivo *f*. Mediante este apuntador se manipulará al archivo que el programa leerá.

Luego se abre el archivo con *fopen("texto.txt","r")* asignando el valor que devuelve la función al apuntador *f*, o sea, la dirección de la primera localidad de memoria del archivo.

Con la función *fgetc(f)*, se obtiene el carácter que está apuntando *f*; al término de esta operación, *f* actualiza su valor para apuntar al siguiente carácter. Con la sentencia

```
c=fgetc(f);
```

la variable tipo carácter *c* recibe el carácter apuntado por *f*. Esta sentencia se encuentra en un ciclo ***while*** para leer carácter por carácter del archivo e irlo imprimiendo.

Cabe mencionar que al final de un archivo siempre hay un carácter especial para indicar que ya no hay más datos contenidos en el archivo. Este carácter especial, en el lenguaje C, está definido con la constante simbólica EOF (End Of File); es por ello que en el ciclo ***while*** del programa tiene como condición que mientras ***c!=EOF*** siga imprimiendo y leyendo un carácter**.**

Una vez que se ha terminado de utilizar el archivo es conveniente "cerrarlo"; por esto, el programa contiene la sentencia

```
fclose(f);
```

Escritura en un archivo

Ejemplo 2

Programa que crea un archivo y escribe datos en él. Los datos que se almacenan en el archivo creado se leen del teclado y se toman como caracteres. Cuando ya no se ingresen más datos desde el teclado se deben dar dos saltos de línea juntos.

-Explicación de variables

f: Apuntador de tipo *FILE*, auxiliar en la escritura de un archivo
c: Variable de tipo *char*, auxiliar en el almacenamiento de los caracteres dentro del archivo.
salir: Variable de tipo entero, auxiliar para finalizar el programa

-Código

```
#include<stdio.h>

void main(){
     FILE *f;
     char c;
     int salir=0;
     f=fopen("escritura.txt","w");
     c=getchar();
     while(salir<1){
          if(c=='\n'){
               salir++;
          }
          else{
               salir=0;
          }
          fputc(c,f);
          c=getchar();
     }
     fclose(f);
}
```

Descripción del ejemplo 2

Al igual que con el programa anterior, se debe declarar un apuntador de tipo FILE, y abrir el archivo en modo escritura.

```
FILE *f;
f=fopen("escritura.txt","w");
```

Con la función getchar() se obtiene el carácter tecleado y se asigna a la variable *c*

```
c=getchar();
```

El ciclo ***while*** se emplea para ir leyendo carácter por carácter desde el teclado e irlo almacenándolo en el archivo utilizando la función ***fputc()***, la cual recibe como parámetros, el carácter a escribir, y el apuntador al archivo donde se va a escribir. En el programa la sentencia que realiza esta escritura es

```
fputc(c,f);
```

El ciclo while se detiene cuando el programa lea dos saltos de línea juntos.

Por último, es conveniente cerrar el archivo cuando se termina de escribir datos, para asegurar el correcto almacenamiento de ellos en el archivo. Para realizar esta operación, en el programa se encuentra la sentencia

```
fclose(f);
```

Otras funciones para el manejo de archivos.

Existen otras funciones en el lenguaje C que permiten manejar archivos, algunas de ellas se muestran en la siguiente tabla:

Función	Descripción	Ejemplo

feof()	Comprueba el indicador de final de archivo. Regresa un 1 si encuentra el EOF.	while(!feof(f)){
		}
		Verifica que no haya llegado al final del archivo.

fgets()	Obtiene una secuencia de caracteres de numero_MAX-1.	fgets(variable_donde_guardar, numero_MAX, apuntador_a_archivo);
		
		fgets(cadena,20,f);

fprintf()	Realiza la misma función que *printf()*, pero escribe en archivo en lugar de en pantalla.	fprintf(f,"El número escogido es: %d",num);
fscanf()	Realiza la misma función que *scanf()*, pero lee de un archivo.	fscanf(f,"%d %d",&var1,&var2);

Manejo de archivos de datos numéricos.

Es común, para un estudiante de ingeniería, almacenar datos numéricos en archivos. La manipulación de archivos que contengan datos numéricos no difiere mucho de uno de tipo texto.

Existen varias formas de almacenar números en un archivo. Por ejemplo, si se desea almacenar los elementos de un arreglo de 5 enteros en un archivo, el siguiente código podría ser parte de un programa que realizara esta operación.

```
FILE *apArch;
int arr[]={10,20,14,9,5};
/*Apertura del archivo para escritura */
apArch=fopen("numeros.txt","w");
for (int i=0;i<5;i++) {
     fprintf(apArch,"%d ",arr[i]);
}
fclose(apArch);
```

La sentencia

```
fprintf(apArch,"%d ",arr[i]);
```

escribe en el archivo apuntado por `apArch` el valor del *i*-ésimo elemento del arreglo `arr` y deja un espacio. Como esta sentencia está en un ciclo `for`, los cinco números escritos en el archivo estarán situados en un mismo renglón separados por un espacio.

Para leer 5 números enteros de un archivo que están separados por un espacio, y colocarlos en un arreglo, puede emplearse el siguiente código como parte de un programa:

```
FILE *apArch;
int arr[5];
/*Apertura del archivo para lectura */
apArch=fopen("numeros.txt","r");
for (int i=0;i<5;i++) {
     fscanf(apArch,"%d ",&arr[i]);
}
fclose(apArch);
```

En caso de que los números a almacenar o a leer de un archivo sean reales o flotantes, sólo se cambia el formato por el adecuado (%f o %g).

En ocasiones los datos numéricos están almacenados en el archivo en modo texto, por lo que hay que hacer la conversión de cadena a número o de número a cadena; para ello se sugiere revisar las siguientes funciones de C, que realizan las conversiones antes descritas

atoi()
atof()
itoa()
ftoa()

EJERCICIOS PROPUESTOS

1. Elaborar un programa que lea texto de un archivo y que lo escriba tal cual en otro archivo.
2. Construir un programa que reciba un carácter, lea un archivo e imprima en pantalla el número de incidencias del carácter dentro del archivo.
3. Hacer un programa que lea de un archivo e imprima el texto incluido pero en orden inverso.
4. Elaborar un programa que lea de un archivo una matriz cuadrada entera de orden 5 (ver NOTA) y que la despliegue en pantalla.
5. Hacer un programa que lea de un archivo una matriz cuadrada entera de orden 5 (ver NOTA) y que despliegue en pantalla sólo los elementos de la diagonal principal.
6. Modificar el programa del punto 4 para que guarde en otro archivo la matriz por renglones, cada renglón seguido de su suma, y después del último renglón, que agregue otro renglón con la suma de las columnas.
7. Elaborar un programa que lea, de dos archivos diferentes, dos matrices enteras de 3x3 y que guarde en otro archivo la suma de las matrices.
8. Modificar el programa anterior, para que realice la multiplicación y que despliegue en pantalla la matriz resultante.
9. Modificar el programa del inciso 8, para que obtenga el determinante de ambas matrices y que imprima en otro archivo únicamente la matriz con el valor más alto del determinante.

NOTA: La separación entre valores de un renglón, deberá ser únicamente un espacio o tabulador, y la separación entre renglones, por un solo salto de línea.

APROXIMACIÓN Y SOLUCIÓN NUMÉRICA DE ECUACIONES

Objetivos

El alumno conocerá y aplicará el concepto aproximación numérica. Conocerá y resolverá, utilizando la computadora, ecuaciones algebraicas.

Al final de esta práctica el alumno podrá:

5. *Calcular el error en una aproximación.*
6. *Resolver una ecuación algebraica a través de dos métodos numéricos.*

Antecedentes

9. Manejar ciclos de repetición en lenguaje C
10. Haber empleado sentencias de control de flujo en lenguaje C.

Introducción

Errores de aproximación

En el análisis numérico, al error que existe entre el valor real y el obtenido, se le llama *error de aproximación.*

Tipos de error

Existen varios tipos de error, los cuales se pueden presentar cuando se realizan cálculos numéricos, sin embargo en esta práctica se revisarán aquéllos que se deben considerar cuando se aplican los diferentes métodos numéricos para la resolución de problemas matemáticos.

a) Error por truncamiento.

Al realizar el cálculo de la fracción $\frac{24}{7}$, el resultado aproximado a seis decimales, (debido a que no se puede obtener el valor exacto) es 3.428571...

Si truncamos a dos decimales, es decir 3.42 solamente, su expresión como quebrado sería $\frac{171}{50}$, y esto, como se puede observar, está generando un error.

b) Error por redondeo.

Tomando el ejemplo anterior, si se redondea a dos decimales, es decir 3.43 solamente, su expresión como quebrado sería $\frac{343}{100}$, y esto genera un error, al igual que en el caso anterior.

Cálculo de error

Cuando se obtiene un valor por aproximación, independientemente del método utilizado, se produce un error. En el análisis numérico es muy conveniente efectuar el cálculo del error con la finalidad de conocer qué tan cercano o lejano está el valor obtenido del valor real:

a) Error absoluto.- Es la diferencia que existe entre el valor real (V_R) y el valor aproximado (V_A). Es decir $|V_R - V_A|$.

b) Error relativo.- Es la diferencia porcentual que existe entre el valor absoluto y el valor real. Y se calcula como $\frac{|V_R - V_A|}{V_R} \times 100 = \%Error$.

Ejemplo de cálculo de error

Calcular el error absoluto y relativo de la fracción $\frac{24}{7}$, tratada anteriormente.

a) Error absoluto

1) Por truncamiento

$$\left|\frac{24}{7} - \frac{171}{50}\right| = \frac{3}{350} = 0.00857...$$

2) Por redondeo

$$\left|\frac{24}{7} - \frac{343}{100}\right| = \frac{1}{700} = 0.00142...$$

b) Error relativo

1) Por truncamiento

$$\frac{\left|\frac{24}{7} - \frac{171}{50}\right|}{\frac{24}{7}} \times 100 = \frac{1}{4} = 0.25\%$$

2) Por redondeo

$$\frac{\left|\frac{24}{7} - \frac{343}{100}\right|}{\frac{24}{7}} \times 100 = \frac{1}{24} = 0.0417\%$$

Conclusión: en este caso es mejor realizar redondeo que truncamiento ya que el error es mucho menor. Hay que considerar además, que dependiendo del problema a resolver es importante saber si aún este error es aceptable o no.

Solución de ecuaciones algebraicas

Cuando de una ecuación de una sola variable, se desea calcular sus raíces, existen métodos que permiten encontrar su solución, aun cuando no se pueda despejar a la variable.

Algunos de estos métodos son:

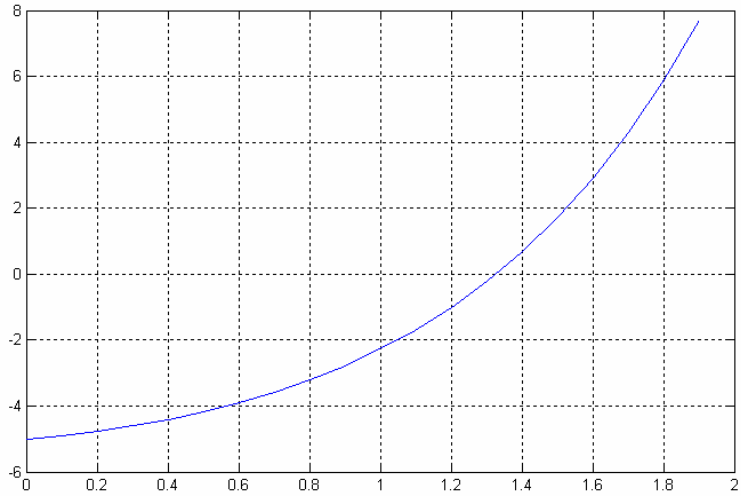
a) Método de bisección o búsqueda binaria

Este método consiste en buscar por medio de un intervalo dado, la raíz de una ecuación. Esto se hace primero dando los extremos del intervalo [a,b] donde se encuentra la raíz; así se puede observar si hay un cambio de signo en la evaluación de ambos puntos en la ecuación; si esto es cierto, se obtiene un tercer punto, que es el punto medio del intervalo, y se evalúa en la función. Ahora se verifica en qué intervalo entre [a,c] y [c,b] sigue estando el cambio de signo; donde esté el cambio de signo se considerará como el nuevo intervalo de búsqueda, este proceso se continúa hasta que la diferencia que haya entre los extremos del intervalo sea pequeña (nombrado tolerancia), siendo la solución el punto intermedio.

El algoritmo del método de bisección se presenta a continuación:

```
a = inicio_intervalo
b = fin_intervalo
REPETIR
    c = (a+b)/2
    SI f(c)*f(a)<0
        b = c
    CASO CONTRARIO
        a = c
HASTA |a – b|<tolerancia o f(c) = 0
```

Ejemplo:
De la siguiente ecuación $f(x) = xe^x - 5$ obtener el valor de la raíz.



De la gráfica, se observa que la raíz está entre 1.2 y 1.4, por lo que éste será el rango inicial.

Siguiendo el algoritmo, y tomando como tolerancia 0.002, los resultados de cada iteración se presentan en la siguiente tabla:

iteración 1)		iteración 2)	
a = 1.2	f(a) = -1.016	a = 1.3	F(a) = -0.2299
c = 1.3	f(c) = -0.2299	c = 1.35	F(c) = 0.20752
b = 1.4	f(b) = 0.67728	b = 1.4	F(b) = 0.67728
iteración 3)		iteración 4)	
a = 1.3	f(a) = -0.2299	a = 1.325	F(a) = -0.0151
c = 1.325	f(c) = -0.0151	c = 1.3375	F(c) = 0.09522
b = 1.35	f(b) = 0.2075	b = 1.35	F(b) = 0.2075
iteración 5)		iteración 6)	
a = 1.325	f(a) = -0.0151	a = 1.325	F(a) = -0.0151
c = 1.33125	f(c) = 0.03981	c = 1.328125	F(c) = 0.01229
b = 1.3375	f(b) = 0.09522	b = 1.33125	F(b) = 0.03981
iteración 7)		iteración 8)	
a = 1.325	f(a) = -0.0151	a = 1.3265625	F(a) = -0.0014
c = 1.3265625	f(c) = -0.0014	c = 1.32734375	F(c) =0.00543
b = 1.328125	f(b) = 0.01229	b = 1.328125	F(b) = 0.01229

Terminada la octava iteración se puede decir que la raíz está muy cerca de 1.32734375 que es el último valor obtenido en el centro del intervalo.

b) Método de Newton-Raphson

Este método para calcular la raíz de una ecuación algebraica es, al igual que el de bisección, es iterativo, pero la diferencia radica en que hay que darle un valor a la raíz, y las iteraciones se realizan utilizando la fórmula $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, donde

f '(x) es la derivada de la función.

Las iteraciones se suspenden cuando se llega a una cierta tolerancia o se ha encontrado la raíz.

El algoritmo del método de Newton-Raphson es:

x_0 = valor_inicial
REPETIR

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

HASTA $|x_{n+1} - x_n| <$ tolerancia o $f(x_{n+1}) = 0$

Ejemplo:

Tomando la misma ecuación del ejemplo de la aplicación del método de bisección $f(x) = xe^x - 5$, obtener la raíz utilizando ahora el método de Newton-Raphson.

Como se requiere la derivada de la función, se calcula y se obtiene: $f'(x) = xe^x + e^x$

Siguiendo el algoritmo y tomando como valor inicial 1.4 y una tolerancia de 0.002, los resultados de cada iteración se presentan en la siguiente tabla:

iteración 1)
$x_n = 1.2$
$x_{n+1} = 1.339078$

iteración 2)
$x_n = 1.339078$
$x_{n+1} = 1.326833$

iteración 3)
$x_n = 1.326833$
$x_{n+1} = 1.326725$

Comparando este método con el anterior, se observa que el método de Newton-Raphson, hace menos iteraciones para llegar al resultado, y obtiene un valor más cercano al real.

Para ambos métodos, los datos iniciales son de suma importancia ya que de esto dependerá si el proceso converge (se acerca a la solución) o no.

Ejercicios propuestos

1) Elabore un programa que reciba dos números de tipo *float*, y que calcule el error absoluto y el error relativo. Considere que el primer número sea el valor real, y el segundo el valor aproximado.

2) De la ecuación $f(x) = x^2 + 9x + 4$, elabore un programa que implemente el método de bisección, y calcule el error relativo en cada iteración. Considere el intervalo [-1,0] para la raíz en $x = -0.468871126$, o el intervalo [-10,-8.3] para la raíz en $x = -8.531128874$.

3) De la ecuación $f(x) = x\ln(x) - 5$, elabore un programa que implemente el método de bisección, y calcule el error relativo en cada iteración. Considere el intervalo [3,4], y la raíz en $x = 3.768679464$.

4) Elabore un programa que implemente el método de bisección, para la ecuación $f(x) = x^2 e^x - 2$ y calcule el error relativo en cada iteración. Los resultados los debe guardar en un archivo llamado "bisecc.txt". Considere el intervalo [0,1], y la raíz en $x = 0.901201032$

5) Considere la ecuación del inciso 2, elabore un programa que implemente el método de Newton-Raphson, y calcule el error relativo en cada iteración. Considere como valor inicial *0* para la raíz en $x = -0.468871126$, o *-10* para la raíz en $x = -8.531128874$.

6) Elabore un programa que implemente el método de Newton-Rapshon para la ecuación del inciso 3, y calcule el error relativo en cada iteración. Asigne *4* como valor inicial.

7) Considere la ecuación del inciso 4, elabore un programa que implemente el método de Newton-Raphson, y calcule el error relativo en cada iteración. Los resultados los debe guardar en un archivo llamado "newton.txt". Considere como valor inicial *1*.

8) Elabore un programa que implemente el método de bisección para resolver polinomios de orden *n*. El programa deberá recibir el grado máximo del polinomio, los coeficientes de cada uno de los términos, indicando con un *0* si ese término no existe, y el intervalo en el cual deberá aplicar el método. *NOTA: Dentro del intervalo deberá existir una raíz para que pueda converger el método.*

9) Elabore un programa que implemente el método de Newton-Raphson para resolver polinomios de orden *n*, el programa deberá recibir el grado máximo del polinomio, los coeficientes de cada uno de los términos, indicando con un *0* si el término en cuestión es nulo, y un valor inicial.

10) Utilizando el código de los incisos 8 y 9, implemente un programa para resolver numéricamente polinomios de orden *n*, dejando que el usuario escoja la opción de resolución por bisección o por Newton-Raphson.

SOLUCIÓN NUMÉRICA DE SISTEMAS DE ECUACIONES LINEALES (PARTE I)

Objetivos

El alumno conocerá y aplicará diferentes métodos de solución numérica para la resolución de sistemas de ecuaciones lineales.

Al final de esta práctica el alumno podrá:

7. *Resolver sistemas de ecuaciones lineales mediante diversas técnicas de solución numérica (Eliminación Gaussiana y Gauss-Jordan)*
8. *Implementar dichas técnicas en el lenguaje de programación C*

Antecedentes

11. Manejar ciclos de repetición en lenguaje C
12. Manejar arreglos y estructuras en lenguaje C
13. Realizar operaciones con matrices
14. Calcular la matriz inversa

Introducción

Existen varios métodos para la solución numérica de sistemas de ecuaciones lineales, entre ellos están: Eliminación Gaussiana y Gauss-Jordan. Esta práctica se enfoca a dichos métodos por lo que a continuación se da una pequeña explicación de cada uno de ellos; aunado a esto se recomienda revisar estos métodos en la bibliografía citada al final de la práctica.

Eliminación Gaussiana

Este método propone la eliminación progresiva de variables en el sistema de ecuaciones, manejando una matriz cuadrada hasta dejarla en su forma triangular, para así encontrar el valor de una variable y dejar las demás ecuaciones con 2, 3, 4, …, n variables consecutivamente, para de igual forma ir encontrando el valor de la segunda, tercera, …, enésima variable haciendo una sustitución hacia atrás.

Este proceso se realiza mediante la utilización de operaciones básicas tales como la suma, resta, multiplicación y división; es muy frecuente la multiplicación de un renglón por una constante, y la suma de éste con otro renglón, ya que de esta manera se puede llegar hasta obtener una ecuación en donde el coeficiente de una variable sea 0.

Al renglón que se selecciona para realizar este proceso se le llama renglón *pivote*.

Por ejemplo, sea el sistema de ecuaciones:

$$x_1 + 2x_2 + 3x_3 = 9$$
$$4x_1 + 5x_2 + 6x_3 = 24$$
$$3x_1 + x_2 + 2x_3 = 4$$

Las operaciones realizadas en la aplicación del método de Eliminación Gaussiana para la obtención de los valores de las variables se presentan a continuación:

$$\begin{array}{r} -4x_1 - 8x_2 - 12x_3 = -36 \\ 4x_1 + 5x_2 + 6x_3 = 24 \\ \hline -3x_2 - 6x_3 = -12 \end{array}$$

⇐Seleccionamos la primera ecuación como pivote y la multiplicamos por -4 y el resultado de ésta se la sumamos a la segunda. Esto simplificará el sistema.

$$\begin{array}{l} x_1 + 2x_2 + 3x_3 = 9 \\ 0x_1 - 3x_2 - 6x_3 = -12 \\ 3x_1 + x_2 + 2x_3 = 4 \end{array}$$

⇐La nueva ecuación encontrada, la sustituimos en el segundo renglón.

$$\begin{array}{l} x_1 + 2x_2 + 3x_3 = 9 \\ 0x_1 + x_2 + 2x_3 = 4 \\ 0x_1 - 5x_2 - 7x_3 = -23 \end{array}$$

⇐Luego, la primera se multiplica por -3 y se le suma a la tercera; y la segunda ecuación se divide entre -3.

$$\begin{array}{l} x_1 + 2x_2 + 3x_3 = 9 \\ 0x_1 + x_2 + 2x_3 = 4 \\ 0x_1 + 0x_2 + 3x_3 = -3 \end{array}$$

⇐Ahora se multiplica la segunda por 5 y se le suma a la tercera.

⇐En este momento ya tenemos el valor de x3

$$\begin{array}{l} x_3 = \frac{-3}{3} = -1 \\ x_2 = 4 - 2x_3 = 4 - 2(-1) = 6 \\ x_1 = 9 - 3x_3 - 2x_2 = 9 - 3(-1) - 2(6) = 0 \end{array}$$

⇐Ahora simplemente se procede a hacer la sustitución hacia atrás, y automáticamente se van obteniendo los valores de las otras incógnitas

El pseudocódigo para la triangulación de un sistema de *n* ecuaciones, con *n* incógnitas es:

```
DESDE   i =  1 HASTA  n
  DESDE   j = i+1 HASTA  n
    cte = A(j,i)/A(i,i)
    DESDE  k = i  HASTA  n
      A(j,k) = A(j,k)-cte*A(i,k)
    FIN
    B(j) = B(j)-cte*B(i)
  FIN
FIN
```

donde: *A* es la matriz de los coeficientes de las incógnitas y *B* es el vector de términos independientes.

NOTA: Las limitantes que presenta este pseudocódigo es que si encuentra algún cero dentro de la diagonal principal, mientras va haciendo la eliminación, el algoritmo no podrá continuar ya que tratará de realizar una división entre cero.

Método De Gauss-Jordan

Este método constituye una variación del método de Eliminación de Gauss. Este procedimiento se diferencia del método Gaussiano, porque ahora no se busca obtener una matriz triangular, sino obtener la matriz identidad, siguiendo los mismos pasos de multiplicar por constantes los renglones y sumarlos a los demás.

Por ejemplo, sea el siguiente sistema de ecuaciones:

$$\begin{matrix} x+2y+3z=9 \\ 4x+5y+6z=24 \\ 3x+y+2z=4 \end{matrix} \xrightarrow{paso1} \begin{bmatrix} 1 & 2 & 3 & 9 \\ 4 & 5 & 6 & 24 \\ 3 & 1 & 2 & 4 \end{bmatrix} \xrightarrow[R_3=r_3-3r_1]{R_2=r_2-4r_1} \begin{bmatrix} 1 & 2 & 3 & 9 \\ 0 & -3 & -6 & -12 \\ 0 & -5 & -7 & -23 \end{bmatrix} \rightarrow$$

$$\xrightarrow{R_2=\frac{1}{3}r_2} \begin{bmatrix} 1 & 2 & 3 & 9 \\ 0 & 1 & 2 & 4 \\ 0 & -5 & -7 & -23 \end{bmatrix} \xrightarrow[R_3=r_3+5r_2]{R_1=r_1-2r_2} \begin{bmatrix} 1 & 0 & -1 & 1 \\ 0 & 1 & 2 & 4 \\ 0 & 0 & 3 & -3 \end{bmatrix} \xrightarrow{R_3=\frac{1}{3}r_3} \begin{bmatrix} 1 & 0 & -1 & 1 \\ 0 & 1 & 2 & 4 \\ 0 & 0 & 1 & -1 \end{bmatrix} \rightarrow$$

$$\xrightarrow[R_2=r_2-2r_2]{R_1=r_1+r_3} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & -1 \end{bmatrix} \xrightarrow{solución\ final} \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 6 \\ 0 & 0 & z & -1 \end{bmatrix}$$

El pseudocódigo que realiza el método de Gauss-Jordan es:

```
DESDE  i =  1 HASTA  n
  Se normaliza el renglón pivote
  cte = A(i,i)
  DESDE  j = 1 HASTA  n
    A(i,j) = A(i,j)/cte
  FIN
  B(i)=B(i)/cte

  Sigue el algoritmo de eliminación gaussiana
  DESDE  j = i+1 HASTA  n
    cte = A(j,i)
    DESDE  k = 0 HASTA  n
      A(j,k) = A(j,k)-cte*A(i,k)
    FIN
    B(j) = B(j)-cte*B(i)
  FIN
FIN

Se realiza la eliminación inversa
DESDE  i = n  hasta 1
  DESDE  j = (i-1) hasta 1
    cte = A(j,i)
    A(j,i) = A(j,i)-cte*A(i,i)
    B(j) = B(j)-cte*B(i)
  FIN
FIN
```

Como se puede observar, es una modificación del pseudocódigo que se utilizó para la eliminación gaussiana; se le agregaron líneas de código tanto para normalizar el renglón pivote requerido en la eliminación hacia abajo, así como para la eliminación inversa y obtener la solución completa del sistema.

NOTA: Este pseudocódigo, al igual que el usado para la eliminación gaussiana, tiene la particularidad que no admite tampoco ceros en la diagonal principal, cuando realiza la eliminación hacia abajo.

EJERCICIOS PROPUESTOS

1. Elaborar un programa que implemente la Eliminación Gaussiana para un sistema de ecuaciones lineales de 3 x 3.

2. Utilizando el programa anterior, resolver el siguiente sistema de ecuaciones.

$$\begin{aligned} 5x_1 + 3x_2 - 8x_3 &= 85.3 \\ -x_1 + 4x_2 - 6x_3 &= 14.32 \\ 4x_1 - 6x_2 + x_3 &= 17.61 \end{aligned}$$

3. Modificar el programa del inciso 1 para que solucione la limitante de no aceptar ceros en la diagonal principal, intercambiando el renglón con el cero, por otro de los que están abajo del mismo. Y verificar con el siguiente sistema:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 2.4 \\ -2x_1 - 4x_2 + 8x_3 &= 6.4 \\ 3x_1 + x_2 + 2x_3 &= 7.6 \end{aligned}$$

4. Utilizando el sistema de ecuaciones del inciso 2, resolverlo por Gauss-Jordan.

5. Elaborar un programa que implemente el método de Gauss-Jordan, para sistemas de hasta 5 incógnitas. Y que resuelva la limitante de ceros en la diagonal principal, intercambiando por otro renglón por debajo de la característica. Utilice el sistema de ecuaciones del inciso 3, para su comprobación.

6. Con el programa anterior resolver el siguiente sistema de ecuaciones.

$$\begin{aligned} 4x_1 + 2x_2 + x_3 + 2x_4 + 6x_5 &= 22.26 \\ 5x_1 + x_2 + x_3 - 3x_4 + 8x_5 &= 9.95 \\ 5x_1 + 7x_2 + 2x_3 + 5x_4 + 2x_5 &= 54.81 \\ x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 &= 7.49 \\ 3x_1 + 2x_2 + 6x_3 - 8x_4 - 11x_5 &= 76.81 \end{aligned}$$

7. Elaborar un programa que lea de un archivo un sistema de ecuaciones de 3 x 3 junto con el vector $\bar{b}$ y que lo resuelva por Gauss-Jordan.

SOLUCIÓN NUMÉRICA DE SISTEMAS DE ECUACIONES LINEALES (PARTE II)

Objetivos

El alumno conocerá y aplicará diferentes métodos de solución numérica para la resolución de sistemas de ecuaciones lineales.

Al final de esta práctica el alumno podrá:

9. *Resolver sistemas de ecuaciones lineales mediante diversas técnicas de solución numérica (Método de LU, Método de Gauss-Seidel)*
10. *Implementar dichas técnicas en el lenguaje de programación C*

Antecedentes

15. Manejar ciclos de repetición en lenguaje C
16. Manejar arreglos y estructuras en lenguaje C
17. Realizar operaciones con matrices
18. Calcular la matriz inversa

Introducción

En la solución numérica de sistemas de ecuaciones lineales, existen varios métodos numéricos como el Método de LU y el Método de Gauss-Seidel. En esta práctica emplearemos dichos métodos, por lo que se da una pequeña explicación de cada uno de ellos; aunado a esto se recomienda revisar estos métodos en la bibliografía citada al final de la práctica.

Método de LU

El método de descomposición LU para la solución de sistemas de ecuaciones lineales debe su nombre a que se basa en la descomposición de la matriz original de coeficientes (A) en el producto de dos matrices (L y U).

Esto es: **A= LU**

Donde:
L: Matriz triangular inferior
U: Matriz triangular superior.

Estas matrices son triangulares, y para facilidad de cálculo, cualquiera de ellas puede contener sólo unos en la diagonal principal. En esta práctica seleccionamos a la matriz U con dicha característica.

Por ejemplo, para matrices de 3x3 se escribe:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

Si efectuamos la multiplicación de L y U, igualando los elementos de ese producto con los de la matriz A correspondientes, se obtiene:

$$a_{11} = l_{11} \qquad a_{12} = l_{11}u_{12} \qquad a_{13} = l_{11}u_{13}$$

$$a_{21} = l_{21} \qquad a_{22} = l_{21}u_{12} + l_{22} \qquad a_{23} = l_{21}u_{13} + l_{22}u_{23}$$

$$a_{31} = l_{31} \qquad a_{32} = l_{31}u_{12}l_{32} \qquad a_{33} = l_{31}u_{13} + l_{32}u_{23} + l_{33}$$

De aquí que los elementos de L y U son, en este caso:

$$l_{11} = a_{11} \qquad u_{12} = a_{12}/l_{11} \qquad u_{13} = a_{13}/l_{11}$$

$$l_{21} = a_{21} \qquad l_{22} = a_{22} - l_{21}u_{12} \qquad u_{23} = (a_{23} - l_{21}u_{13})/l_{22}$$

$$l_{31} = a_{31} \qquad l_{32} = a_{32} - l_{31}u_{12} \qquad l_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23}$$

Si el sistema de ecuaciones original se escribe como A x = b, resulta lo mismo escribir L U x = b. Definiendo a U x = Y, podemos escribir L Y = b

Resolviendo para Y, encontramos:

$$y_1 = b_1/l_{11}$$

$$y_2 = (b_2 - l_{21}y_1)/l_{22}$$

$$y_3 = (b_3 - l_{31}y_1 - l_{32}y_2)/l_{33}$$

Una vez conocidas L, U y b, prosigue encontrar primeramente los valores de "Y" por sustitución progresiva sobre "L Y = b". Posteriormente se resuelve "U x = Y " por sustitución regresiva para encontrar los valores de "x", obteniendo:

$$x_3 = y_3$$

$$x_2 = y_2 - u_{23}x_3$$

$$x_1 = y_1 - u_{12}x_2 - u_{13}x_3$$

Por ejemplo, sea el siguiente sistema de ecuaciones, factorizando la matriz en LU:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 3 & 1 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ 24 \\ 4 \end{bmatrix} \rightarrow$$

Las matrices de factores L y U de A son:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 4 & -3 & 0 \\ 3 & -5 & -9 \end{bmatrix}; \quad U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

Resolver la ecuación L Y = b por sustitución progresiva para obtener los elementos del vector auxiliar Y: [L][Y]=[b]

$$\rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 4 & -3 & 0 \\ 3 & -5 & 3 \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 9 \\ 24 \\ 4 \end{bmatrix}$$

Donde:

$$y_1 = 9$$

$$y_2 = 4$$

$$y_3 = -1$$

Elaborada por:
Ing. Laura Sandoval Montaño
Viridiana del Carmen De Luna Bonilla
Virgilio Green Pérez

Resolver la ecuación U x = Y para encontrar los elementos de x, por sustitución regresiva:

[U][x]=[Y]

$$\rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 9 \\ 4 \\ -1 \end{bmatrix}$$

De donde se obtiene:

$$x_1 = 0$$
$$x_2 = 6$$
$$x_3 = -1$$

A continuación se presenta el algoritmo, en pseudocódigo, de la descomposición de la matriz A en L y U, donde el programador puede seleccionar la matriz que contendrá la diagonal principal unitaria (como se mencionó anteriormente).

DESDE k = 1 HASTA n
  U(k,k)=1
  L(k,k) = $A(k,k) - \sum_{s=1}^{k-1} L(k,s) * U(s,k)$
  DESDE i = (k+1) HASTA n
    $L(i,k) = A(i,k) - \sum_{s=1}^{k-1} L(i,s) * U(s,k)$
  FIN
  DESDE j = (k+1) HASTA n
    $U(k,j) = \dfrac{A(k,j) - \sum_{s=1}^{k-1} L(k,s) * U(s,j)}{L(k,k)}$
  FIN
FIN

Método de Gauss-Seidel

El método de Gauss-Seidel, es un método iterativo y por lo mismo, resulta ser un método bastante eficiente.
El paso inicial de este método consiste en despejar por cada ecuación una variable diferente, así se obtiene un conjunto de ecuaciones. Considerando el siguiente sistema de ecuaciones lineales, se muestra a su derecha la ecuación resultante en la que de la ecuación 1 despejamos x_1, de la 2 despejamos x_2 y así sucesivamente.

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \qquad x_1 = \frac{b_1 - a_{22}x_2 - \ldots - a_{1n}x_n}{a_{11}}$$

$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \qquad x_2 = \frac{b_2 - a_{21}x_1 - \ldots - a_{2n}x_n}{a_{22}}$$

. .

. .

. .

$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \qquad x_n = \frac{b_n - a_{11}x_1 - \ldots - a_{nn-1}x_{n-1}}{a_{nn}}$$

Para comenzar el proceso iterativo, le damos un valor inicial a las variables x_2, x_3, …, x_n; sustituyendo estos valores en la ecuación de x_1, obtenemos un primer valor para éste; luego utilizamos este nuevo valor para sustituirlo en la siguiente ecuación y obtener un valor para x_2; y continuamos consecutivamente hasta llegar a x_n; con esto completamos la primera iteración. Aplicamos nuevamente las ecuaciones para obtener nuevos valores de x_1, x_2, …,x_n completando de esta manera otra iteración. Este proceso iterativo termina cuando, sacando la diferencia entre cada uno de los valores de la iteración actual con la anterior, se tiene un valor menor de tolerancia.

Por ejemplo, sea el siguiente sistema de ecuaciones lineales:

$$3x_1 + x_2 + 2x_3 = 4$$
$$4x_1 + 5x_2 + 6x_3 = 24$$
$$1x_1 + 2x_2 + 3x_3 = 9$$

Despejamos x_1, x_2 y x_3 de la ecuación 1, 2 y 3, respectivamente.

$$x_1 = \frac{4 - x_2 - 2x_3}{3}$$
$$x_2 = \frac{24 - 4x_1 - 6x_3}{5}$$
$$x_3 = \frac{9 - x_1 - 2x_2}{3}$$

$$x_1 = \frac{4 - 0 - 2(0)}{3} = 1.33333$$

$$x_2 = \frac{24 - 4(1.33333) - 6(0)}{5} 3.73333$$

$$x_3 = \frac{9 - (1.33333) - 2(3.73333)}{3} = 0.06667$$

←Damos valores iniciales a x_2 y x_3 de cero y obtenemos un primer valor de x_1.
←Obtenemos un valor para x_2, utilizando x_3 y el nuevo valor de x_1.
←Y luego para x_3, utilizamos los nuevos valores de x_1 y x_2.

De la primera iteración tenemos que:

$x_1 = 1.33333$
$x_2 = 3.73333$
$x_3 = 0.06667$

Iteración 2

$$x_1 = \frac{4 - (3.73333) - 2(0.06667)}{3} = 0.04444$$

$$x_2 = \frac{24 - 4(0.04444) - 6(0.06667)}{5} = 4.68444$$

$$x_3 = \frac{9 - (0.04444) - 2(4.68444)}{3} = -0.13777$$

←Para la segunda iteración se usa $x_2 = 3.73333$ y $x_3 = 0.06667$ para obtener los nuevos valores. Y procedemos de la misma manera que en la iteración anterior.

Calculamos los errores: para una tolerancia de 0.1

Error x_1:

$|0.04444 - 1.3333| = 1.288$

Error x_2:

$|4.68444 - 3.7333| = 0.95111$

Error x_3:

$|-0.13777 - 0.06666| = 0.20444$

Como los errores obtenidos, en general, son todavía grandes, continuamos con la siguiente iteración, realizando el mismo procedimiento.

Iteración 3

$$x_1 = \frac{4 - (4.68444) - 2(-0.13777)}{3} = -0.13629$$

$$x_2 = \frac{24 - 4(-0.13629) - 6(-0.13777)}{5} = 5.07437$$

$$x_3 = \frac{9 - (-0.13629) - 2(5.07437)}{3} = -0.33748$$

Error x_1:

$|-0.13629 - (-0.0444)| = 0.18074$

Error x_2:

$|5.07437 - 4.68444| = 0.38992$

Error x_3:

$|-0.33748 - (-0.13777)| = 0.19970$

errores mayores a la tolerancia de 0.1

Iteración 4

$$x_1 = \frac{4 - 5.07437 - 2(-0.33748)}{3} = -0.13313$$
$$x_2 = \frac{24 - 4(-0.13313) - 6(-0.33748)}{5} = 5.31148$$
$$x_3 = \frac{9 - (-0.13313) - 2(5.31148)}{3} = -0.49661$$

Error x_1:
$|-0.13313 - (-0.13629)| = 0.00316$
Error x_2:
$|5.31148 - 5.07437| = 0.23711$
Error x_3:
$|-0.49661 - (-0.33748)| = 0.15913$
errores mayores a la tolerancia de 0.1

Iteración 5

$$x_1 = \frac{4 - (5.31148) - 2(-0.49661)}{3} = -0.10608$$
$$x_2 = \frac{24 - 4(-0.10608) - 6(-0.49661)}{5} = 5.48080$$
$$x_3 = \frac{9 - (-0.10608) - 2(5.48080)}{3} = -0.61850$$

Error x_1:
$|-0.10608 - (-0.13313)| = 0.27048$
Error x_2:
$|5.48080 - 5.31148| = 0.16931$
Error x_3:
$|-0.61850 - (-0.49661)| = 0.12189$
errores mayores a la tolerancia de 0.1

Iteración 6

$$x_1 = \frac{4 - (5.48080) - 2(-0.61850)}{3} = -0.08126$$
$$x_2 = \frac{24 - 4(-0.08126) - 6(-0.61850)}{5} = 5.60721$$
$$x_3 = \frac{9 - (-0.08126) - 2(5.60721)}{3} = -0.71105$$

Error x_1:
$|-0.08126 - (-0.10608)| = 0.02482$
Error x_2:
$|5.60721 - 5.48080| = 0.12641$
Error x_3:
$|-0.71105 - (-0.61850)| = 0.09255$
errores mayores a la tolerancia de 0.1

Iteración 7

$$x_1 = \frac{4 - (5.60721) - 2(-0.71105)}{3} = -0.06170$$
$$x_2 = \frac{24 - 4(-0.06170) - 6(-0.71105)}{5} = 5.70263$$
$$x_3 = \frac{9 - (-0.06170) - 2(5.70263)}{3} = -0.78118$$

Error x_1:
$|-0.06170 - (-0.08126)| = 0.01956$
Error x_2:
$|5.70263 - 5.60721| = 0.09541$
Error x_3:
$|-0.78118 - (-0.71105)| = 0.07012$
errores menores a la tolerancia de 0.1

Como la tolerancia se cumple en estos tres casos, aquí termina el proceso de iteraciones. Entonces tenemos una solución del sistema de ecuaciones:

$x_1 = -0.06170$

$x_2 = 5.70263$

$x_3 = -0.78118$

La tolerancia fijada en este ejercicio es muy grande por lo que los resultados no están muy cercanos al valor exacto:

$x_1 = 0$

$x_2 = 6$

$x_3 = -1$

Se recomienda utilizar una tolerancia mucho menor.

EJERCICIOS PROPUESTOS

8. Elaborar un programa que implemente el método de LU, para resolver sistemas de 3 incógnitas.

9. Con el programa anterior, resolver el siguiente sistema de ecuaciones.

$$\begin{aligned} 5x_1 + 3x_2 - 8x_3 &= 85.3 \\ -x_1 + 4x_2 - 6x_3 &= 14.32 \\ 4x_1 - 6x_2 + x_3 &= 17.61 \end{aligned}$$

10. Elaborar un programa que implemente el método de Gauss-Seidel, para resolver sistemas de 3 incógnitas. *TIP: Sólo es implementar en funciones, las ecuaciones resultantes del despeje de las ecuaciones del sistema.*

11. Utilizando el programa anterior, resolver el sistema de ecuaciones del inciso 2. Recordar que el sistema de ecuaciones debe estar acomodado.

12. Resolver el siguiente sistema de ecuaciones utilizando el método de LU. *TIP: Implementar el algoritmo visto anteriormente y obtener ciclos adicionales para calcular Y y X.*

$$\begin{aligned} 4x_1 + 2x_2 + x_3 + 2x_4 + 6x_5 &= 22.26 \\ 5x_1 + x_2 + x_3 - 3x_4 + 8x_5 &= 9.95 \\ 5x_1 + 7x_2 + 2x_3 + 5x_4 + 2x_5 &= 54.81 \\ x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 &= 7.49 \\ 3x_1 + 2x_2 + 6x_3 - 8x_4 - 11x_5 &= 76.81 \end{aligned}$$

INTERPOLACIÓN NUMÉRICA

Objetivos

El alumno conocerá, aplicará y comparará algunos métodos de interpolación numérica de funciones.

Al final de esta práctica el alumno podrá:

11. *Obtener una función que contenga un conjunto dado de puntos en un plano, utilizando los métodos de interpolación de Lagrange y Newton.*
12. *Implementar dichos métodos en el lenguaje de programación C*

Antecedentes

19. Dominar el uso de ciclos de repetición en lenguaje C
20. Manejar arreglos y estructuras en lenguaje C
21. Haber creado y utilizado funciones de tipo entero y real en lenguaje C
22. Manejar operaciones con polinomios

Introducción

De un conjunto de puntos en un plano, pueden existir varias funciones que unan dichos puntos. Este tipo de funciones nos ayuda a modelar sistemas físicos.
Dichas funciones se pueden obtener por diversos métodos, y nos ayudan a calcular valores intermedios entre el conjunto de puntos dado; a este proceso, se le llama interpolación.

Polinomio de Interpolación con Diferencias Divididas de Newton.

Este método se basa en la obtención de un polinomio a partir de un conjunto de puntos dado, aproximándose lo más posible a la curva buscada.

La ecuación general para la obtención de la función por este método es:

$$f_n(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \ldots + b_n(x - x_0)(x - x_1)\ldots(x - x_{n-1})$$

Donde las 'b_i' se obtienen mediante la aplicación de una serie de funciones incluidas en una tabla de diferencias.

NOTA: Con este método, haciendo la multiplicación de los binomios para cada diferencia, y simplificando términos, es posible encontrar un polinomio característico al conjunto de puntos dado.

Por ejemplo, suponiendo que tenemos 4 puntos, la tabla de diferencias tiene la siguiente forma:

x	f(x)	Primera Diferencia	Segunda Diferencia	Tercera Diferencia
x_0	$f(x_0)=b_0$	$f[x_1,x_0]=b_1=\frac{f(x_1)-f(x_0)}{x_1-x_0}$	$f[x_2,x_1,x_0]=b_2=\frac{f[x_2,x_1]-f[x_1,x_0]}{x_2-x_0}$	$f[x_3,x_2,x_1,x_0]=b_3$ $b_3=\frac{f[x_3,x_2,x_1]-f[x_2,x_1,x_0]}{x_3-x_0}$
x_1	$f(x_1)$	$f[x_2,x_1]=\frac{f(x_2)-f(x_1)}{x_2-x_1}$	$f[x_3,x_2,x_1]=\frac{f[x_3,x_2]-f[x_2,x_1]}{x_3-x_1}$	
x_2	$f(x_2)$	$f[x_3,x_2]=\frac{f(x_3)-f(x_2)}{x_3-x_2}$		
x_3	$f(x_3)$			

Con esto, la ecuación quedaría de la siguiente forma:

$$f_3(x)=b_0+b_1(x-x_0)+b_2(x-x_0)(x-x_1)+b_3(x-x_0)(x-x_1)(x-x_2)$$

Por ejemplo:
Si tenemos los siguientes puntos

x	f(x)
0.1	2.31
0.4	3.36
0.7	4.59
1	6

Calculamos su tabla

x	f(x)	Primera	Segunda	Tercera
0.1	2.31	3.5	1	0
0.4	3.36	4.1	1	
0.7	4.59	4.7		
1	6			

Obteniendo el siguiente polinomio:

$$f_3(x) = 2.31 + 3.5(x - 0.1) + 1(x - 0.1)(x - 0.4) + 0(x - 0.1)(x - 0.4)(x - 0.7)$$
$$f_3(x) = x^2 + 3x + 2$$

El algoritmo que calcula la tabla de diferencias para *N* puntos es el siguiente:

```
Lee  N pares de puntos y colócalos en la matriz M
DESDE QUE i = 1 HASTA N
  DESDE QUE j = 0 HASTA (N-i)
    M[j][i+1]=(M[j+1][i]-M[j][i])/(M[j+i][0]-M[j][0])
  FIN
FIN
Imprime en pantalla la tabla de las diferencias calculadas.
```

El programa utiliza una matriz *M* de *N* renglones por *N+1* columnas, donde *N* es el número de puntos dados para obtener el polinomio de interpolación.

TIP: Para poder calcular la aproximación dentro del mismo programa, es recomendable usar un arreglo de N elementos que tenga como primer elemento un 1, en el segundo ($x-x_0$), en el tercero, el resultado multiplicar el segundo elemento del vector con ($x-x_1$), y así sucesivamente para solo multiplicar las b_i de la tabla por este vector y obtener la aproximación en el punto deseado (x).

Método de Lagrange

La interpolación permite el cálculo de valores intermedios de datos experimentales los cuales no tienen una función que los represente. El método más común para interpolar valores intermedios, es la interpolación polinomial, la cual consiste en determinar el polinomio de orden ***n*** que ajusta a ***n+1*** datos. La interpolación de Lagrange es una de las alternativas más atractivas que existe para interpolar, debido a la facilidad de programar.

La interpolación de Lagrange se expresa de la siguiente manera:

$$f_n(X) = \sum_{i=0}^{n} Li(X) f(X_i)$$

en donde: $L_i(X) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{X - X_j}{X_i - X_j}$, $\prod$ denota el *"producto de"*.

Por ejemplo, siguiendo la fórmula, para el polinomio de 1er orden (n = 1) es:

$$f_1(X) = \frac{X - X_1}{X_0 - X_1} f(X_0) + \frac{X - X_0}{X_1 - X_0} f(X_1)$$

y para el polinomio de segundo orden es:

$$f_2(X) = \frac{(X - X_1)(X - X_2)}{(X_0 - X_1)(X_0 - X_2)} f(X_0) + \frac{(X - X_0)(X - X_2)}{(X_1 - X_0)(X_1 - X_2)} f(X_1) + \frac{(X - X_0)(X - X_1)}{(X_2 - X_0)(X_2 - X_1)} f(X_2)$$

para un polinomio de 3er orden sería:

$$f_2(X) = \frac{(X - X_1)(X - X_2)(X - X_3)}{(X_0 - X_1)(X_0 - X_2)(X_0 - X_3)} f(X_0) + \frac{(X - X_0)(X - X_2)(X - X_3)}{(X_1 - X_0)(X_1 - X_2)(X_1 - X_3)} f(X_1)$$

$$+ \frac{(X - X_0)(X - X_1)(X - X_3)}{(X_2 - X_0)(X_2 - X_1)(X_2 - X_3)} f(X_2) + \frac{(X - X_0)(X - X_1)(X - X_2)}{(X_3 - X_0)(X_3 - X_1)(X_3 - X_2)} f(X_3)$$

Ejemplo:

Usando un polinomio de interpolación de Lagrange de primer y segundo orden, evaluar **X=0.2** con base en los datos:

x	f(x)
0.1	2.31
0.4	3.36
0.7	4.59
1	6

Solución:

El polinomio de primer orden es: $f_1(X) = \frac{X - X_1}{X_0 - X_1} f(X_0) + \frac{X - X_0}{X_1 - X_0} f(X_1)$

Sustituyendo los valores de la tabla, el polinomio de primer orden queda como sigue:

$$f_1(X) = \frac{X - 0.4}{0.1 - 0.4}(2.31) + \frac{X - 0.1}{0.4 - 0.1}(3.36)$$

Sustituyendo X = 0.2:

$$f_1(X) = \frac{0.2 - 0.4}{0.1 - 0.4}(2.31) + \frac{0.2 - 0.1}{0.4 - 0.1}(3.36) = 1.54 + 1.12 = 2.66$$

De manera similar, se desarrolla el polinomio de segundo orden:

$$f_2(X) = \frac{(X - X_1)(X - X_2)}{(X_0 - X_1)(X_0 - X_2)} f(X_0) + \frac{(X - X_0)(X - X_2)}{(X_1 - X_0)(X_1 - X_2)} f(X_1) + \frac{(X - X_0)(X - X_1)}{(X_2 - X_0)(X_2 - X_1)} f(X_2)$$

Sustituyendo los valores de la tabla, el polinomio de segundo orden queda como:

$$f_2(X) = \frac{(X - 0.4)(X - 0.7)}{(0.1 - 0.4)(0.1 - 0.7)}(2.31) + \frac{(X - 0.1)(X - 0.4)}{(0.4 - 0.1)(0.4 - 0.7)}(3.36) + \frac{(X - 0.1)(X - 0.4)}{(0.7 - 0.1)(0.7 - 0.4)}(4.59)$$

Sustituyendo X = 0.2

$$f_2(X) = \frac{(0.2 - 0.4)(0.2 - 0.7)}{(0.1 - 0.4)(0.1 - 0.7)}(2.31) + \frac{(0.2 - 0.1)(0.2 - 0.4)}{(0.4 - 0.1)(0.4 - 0.7)}(3.36) + \frac{(0.2 - 0.1)(0.2 - 0.4)}{(0.7 - 0.1)(0.7 - 0.4)}(4.59) = 2.64$$

Para el polinomio de tercer orden, sustituyendo los valores de la tabla en la ecuación general, tenemos que:

$$f_2(X) = \frac{(X - 0.4)(X - 0.7)(X - 1.0)}{(0.1 - 0.4)(0.1 - 0.7)(0.1 - 1.0)}(2.31) + \frac{(X - 0.1)(X - 0.7)(X - 1.0)}{(0.4 - 0.1)(0.4 - 0.7)(0.4 - 1.0)}(3.36)$$

$$+ \frac{(X - 0.1)(X - 0.4)(X - 1.0)}{(0.7 - 0.1)(0.7 - 0.4)(0.7 - 1.0)}(4.59) + \frac{(X - 0.1)(X - 0.4)(X - 0.7)}{(1.0 - 0.1)(1.0 - 0.4)(1.0 - 0.7)}(6) = -1.897$$

Sustituyendo X= = 0.2

$$f_2(X) = \frac{(0.2 - 0.4)(0.2 - 0.7)(0.2 - 1.0)}{(0.1 - 0.4)(0.1 - 0.7)(0.1 - 1.0)}(2.31) + \frac{(0.2 - 0.1)(0.2 - 0.7)(0.2 - 1.0)}{(0.4 - 0.1)(0.4 - 0.7)(0.4 - 1.0)}(3.36)$$

$$+ \frac{(0.2 - 0.1)(0.2 - 0.4)(0.2 - 1.0)}{(0.7 - 0.1)(0.7 - 0.4)(0.7 - 1.0)}(4.59) + \frac{(0.2 - 0.1)(0.2 - 0.4)(0.2 - 0.7)}{(1.0 - 0.1)(1.0 - 0.4)(1.0 - 0.7)}(6) = -1.897$$

EJERCICIOS PROPUESTOS

1. Hacer un programa que calcule el polinomio de primer orden de Lagrange.
2. Dada la función tabular, hacer un programa que calcule el polinomio de interpolación de Lagrange de primer y segundo orden, para aproximar en x=2.

x	f(x)
0	2
1	3
4	18
6	38

3. Implemente el algoritmo para el método de Newton de Diferencias Divididas para N puntos.
4. De los siguientes puntos:

x	f(x)
0.1	5.77
1	6
1.5	6.61
2	7.39

   Utilice el programa anterior para obtener los coeficientes del polinomio. Y obtenga el valor de la función para x = 1.1

5. De la siguiente tabla:

x	f(x)
-1	-2.28
0	0
1	2.28
1.2	0.935

   Resuelva utilizando Lagrange para tercer orden y Newton; compare resultados para x = 0.8.

6. Hacer un programa que implemente el método de interpolación de Lagrange, es decir la fórmula: $f_n(X) = \sum_{i=0}^{n} Li(X)\, f(X_i)$

INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS (PARTE I)

Objetivos

El alumno conocerá y aplicará el concepto de programación orientada a objetos para la realización de programas.

Al final de esta práctica el alumno podrá:

Implementar programas orientados a objetos, que utilicen los conceptos del paradigma de la programación orientada a objetos.

Antecedentes

23. Tener las bases de la programación orientada a objetos.
24. Manejar sentencias de control de flujo en algún lenguaje de programación

Introducción

Para entender el paradigma de programación orientada a objetos, es necesario primeramente definir algunos conceptos.

- Objeto.- Es un conjunto de atributos y métodos que conforman una entidad.
- Clase.- Es un tipo de objeto definido por el usuario. Es decir, una clase, visto desde la programación estructurada, es como un nuevo tipo de dato, creado por el usuario.
- Atributos.- Son las características que tiene un objeto. Por ejemplo, un lapicero tendría como atributos marca, color, diámetro de puntilla, número de puntillas, goma, etc.
- Métodos.- Son todas las acciones que puede realizar un objeto. Por ejemplo, el lapicero tendría como método *escribir()*;si además tuviera goma, también podría tener como método *borrar()*.

En la programación orientada a objetos se definió, por convención, una nomenclatura para escribir los programas, y así identificar a simple vista, clases, atributos y métodos:

1) No debe existir ningún espacio en blanco, ni guiones bajos dentro de los nombres.
2) En el caso de las clases, la primera letra de cada palabra que conforma el nombre, deberá ser mayúscula, por ejemplo: RaizCuadrada, MetodoNewton, etc.
3) Para los nombres de atributos, las letras de todo el nombre deberán ser minúsculas; en el caso de que incluya varias palabras, a partir de la segunda deberá iniciar con mayúscula; por ejemplo: banderaIf, parteReal, parteImaginaria, anguloExpresionPolar, etc.
4) Los nombres de métodos, siguen la misma nomenclatura que con los atributos, la diferencia es que los métodos llevan un par de paréntesis después del nombre.

A continuación se presenta el clásico programa que imprime “hola mundo”, escrito en el lenguaje de programación Java

Ejemplo 1
Imprimir “hola mundo” en pantalla

```
public class HolaMundo{
   public static void main(String args[]){
      System.out.println("hola mundo");
   }
}
```

Se puede observar que este programa inicia definiendo una clase. Esto ocurre siempre que se elabora un programa escrito en Java. En este caso, se definió una clase que se llama HolaMundo:

```
public class HolaMundo{
}
```

Como se indicó anteriormente, toda clase genera objetos, y estos objetos deben tener atributos y métodos. Pero cuando se requiere ejecutar una clase en lugar de crear un objeto, ésta debe contener un método ***main***, y siempre se define de la misma forma.

```
public static void main(String args[]){
}
```

Y el equivalente al ***printf()*** de C, en Java es:

```
System.out.println();
```

Habiendo visto las bases de lo que conforma una clase, ahora se establecerán pasos en el diseño, desarrollo e implementación de una clase.

En el diseño de programas orientados a objetos, existe una serie de pasos a seguir:

a) Leer y analizar el enunciado.
b) Identificar las clases a utilizar.
c) Para cada clase definir atributos y métodos.
d) Elaborar el diagrama de clases.

Ejemplo 2

Elaborar un programa que defina una entidad Persona que tenga como datos nombre, dirección, teléfono y edad; además que tenga como acciones caminar, correr y hablar por teléfono. Posteriormente que cree dos instancias de esta entidad y a cada una le asigne valores a sus datos e invoque a las actividades definidas.

a) Leer y analizar el enunciado.
Se requiere implementar una clase llamada Persona y crear dos objetos a partir de dicha clase.

b) Identificar las clases a utilizar.
Sólo se necesita una clase, en este caso Persona.

c) Para cada clase definir atributos y métodos.
De acuerdo al enunciado los datos de la entidad son los atributos y los métodos son las acciones que realiza la entidad.

Atributos:
1) nombre, que será de tipo String.
2) direccion, que será de tipo String
3) telefono, que será de tipo String
4) edad, que será de tipo int

Métodos:
1) caminar()
2) correr()
3) hablarPorTelefono()

d) Elaborar el diagrama de clases

Un diagrama de clases representa, de manera esquemática, a las clases y las relaciones que hay entre ellas. Cada clase se visualiza como una caja con tres secciones, donde en la primera sección se pone el nombre de la clase, en la segunda sección los atributos de ésta, y en la tercera sus métodos.

Por lo que el diagrama de clases para este ejemplo queda de la siguiente forma:

Persona
String nombre String direccion String telefono int edad
caminar() correr() hablarPorTelefono()

Al llegar a este punto, se puede empezar a programar. Partiendo del esqueleto básico visto en el ejemplo anterior, definimos la clase y el método *main*.

```
public class Persona{
   public static void main(String args[]){
   }
}
```

Ahora sobre esta base, se le agregan los atributos y métodos.

```
public class Persona{
   String nombre;
   String direccion;
   String telefono;
   int edad;
   public static void main(String args[]){
   }
   public void caminar(){
   }
   public void correr(){
   }
   public void hablarPorTelefono(){
   }
}
```

Una vez definida la clase, se agregará código para crear objetos.

Para generar un nuevo objeto se utiliza un *constructo*r. Cuando se define una clase, se genera un *constructo*r por omisión, con el mismo nombre que la clase; este constructor es invocado a través de la palabra *new*. Por lo tanto, para crear un objeto es similar a declarar una variable en lenguaje estructurado: se pone el tipo de dato, que en este caso es el nombre de la clase, después la variable que será el nombre del objeto, y luego, a través de una asignación, se invoca al constructor.

Clase variableObjeto = new Clase();

Para el ejemplo, el código para la creación de los dos objetos requeridos es:

```
Persona p1 = new Persona();
Persona p2 = new Persona();
```

Así, cuando se escribe new Persona(), se invoca al constructor Persona() para crear un objeto de la clase Persona. Este objeto creado se manipulará a través de la variable correspondiente (p1 o p2).

Como ya se mencionó, cuando se crea una clase se genera automáticamente un método constructor por omisión, por lo que no es necesario ponerlo explícitamente. Para este ejemplo se pondrá de forma explícita para mostrar cómo se define y funciona. El código del método constructor del ejemplo se presenta a continuación.

```
public Persona(){
   System.out.println("Creando un objeto");
}
```

Hasta este punto, el código se muestra a continuación:

```
public class Persona{

   public Persona(){
      System.out.println("Creando un objeto");
   }

   String nombre;
   String direccion;
   String telefono;
   int edad;
   public static void main(String args[]){
      Persona p1 = new Persona();
      Persona p2 = new Persona();
   }
   public void caminar(){
   }
   public void correr(){
   }
   public void hablarPorTelefono(){
   }
}
```

Si se compila y ejecuta este código, se tendrá una salida como la siguiente:

```
Creando un objeto
Creando un objeto
```

Esta salida se debe a que el constructor además de crear un objeto de la clase Persona imprime el mensaje: `Creando un objeto`

Ahora se agregará código al método constructor para que al momento de que genere el objeto le asigne valores a sus atributos. Para realizar esto, el método constructor recibirá como parámetros los valores de los atributos; esto es similar al uso de parámetros en funciones de un lenguaje estructurado. El código final del constructor se muestra a continuación.

```
public Persona(String nom, String dir, String tel, int ed){
   System.out.println("Creando un objeto");
   this.nombre = nom;
   this.direccion = dir;
   this.telefono = tel;
   this.edad = ed;
}
```

Como se observa en este código, los parámetros tienen indicado el tipo y van separados por coma. Además se ha introducido una nueva palabra: ***this***; esta palabra, anteponiéndola a los nombres de los atributos, sirve para indicar que los atributos son del objeto que está invocando al método.

De esta forma, cuando se invoque al constructor, se deberán dar los valores como argumentos. En este ejemplo, las líneas de código para crear un objeto, al tiempo que se le asignan valores a los atributos, se muestran a continuación.

```
  Persona p1 = new Persona("Juan Perez","Azucenas #12","12345678",28);
  Persona p2 = new Persona("Ana Lopez","Cipreses #45","87654321",24);
```

Una vez asignados valores a los atributos, ahora se desplegarán en pantalla todos estos datos, por lo que dentro de ***main***, agregaremos unas líneas de impresión en pantalla.

```
System.out.println("\n\nDatos de p1:");
System.out.println("Nombre: "+p1.nombre);
System.out.println("Direccion: "+p1.direccion);
System.out.println("Telefono: "+p1.telefono);
System.out.println("Edad: "+p1.edad);
System.out.println("\n\nDatos de p2:\nNombre: "+p2.nombre+"\nDireccion:
"+p2.direccion+"\nTelefono: "+p2.telefono+"\nEdad: "+p2.edad);
```

Nótese que las primeras 5 líneas, despliegan los datos de la persona 1, mientras que la sexta despliega los datos de la persona2, esto con el objeto de observar que cuando se manda a imprimir algo, lo que se hace es concatenar cadenas, estas cadenas pueden ser variables o segmentos de texto entre un par de comillas, y el signo +, es el que hace la concatenación. Cabe destacar, que cuando se va a hacer referencia a algún atributo o método de un objeto, se debe poner el nombre del objeto seguido de un punto, y luego el nombre del atributo o método al que hace referencia.

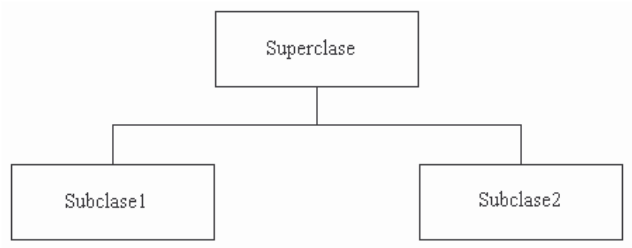
Ahora sólo falta agregarle algunas líneas de impresión a cada método para mandar llamar a los métodos y ver qué hacen los objetos. El código completo queda de la siguiente forma:

```
public class Persona{
   /*Se define un constructor explícito*/
   public Persona(String nom, String dir, String tel, int ed){
      System.out.println("Creando un objeto");
      this.nombre = nom;
      this.direccion = dir;
      this.telefono = tel;
      this.edad = ed;
   }

   /*Se declaran atributos*/
   String nombre;
   String direccion;
   String telefono;
   int edad;

   /*Método main de la clase*/
   public static void main(String args[]){

      /*Se crean los 2 objetos*/
      Persona p1 = new Persona("Juan Perez","Azucenas
#12","12345678",28);
```

```
      Persona p2 = new Persona("Ana Lopez","Cipreses #45","87654321",24);

      /*Se manda a imprimir la información de cada objeto*/
      System.out.println("\n\nDatos de p1:");
      System.out.println("Nombre: "+p1.nombre);
      System.out.println("Direccion: "+p1.direccion);
      System.out.println("Telefono: "+p1.telefono);
      System.out.println("Edad: "+p1.edad);
      System.out.println("\n\nDatos de p2:\nNombre: "+p2.nombre+
                         "\nDireccion: "+p2.direccion+"\nTelefono: "+
                         p2.telefono+"\nEdad: "+p2.edad);

      /*Se llaman a los métodos con alguno de los objetos*/
      p1.caminar();
      p2.hablarPorTelefono();
      p1.correr();
   }

   /*Se declaran los métodos*/
   public void caminar(){
      System.out.println(this.nombre+" se fue a caminar");
   }
   public void correr(){
      System.out.println(this.nombre+" se fue a correr");
   }
   public void hablarPorTelefono(){
      System.out.println(this.nombre+" esta hablando por telefono");
   }
}
```

Otro concepto de gran importancia en la programación orientada a objetos es:

- Herencia: Permite el acceso a atributos y métodos de una clase desde otra. Visto en una jerarquía de clases, se dice que las subclases heredan de las superclases. Una superclase puede tener varias subclases, pero una subclase solo puede tener una superclase. Vea el siguiente diagrama.



Ejemplo 3
Elaborar un programa que maneje los datos de un Empleado. Considere que el Empleado es una Persona y que además de tener los datos de Persona contiene datos propios de su condición de Empleado como lo son: clave de empleado, sueldo y puesto. Las actividades de empleado serán: atender a cliente y contactar a proveedores.

a) Leer y analizar el enunciado.

Como el Empleado es una Persona y debe contener los datos o atributos de ésta, entonces Empleado hereda de Persona. Para manipular los datos de Empleado se debe crear un objeto de este tipo ya que contiene datos y actividades adicionales a Persona.

b) Identificar las clases a utilizar.

Con base en el análisis, el Empleado es una subclase de la superclase Persona. Por lo que en este problema se contemplan dos clases: Persona y Empleado.

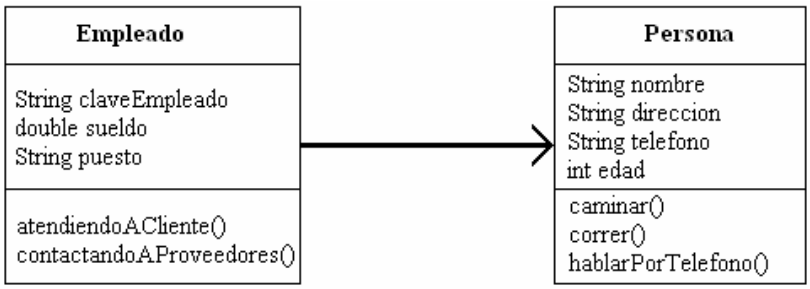
c) Para cada clase definir atributos y métodos.

Para la clase Empleado se necesitan los siguientes atributos y métodos.

Atributos:
1) nombre, direccion, telefono, edad: estos atributos ya se encuentran en Persona.
2) claveEmpleado
3) sueldo
4) puesto

Métodos:
atendiendoACliente()
contactandoAProveedores()

d) Elaborar el diagrama de clases.



En el diagrama se presenta la clase Empleado y se toma la clase Persona del ejemplo anterior. La línea flechada que une a las dos clases indica que empleado *usa* los atributos y métodos de Persona.

El código base de la clase Empleado se muestra a continuación

```
public class Empleado{
   String claveEmpleado;
   double sueldo;
```

```
   String puesto;
   public static void main(String args[]){
   }
   public void atendiendoACliente(){
   }
   public void contactandoAProveedores(){
   }
}
```

Con este código, solamente se está definiendo una clase llamada Empleado, pero todavía no se le heredan los atributos y métodos de Persona. Para hacer que la clase Empleado sea una subclase de Persona, se utiliza la palabra ***extends***, quedando el código como sigue.

```
public class Empleado extends Persona{
   String claveEmpleado;
   double sueldo;
   String puesto;
   public static void main(String args[]){
   }
   public void atendiendoACliente(){
   }
   public void contactandoAProveedores(){
   }
}
```

Cabe hacer notar que cuando se utilizan clases para hacer subclases, no deben existir constructores explícitos. Además, la superclase puede no tener método ***main***. Quitando el constructor y el método ***main*** del código del ejemplo anterior:

```
public class Persona{
   /*Se declaran atributos*/
   String nombre;
   String direccion;
   String telefono;
   int edad;

   /*Se declaran métodos*/
   public void caminar(){
      System.out.println(this.nombre+" se fue a caminar");
   }
   public void correr(){
      System.out.println(this.nombre+" se fue a correr");
   }
   public void hablarPorTelefono(){
      System.out.println(this.nombre+" esta hablando por telefono");
   }
}
```

Ahora, en el código de la clase Empleado, hay que agregar en el método ***main***, unas líneas para crear el objeto, asignarle valores a sus atributos, y mandar llamar a los métodos de hablarPorTelefono() y contactandoAProveedores(). Por lo que el código final queda de la siguiente forma:

```
public class Empleado extends Persona{
   String claveEmpleado;
   double sueldo;
   String puesto;
   public static void main(String args[]){

      /*Se crea un objeto de tipo Empleado*/
      Empleado e = new Empleado();

      /*Se asignan valores a los atributos heredados*/
      e.nombre = "Arturo Martinez";
      e.direccion = "Siempreviva #83";
      e.telefono = "98745632";
      e.edad = 23;

      /*Se asignan valores a los atributos propios de la clase*/
      e.claveEmpleado = "ArMart65874";
      e.sueldo = 4532.80;
      e.puesto = "Capturista";

      /*Desplegamos los valores asignados en pantalla*/
      System.out.println("Nombre: "+e.nombre+"\nDireccion: "
                        +e.direccion+"\nTelefono: "+e.telefono);
      System.out.println("Edad: "+e.edad+"\n\nClave de empleado: "
                        +e.claveEmpleado+"\nPuesto: "+e.puesto);

      /*Se mandan llamar a algunos métodos*/
      e.hablarPorTelefono();
      e.contactandoAProveedores();
   }

   /* se definen los métodos propios del cliente */
   public void atendiendoACliente(){
      System.out.println(this.nombre+" esta atendiendo a un cliente");
   }

   public void contactandoAProveedores(){
      System.out.println(this.nombre+" esta contactando proveedores");
   }
}
```

Si se compila y ejecuta este código se observa una salida como la siguiente:

```
Nombre: Arturo Martinez
Direccion: Siempreviva #83
Telefono: 98745632
Edad: 23

Clave de Empleado: ArMart65874
Puesto: Capturista

Arturo Martinez esta hablando por telefono
Arturo Martinez esta contactando proveedores
```

Cómo compilar y ejecutar un programa de JAVA

Lo primero que hay que hacer es guardar el código fuente con el mismo nombre de la clase, y además con extensión **java**, por ejemplo si la clase es HolaMundo, el nombre del archivo deberá ser HolaMundo.java.

Ahora debemos abrir una ventana del sistema y desplazarnos por las carpetas hasta ubicarnos en donde se encuentren los programas.

Para compilar usamos la instrucción *javac*. Por ejemplo, para compilar el programa HolaMundo.java, debemos escribir la siguiente instrucción:

>javac HolaMundo.java

Si el programa no tiene errores, regresará el símbolo del sistema. Con esto hemos generado el archivo *HolaMundo.class*.

Ahora para ejecutar, usaremos la instrucción *java* de la siguiente forma:

>java HolaMundo

Cabe hacer notar que para ejecutar, no es necesario escribir la extensión **class**.

EJERCICIOS PROPUESTOS

1. Se tiene el siguiente código:

```
public class OperacionesMatematicas{
   int i;
   float f;
   double d,rd;
   public static void main(String args[]){
      OperacionesMatematicas om = new OperacionesMatematicas();
      om.i = 2;
      om.f = 10/3;
      om.d = 2673.8426;

      om.rd = i*f*d;

      System.out.println("El resultado almacenado en rd es: "+rd);
   }
}
```

¿Compila correctamente? ¿Qué errores aparecen?
Corrija dichos errores, sin mover de posición los atributos de la clase, y verifique que se ejecute correctamente.

NOTA: Si se cambian valores en los atributos, específicamente para f*, si pone un número como 25.89, se debe hacer un* cast *explícito, es decir debe anteponer* (float) *al número que se va a utilizar. Quedando de la siguiente forma:*

```
om.f = (float)25.89;
```

2. Investigue, ¿qué es un *cast* entre tipos de dato? Y explique, ¿cuándo se debe hacer explícito?

3. Implemente una clase llamada Punto2D, la cual almacene en sus atributos las coordenadas de un punto en un plano XY, y defina un método para desplegar en pantalla dichas coordenadas.

4. Implemente una clase llamada Punto3D, la cual almacene las coordenadas de un punto en un espacio XYZ. Tenga en cuenta que la clase Punto2D deberá ser la superclase de Punto3D. También deberá contar con un método de impresión de las coordenadas del punto.

5. Implemente una clase llamada Computadora, proponga por lo menos 5 atributos, un atributo extra que indicará el estado de la computadora, si está encendida o apagada. Deberá tener 3 métodos, uno para prender la computadora, otro para apagar la computadora, y otro que indique que se está trabajando en la computadora. Tenga en cuenta que para el método donde se está trabajando en la computadora, no se puede trabajar si está apagada. El estado inicial de la computadora es apagada.

INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS (PARTE II)

Objetivos

El alumno conocerá y aplicará el concepto de programación orientada a objetos para la realización de programas que resuelvan problemas de tipo numérico.

Al final de esta práctica el alumno podrá:

Implementar programas orientados a objetos que resuelvan problemas de tipo numérico.

Antecedentes

25. Tener las bases de la programación orientada a objetos.
26. Manejar sentencias de control de flujo en algún lenguaje de programación

Introducción

Una vez que se conocen los conceptos básicos de la programación orientada a objetos como lo son: objeto, clase, atributo, método y herencia; en esta práctica se aplicarán dichos conceptos para la elaboración de programas que resuelvan problemas de tipo numérico.

Desarrollo

Ejemplo 1

Elaborar un programa que convierta un número complejo de forma rectangular a forma polar.

a) Análisis

El programa requiere como datos de entrada 2 números: el primero es el valor de la parte real y el segundo es el valor de la parte imaginaria del número complejo. A partir de los datos de entrada aplicando las fórmulas correspondientes se obtienen la magnitud y el ángulo del número en su forma polar.

b) Definición de clases

Para resolver este problema, se necesita sólo una clase, ya que únicamente se realizarán dos operaciones sobre los datos de entrada; dicha clase se nombrará *ComplejoAPolar*.

c) Definición de atributos y métodos.

Atributos:

- Dos variables numéricas que contengan la parte real e imaginaria del número complejo. (real, imaginario)
- Dos variables numéricas que contengan la magnitud y ángulo de su forma polar. (r, ang)

Métodos:

- Un método que obtenga la magnitud. (*obtenMagnitud()*)
- Un método que obtenga el ángulo. (*obtenAngulo()*)

d) Diagrama de clases

Por lo tanto el diagrama de clases es:

ComplejoAPolar
double real double imaginario double r double ang
obtenMagnitud() obtenAngulo()

Al iniciar la construcción del programa, se escribe la estructura básica, incluyendo también la creación de un objeto, como sigue:

```
public class ComplejoAPolar{

   double real;
   double imaginario;
   double r;
   double ang;

   public static void main (String[] args){
      ComplejoAPolar cap = new ComplejoAPolar();
   }

   public double obtenMagnitud(){
   }

   public double obtenAngulo(){
   }
}
```

Ahora, se agrega el código para que el programa obtenga los datos de entrada (valores del número complejo) desde la línea de comandos:

```
>java ComplejoAPolar num_real num_imaginario
```

Los valores de num_real y num_imaginario, se asignarán en la variable tipo arreglo ***args***, como argumento de la función ***main***. Para este ejemplo, sólo se tendrán dos elementos almacenados en dicho arreglo:

args[0]→ num_real
args[1]→ num_imaginario

Como los valores leídos se guardan en ***args*** como tipo String, se debe hacer uso de otra clase ya definida por el lenguaje, para hacer su conversión a números; en este caso se usará la clase Double. De esta clase se usará el método parseDouble(), para hacer la conversión de String a double, y poder asignar dichos valores a los atributos real e imaginario; por lo que se deben agregar las siguientes líneas de código:

```
cap.real = Double.parseDouble(args[0]);
cap.imaginario = Double.parseDouble(args[1]);
```

Como se está haciendo uso del método de una clase externa a la clase del ejemplo, se debe utilizar la siguiente sintaxis: Clase.metodoClase()

Ahora se modificarán los métodos de la clase *ComplejoAPolar*, para que reciban parámetros y que realicen la conversión de un número complejo de su forma rectangular a su forma polar. El tipo de argumentos que recibirán los métodos es double; así los encabezados de los métodos tienen la siguiente forma:

```
public double obtenMagnitud(double r, double i){
}
public double obtenAngulo(double r, double i){
}
```

Agregando al código base las modificaciones de los métodos, y la conversión de String a double, el programa queda de la siguiente forma:

```
public class ComplejoAPolar{
   double real;
   double imaginario;
   double r;
   double ang;

   public static void main (String[] args){
      ComplejoAPolar cap = new ComplejoAPolar();
      cap.real = Double.parseDouble( args[0] );
      cap.imaginario = Double.parseDouble( args[1] );
   }

   public double obtenMagnitud(double r, double i){
   }

   public double obtenAngulo(double r, double i){
   }
}
```

Para obtener la magnitud y el ángulo del número complejo, se necesitan los métodos de *sqrt(double número), pow(double base, double exponente)* y *atan(double número)* que realizan las operaciones de raíz cuadrada, potencia y ángulo tangente respectivamente; estos métodos se encuentran en la clase *Math*, que a su vez está dentro del paquete java.lang, por lo que para hacer uso de ellos, se tiene que incluir dicho paquete mediante la siguiente línea:

```
import java.lang.*;
```

Con esto se indica al compilador java, que agregue al programa todas las clases que están contenidas dentro de *java.lang*. Es importante destacar que todos los paquetes que se incluyan, deben estar antes que la definición de la clase.

Ahora se implementan las fórmulas para obtener la magnitud y ángulo dentro de los métodos.

```
public double obtenMagnitud(double r, double i){
   double resultado;
   resultado = Math.sqrt(Math.pow(r,2)+Math.pow(i,2));
   return resultado;
}
public double obtenAngulo(double r, double i){
   double resultado;
   resultado = Math.atan(i/r);
   //Como el resultado de atan está en radianes hacemos la conversión
   resultado = resultado*180/Math.PI;
   return resultado;
}
```

Sólo resta asignar el valor que devuelven estos métodos a los respectivos atributos, por lo que el código completo de este programa queda como se muestra a continuación:

```
import java.lang.*;

public class ComplejoAPolar{
   /*Se definen los atributos de la clase*/
   double real;
   double imaginario;
   double r;
   double ang;

   /*Se define método main*/
   public static void main (String[] args){

      /*Se crea un objeto de la clase*/
      ComplejoAPolar cap = new ComplejoAPolar();

      cap.real = Double.parseDouble( args[0] );
      cap.imaginario = Double.parseDouble( args[1] );

      cap.r = cap.obtenMagnitud(cap.real, cap.imaginario);
      cap.ang = cap.obtenAngulo(cap.real, cap.imaginario);
```

```
      /*Se manda a imprimir en pantalla el resultado*/
      System.out.println("El número complejo "+cap.real+" + "
                         +cap.imaginario+"i en su forma polar ");
      System.out.println("es "+cap.r+" exp("+cap.ang+")");
   }

   /*Se definen los métodos de la clase*/
   public double obtenMagnitud(double r, double i){
      double resultado;
      resultado = Math.sqrt(Math.pow(r,2)+Math.pow(i,2));
      return resultado;
   }

   public double obtenAngulo(double r, double i){
      double resultado;
      resultado = Math.atan(i/r);
      //Como el resultado de atan está en radianes se hace la conversión
      resultado=resultado*180/Math.PI;
      return resultado;
   }
}
```

NOTA: No olvidar ejecutar la clase ComplejoAPolar como se muestra a continuación:

```
> java ComplejoAPolar num1 num2
```

De lo contrario esto generará una excepción, tema que se abordará en la siguiente práctica.

Por ejemplo, si se quiere convertir el valor de 2+i a su forma polar, al ejecutar el programa, genera la siguiente salida:

```
C:\>java ComplejoAPolar 2 1

El número complejo 2.0 + 1.0i en su forma polar
es 2.23606797749979 exp(26.56505117707799)

C:\>
```

Ejemplo 2

Elaborar un programa que reciba un arreglo unidimensional de hasta 10 números, donde el primero indica cuántos números contiene el arreglo y a continuación el arreglo. Deberá obtener tres sumas: suma total, suma de los números que estén en posiciones pares dentro del arreglo, y de los que estén en posiciones impares.

a) Análisis

El programa recibe hasta 11 números (#elementosDelArreglo elemento1 elemento2 …), y obtendrá tres sumas.

b) Definición de clases

Se usará una sola clase que se llamará SumaArreglo.

c) Definición de atributos y métodos

Atributos:

- Una variable que guarde el número de elementos que tendrá el arreglo.
- Un arreglo, en este caso, el límite es de 10 números.
- Tres variables que almacenen las diferentes sumas, aunque es mejor usar un arreglo llamado suma, donde el primer elemento guardará la suma total, el segundo la suma de los números en posición par, y el tercero la suma de los números en posición impar.

Métodos:
Se necesitan tres métodos que obtengan las diferentes sumas:

sumaTotal()
sumaPar()
sumaImpar()

d) Diagrama de clases

SumaArreglo
int n double arr[10] double suma[3]
sumaTotal() sumaPar() sumaImpar()

Entonces, se parte del esquema básico utilizando el diagrama de clases.

```
public class SumaArreglo{
   int n;
   double arr[] = new double[10];
   double suma[] = new double[3];

   public static void main(String args[]){
      SumaArreglo sa = new SumaArreglo();
   }

   public double sumaTotal(){
   }

   public double sumaPar(){
   }

   public double sumaImpar(){
   }
}
```

Ahora, en la definición de los tres métodos, y de acuerdo al análisis previo, se concluye que es necesario utilizar dos parámetros: uno que indique el número de elementos del vector y otro que contenga el propio vector. Por ejemplo, el encabezado del método ***sumaTotal*** es:

```
public double sumaTotal(int x, double[] vector){
}
```

Para el caso del cálculo de la suma de elementos en posición par o impar, se requiere saber la posición del elemento a sumar; esto se logra dividiendo el índice del elemento actual del arreglo sobre 2 para saber a qué suma se adicionará. En la clase ***Math***, del paquete ***java.lang***, está el método ***IEEEremainder(double f1, double f2)*** que devuelve el residuo de una división, con esto se puede decir si el índice del elemento es par o impar.

Cabe aclarar que aunque el arreglo empieza en cero, la numeración de la posición de los elementos inicia desde uno; por lo que a las posiciones pares, les corresponden índices impares.

Finalmente, el programa completo queda de la siguiente manera:

```
import java.lang.*;

public class SumaArreglo{
   /*Se definen los atributos de la clase*/
   int n;
   double arr[] = new double[10];
   double suma[] = new double[3];

   /*Se define la función principal*/
   public static void main(String args[]){

      /*Se crea el objeto*/
      SumaArreglo sa = new SumaArreglo();

      /*Se obtiene el número de elementos del arreglo y se asigna a n*/
      sa.n = Integer.parseInt(args[0]);

      /*Se obtienen los valores de los elementos del arreglo*/
      for(int i=0;i<sa.n;i++){
         sa.arr[i] = Double.parseDouble(args[i+1]);
      }

      /*Se obtienen las sumas, utilizando los métodos de la clase*/
      sa.suma[0] = sa.sumaTotal(sa.n,sa.arr);
      sa.suma[1] = sa.sumaPar(sa.n,sa.arr);
      sa.suma[2] = sa.sumaImpar(sa.n,sa.arr);

      /*Se imprime en pantalla el resultado*/
      System.out.println("La suma total es: "+sa.suma[0]);
      System.out.println("La suma de los pares es: "+sa.suma[1]);
      System.out.println("La suma de los impares es: "+sa.suma[2]);
   }
```

```
   /*Se definen los métodos de la clase*/
   public double sumaTotal(int x, double[] vector){
      double s=0;
      for(int i=0;i<x;i++){
         s=s+vector[i];
      }
      return s;
   }

   public double sumaPar(int x, double[] vector){
      double s=0;
      for(int i=0;i<x;i++){
         if(Math.IEEEremainder(i,2)!=0){
            s=s+vector[i];
         }
      }
      return s;
   }

   public double sumaImpar(int x, double[] vector){
      double s=0;
      for(int i=0;i<x;i++){
         if(Math.IEEEremainder(i,2)==0){
            s=s+vector[i];
         }
      }
      return s;
   }
}
```

Por ejemplo, si deseamos calcular la suma de un arreglo de 5 elementos, al compilar y ejecutar este programa, la salida es la siguiente:

```
C:\>java SumaArreglo 5 3 5 9 10 2
La suma total es: 29.0
La suma de los pares es: 15.0
La suma de los impares es: 14.0

C:\>
```

EJERCICIOS PROPUESTOS

1) Elabore un programa que reciba dos números complejos, y obtenga la multiplicación.

2) Elabore un programa que reciba dos números, y que calcule el error absoluto y relativo. Considere que el primer número es el valor real.

3) Elabore un programa que reciba una matriz de 3 x 3, determine el valor máximo, así como su posición. Si existen valores iguales, obtener todas las posiciones.

4) Elabore un programa que llene automáticamente una matriz identidad de 20 x 20, y la despliegue en pantalla.

5) Elabore un programa que reciba una matriz de 4 x 4, y que determine cuántos números son positivos, cuántos ceros, y cuántos negativos.

6) Si se tiene la siguiente ecuación $f(x) = x^2 + 9x + 4$, elabore un programa que implemente el método de bisección, y calcule el error relativo en cada iteración. El programa deberá recibir el intervalo en el cual se encontrará x1= - 0.468871 o x2= - 8.5311288

PROGRAMACIÓN ORIENTADA A OBJETOS (MANEJO DE EXCEPCIONES)

Objetivos

El alumno empleará el concepto de excepción en la programación orientada a objetos para el manejo de errores de ejecución.

Al final de esta práctica el alumno podrá:

Implementar programas orientados a objetos que manejen errores de ejecución.

Antecedentes

27. Haber aplicado los conceptos básicos de la programación orientada a objetos en programas sencillos.
28. Manejar sentencias de control de flujo en lenguaje Java.

Introducción

Una excepción se genera cuando ocurre un error en tiempo de ejecución. En Java es común manejar las excepciones para controlar la ejecución del programa. Es por ello la importancia de conocer cómo se pueden detectar y manipular las diferentes excepciones definidas en el lenguaje.

Por ejemplo, si para la correcta ejecución de un programa requiere que los datos se le den desde la línea de comandos, se puede utilizar la excepción llamada **IOException**, la cual se generará si no se dan los datos de la forma establecida.

Una vez que se conoce el tipo de excepción que se puede generar en cierta parte del código de un programa, es necesario emplear la sentencia *try-catch* para indicar las acciones que debe realizar el programa en caso de que ocurra la excepción. La sintaxis de la sentencia *try-catch*, se presenta a continuación:

```
try {
        Código donde puede ocurrir el error
    }
catch (nombre_de_excepción, variable_de_excepción){
        Código que se ejecuta en caso de que ocurra el error
    }
```

Ejemplo 1
Excepción **ArrayIndexOutOfBoundsException**

Cuando se ejecuta una clase definida con parámetros y éstos no se dan desde la línea de comandos, se genera una excepción debido a que se usan localidades de la variable *args*; al no tener elementos que leer, se genera un **ArrayIndexOutOfBoundsException**, excepción

que puede ser manejada usando *try-catch*. El siguiente programa incluye el código de manejo de dicha excepción:

```
import java.lang.*;

public class ComplejoAPolar{
   public static void main (String[] args){
      try{
         ComplejoAPolar cap = new ComplejoAPolar();
          double real = Double.parseDouble( args[0] );
          double imaginario =  Double.parseDouble( args[1] );
          double r;
          double ang;
          r = cap.obtenMagnitud(real, imaginario);
          ang = cap.obtenAngulo(real, imaginario);
         System.out.println("El numero complejo "+real+" +"
                            +imaginario+" i en su forma polar");
         System.out.println("es "+r+" exp("+ang+")");
      }
      catch(ArrayIndexOutOfBoundsException aioobe){
         System.out.println("ERROR!!");
         System.out.println("Sintaxis: java ComplejoAPolar”
                            +” real imaginario");
      }
   }
   public double obtenMagnitud(double r, double i){
      double resultado;
       resultado = Math.sqrt(Math.pow(r,2)+Math.pow(i,2));
       return resultado;
   }
   public double obtenAngulo(double r, double i){
       double resultado;
       resultado = Math.atan(i/r);
       //El resultado de atan es en radianes, se convierte a grados
       resultado=resultado*180/Math.PI;
       return resultado;
   }
}
```

La salida del programa, en caso de no incluir los datos de entrada en la línea de comandos al momento de ejecutar la clase, es la siguiente:

```
>java ComplejoAPolar
ERROR!!
Sintaxis: java ComplejoAPolar real imaginario
```

Ejemplo 2
Lectura de una cadena desde la línea de comandos, y excepción **IOException**.

Este programa leerá una cadena desde la línea de comandos al momento de mandar la orden de ejecución. Para ello se usarán algunas clases predefinidas en java que se encuentran en *java.io*; estas clases son:

- InputStreamReader: La cual va leyendo los *bytes* de la cadena de entrada y los transforma a caracteres. Para leer una cadena completa desde el teclado, se debe generar un objeto de esta clase, cuyo argumento es el medio de entrada de los datos; en este caso es **System.in**. Por lo tanto la creación de dicho objeto se muestra a continuación:

```
InputStreamReader isr = new InputStreamReader(System.in);
```

- BufferedReader: Esta clase permite una lectura eficiente, y sirve de puente entre el origen de los datos y donde se guardarán. Para ello se debe definir un objeto de esta clase usando el *InputStreamReader* creado previamente. Además se utilizará el método *readLine()* definido en esta clase para leer una línea completa.

```
BufferedReader br = new BufferedReader(isr);
```

Este programa podría generar una excepción **IOException** en caso de no incluir una cadena en el momento de ejecutarlo, por lo que habrá que utilizar *try-catch*.

```
import java.io.*;

public class LeerUnaCadena{
   public static void  main(String args[]){
      try{
         // Definir un flujo de caracteres de entrada
         InputStreamReader isr = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(isr);
         // Variable para almacenar una línea de texto
         String sdato;
         System.out.print("Introduzca un texto: ");
         sdato = br.readLine(); // leer una línea de texto
         // Escribir la línea leída
         System.out.println("La cadena leida es: "+sdato);
      }
      catch(IOException ioe){
         //La ignorará
      }
   }
}
```

Otra forma de manejar las excepciones, es utilizando la cláusula *throws*, que identifica la lista de posibles excepciones que un método puede generar. Esta cláusula se aplica a todo un método, a diferencia de la sentencia *try-catch* que acota el código susceptible a generar

una excepción. Como el método que utiliza *throws* no tiene código para manejar las excepciones que pueda generar, los métodos que lo invoquen deberán tener protecciones frente a esas excepciones. La declaración del método con uso de *throws* presenta la siguiente sintaxis.

tipo nombreMetodo(argumentos) throws lista_excepciones { código_del_método}

Ejemplo 3
Programa que solicita una cadena; emplea un método que utiliza la cláusula *throws* para leer la cadena.

```
import java.io.*;

public class LeerUnaCadena{
   public static void  main(String args[]) {
     LeerUnaCadena objCad = new LeerUnaCadena();
     String cadena;
     try {
        // Se invoca al método que puede generar un IOException
        cadena = objCad.leeCadena();
        System.out.println("\n La cadena leida es: "+ cadena);
     } catch (IOException e) {
           System.out.println("Error de entrada salida");
     }
   }

   // Método que da lectura de la cadena; usa throws
   public static String leeCadena() throws IOException{
      // Definición de un flujo de caracteres de entrada
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      // Variable para almacenar una línea de texto
      String sdato;
      System.out.print("Introduzca un texto: ");
      sdato = br.readLine(); // leer una línea de texto
      return sdato;
   }
}
```

Se puede observar que el método leeCadena emplea la cláusula *throws*; esto es obligatorio porque la línea

```
sdato = br.readLine(); // leer una línea de texto
```

puede generar un IOException.

También es necesario que el método main maneje la excepción IOException. Esto lo hace con la sentencia *try-catch* mostrada en negritas.

A manera de ejercicio de prueba, realiza las siguientes actividades:

a) Transcribe el ejemplo 3, compílalo y ejecútalo. Notarás que no hay errores ni de compilación ni de ejecución.
b) Ahora elimina la cláusula *throws* del método leeCadena. La línea debe quedar de la siguiente forma:

```
public static String leeCadena(){
```

Compila el programa. Deberá presentar un mensaje de error y no generará el archivo .class. Explica por qué sucede.

c) Vuelve a incluir la cláusula *throws* al método leeCadena y comenta las siguientes líneas del método main

```
try {

} catch (IOException e) {
      System.out.println("Error de entrada salida");
}
```

Con esto se está excluyendo el manejo de la excepción IOException. Compila el programa. Notarás que también marca error de compilación. Explica porqué.

El programa del ejemplo 3 también se puede reducir empleando exclusivamente el método main con la cláusula *throws*. Finalmente el programa queda como se presenta a continuación:

```
import java.io.*;

public class LeerUnaCadena{
   public static void  main(String args[]) throws IOException{
      // Definir un flujo de caracteres de entrada: flujoE
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      // Variable para almacenar una línea de texto
      String sdato;
      System.out.print("Introduzca un texto: ");
      sdato = br.readLine(); // leer una línea de texto
      // Escribir la línea leída
      System.out.println("La cadena leida es: "+sdato);
   }
}
```

Ejercicios Propuestos

1. El siguiente código:

```
public class Division{
   public static void main(String args[]){
      int x,y,z;
      x=2;
      y=0;
      z=x/y;
      System.out.println("El resultado es: "+z);
   }
}
```

genera una excepción al ejecutarse.¿Qué excepción se genera? ¿Por qué?
Modifique el código utilizando try-catch, para que despliegue en pantalla la razón de la excepción. Además, haga que el programa reciba parámetros desde la línea de comandos.

2. Investigue con qué tipo de datos se genera la excepción del ejercicio anterior. Utilizando el código original del ejercicio anterior, si se cambia el tipo de dato, por ejemplo a double, ¿qué sucede?

3. El siguiente código:

```
public class Arreglo{
   public static void main(String args[]){
      int n = Integer.parseInt(args[0]);
      int arr[] = new int[n];
      for(int i=0;i<n;i++){
         arr[i]=i+1;
      }
      for(int i=0;i<n;i++){
         System.out.println("arr["+i+"]: "+arr[i]);
      }
   }
}
```

genera una excepción al ejecutarse.¿Qué excepción se genera?, ¿por qué?
Modifique el código utilizando try-catch, para que despliegue en pantalla la razón de la excepción.

PROGRAMACIÓN ORIENTADA A OBJETOS (APLICACIÓN DE MÉTODOS NUMÉRICOS)

Objetivos

El alumno aplicará el concepto de programación orientada a objetos para la realización de programas que resuelvan problemas de tipo numérico.

Al final de esta práctica el alumno podrá:

Implementar programas orientados a objetos que resuelvan problemas de métodos numéricos.

Antecedentes

29. Manejar sentencias de control de flujo en lenguaje Java.
30. Identificar y relacionar varias clases en la programación orientada a objetos empleando la teoría de diseño de jerarquía de clases.
31. Haber aplicado métodos numéricos en la solución numérica de ecuaciones y sistemas de ecuaciones lineales, así como interpolación y aproximación numérica.

Introducción

En esta práctica se trabajará con problemas de tipo numérico que serán resueltos aplicando métodos numéricos. Por la complejidad de estos problemas, se requiere seguir una metodología que permita, de una forma clara y sencilla, implementar los métodos numéricos en programas orientados a objetos. Una de estas metodologías es la llamada *Diseño de jerarquía de clases*.

El diseño de jerarquía de clases es el primer y más importante paso en el diseño de un programa orientado a objetos, ya que el desarrollo de una representación correcta de los datos que están involucrados en el problema, lleva a menudo a resolver satisfactoriamente gran parte de dicho problema.

Para descubrir una apropiada representación de datos, es conveniente realizar los siguientes cuatro pasos:

1. *Análisis del problema*. El enunciado del problema, por lo regular, es todo lo que se tiene, por lo que debemos analizarlo detalladamente y determinar con qué tipo de información se va trabajar en el programa. Es importante identificar clases de información y no piezas específicas de información. En este paso se obtiene una lista de nombres y clases relevantes de información y una breve descripción de cada clase.
2. *Ejemplos de información*. Generalmente, la gente que resuelve problemas requiere de ejemplos. Esto es necesario para completar la descripción de clases. Este paso es muy importante cuando los enunciados de los problemas son complejos e involucran distintas formas de información.

3. *Diagrama de clases*. En este paso, se convierte la descripción informal de las clases a un diagrama con notación rigurosa. El diagrama está compuesto por cajas y dos tipos de flechas que las conectan y que definen la jerarquía de clases
4. *Representación e interpretación* Después de haber obtenido el diagrama y la jerarquía de clases, se convierten las clases informales del paso dos a objetos del lenguaje a emplear, en este caso de Java. Este paso asegura que entendemos los resultados que se deben obtener del cálculo efectuado en el programa.

Ejemplo 1

Enunciado del problema:
Elaborar un programa que lea un arreglo de 5 elementos y obtenga la suma de dichos elementos.

1. *Análisis del problema:*

Analizando el enunciado del problema se puede observar que se requiere definir un arreglo de 5 elementos de tipo numérico, los cuales los sumará para generar un número con el resultado de la suma. De aquí que se pueden identificar las siguientes clases:

- LeeUnDouble, que leerá cada elemento del arreglo, y
- LeeYSuma, que realizará la suma de los elementos del arreglo y entregará el resultado

2. *Ejemplos de información:*

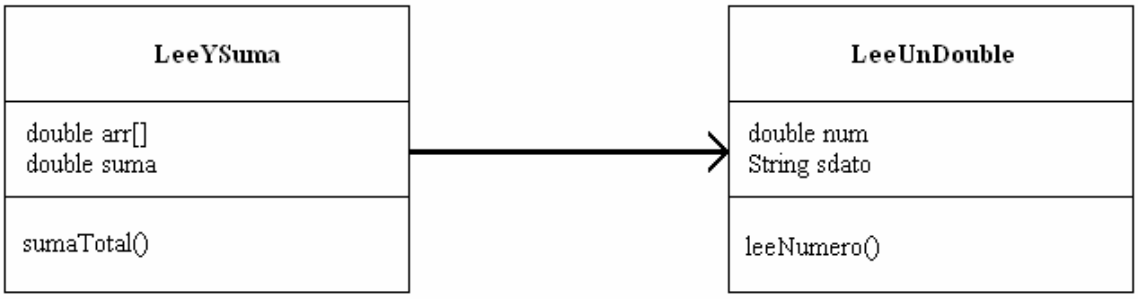
Un arreglo de cinco elementos puede ser arr=[2.3, 4.7, 1.0, 3.1, 7.3]; cuya suma es 18.4
Con base en este ejemplo de información, podemos indicar qué atributos y métodos tendrá cada clase:

a) LeeYSuma

Usará un arreglo de tipo double nombrado ***arr*** con 5 elementos y una variable que guarde la suma llamada ***suma***.
Usará un método para hacer la suma, el cual recibirá el arreglo y devolverá como resultado la suma.

b) LeeUnDouble
Para esta clase se usará un atributo llamado ***num***, que almacenará el número leído; y sólo un método que leerá el número desde la ventana.

3. Diagrama de clases



La flecha que va de la clase LeeYSuma hacia LeeUnDouble, es para indicar dependencia, es decir la clase LeeYSuma depende de LeeUnDouble.

4. Representación e interpretación

LeeUnDouble

Esta clase será una adaptación de la clase LeerUnaCadena definida en la práctica anterior (Programación Orientada a Objetos. Manejo de excepciones) ya que en lugar de leer una cadena leerá un número de tipo double, y lo almacenará en una variable global llamada *num*.

```
import java.io.*;

public class LeeUnDouble{
   double num;
   public void leeNumero(){
      try{
         InputStreamReader isr = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(isr);

         String sdato;
         sdato = br.readLine();
         num = Double.parseDouble(sdato);
      }
      catch(IOException ioe){
      }
   }
}
```

Esta clase no lleva método *main*, ya que no se invocará directamente para su ejecución; en vez de ello, el método leeNumero será usado por la clase LeeYSuma. La clase LeeUnDouble deberá compilarse para que LeeYSuma pueda usar dicho método.

LeeYSuma

```
import java.io.*;

public class LeeYSuma{
   // método main que invoca al método leeNumero, para leer
   // los elementos del arreglo; después invoca al método
   // sumaTotal para obtener la suma de los elementos del arreglo
   public static void main(String args[]){
      LeeUnDouble lud = new LeeUnDouble();
      double suma;
      double arr[] = new double[5];
      for(int i=0;i<5;i++){
         System.out.println("Dame el numero "+i+":");
         lud.leeNumero();//Se obtiene el double
         arr[i]=lud.num;//Se guarda el valor en el arreglo
      }
      suma=sumaTotal(arr);
      System.out.println("La suma del arreglo es: "+suma);
   }

   //método que efectúa la suma de los elementos del arreglo
   public static double sumaTotal(double vector[]){
      double s=0;
      for (int i=0;i<5;i++){
         s=s+vector[i];
      }
      return s;
   }
}
```

En esta clase, para leer el valor de cada elemento del arreglo es necesario crear un objeto de tipo LeeUnDouble, y hacer uso de su método y atributo.

Como cada clase se encuentra en un archivo independiente se deben compilar por separado. Hay que recordar que el nombre del archivo debe ser exactamente el nombre de la clase con la extensión *.java*

Para la ejecución del programa y que dé como resultado la suma de los elementos del arreglo se ejecuta sólo la clase LeeYSuma.class, ya que ésta invocará a la clase LeeUnDouble que debe estar en el mismo directorio desde donde se envía la orden de ejecución.

Ejemplo 2

Enunciado del problema:
Implementar el algoritmo de bisección para la solución numérica de ecuaciones.

1. *Análisis del problema:*

El enunciado es muy concreto, se refiere a un algoritmo de solución numérica de ecuaciones: el de bisección; por lo que primeramente es necesario revisar el método numérico correspondiente y entender muy bien su algoritmo. Para ello hay que remitirse a la bibliografía adecuada. De dicha revisión, el siguiente seudocódigo presenta una solución al problema:

Variables:
`a` y `b` son los límites del rango conde se encuentra una raíz
`c` es el valor medio del rango
`tolerancia`: es el valor de la tolerancia de la raíz

```
LEE  a,b y tolerancia
WHILE  | a-b | > tolerancia
     c=(b+a)/2;
     SI  (funcion(a) *funcion(b) < 0)
         b=c
     SI NO
         a=c
 ESCRIBE ("La raíz aproximada es :", c)
```

2. *Ejemplos de información:*

Una ecuación de ejemplo es $f(x) = x^4 + 5.8x^3 - 22.4x^2 - 31.2x + 57.6$, en el intervalo [1,2] hay una raíz cuyo valor es 1.2

Con base en este ejemplo de información, podemos indicar qué atributos y métodos tendrá cada clase:

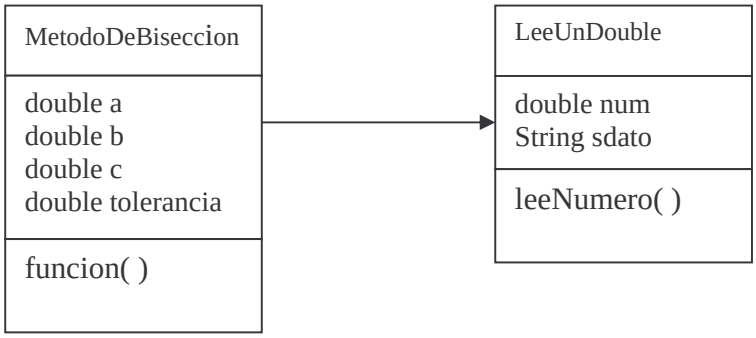
a) MetodoDeBiseccion

Requiere los límites del intervalo: a,b y la tolerancia. También una variable c para ir acotando el intervalo.
Usará un método para hacer la evaluación de la función para cada límite del intervalo; recibirá un límite del intervalo y devolverá como resultado la evaluación de la función.

b) LeeUnDouble

Para esta clase se usará un atributo llamado *num*, que almacenará el número leído; y sólo un método que leerá el número desde la ventana.

3. *Diagrama de clases:*



4. *Representación e interpretación*

La clase MetodoDeBiseccion usará el método leeNumero de la clase LeeUnDouble (la cual ya se creó en el ejemplo anterior), para leer los valores de `a`, `b` y `tolerancia`. También invocará a su método funcion() para evaluar la función en los límites del intervalo. El código de la clase MetodoDeBiseccion es la siguiente

```
import java.io.*;
import java.lang.*;

public class MetodoDeBiseccion{
   // metodo main que invoca al metodo leeNumero, para leer
   // el rango [a,b] y la tolerancia;
   public static void main(String args[]){
        LeeUnDouble lud = new LeeUnDouble();
        double a,b,c=1.0,tolerancia;
        int i=0;

         System.out.println("Dame el límite a: ");
         lud.leeNumero();//Se obtiene el double
         a=lud.num;//Se guarda el valor en a
         System.out.println("Dame el límite b: ");
         lud.leeNumero();//Se obtiene el double
         b=lud.num;//Se guarda el valor en b
         System.out.println("Dame la tolerancia: ");
         lud.leeNumero();//Se obtiene el double
         tolerancia=lud.num;//Se guarda la tolerancia

        // algoritmo de bisección
        while (Math.abs(a-b)> tolerancia) {
                  c=(b+a)/2;
             if (funcion(a)*funcion(c)< 0)
                 b=c;
```

```
            else
                a=c;
            i++;
          }
         System.out.println("La raíz aproximada es:"+c
                        +" en la iteración "+i);
   }

   // método que efectúa la evaluación de la función
   public static double funcion(double x){
      double res;
      res=Math.pow(x,4)+(5.8*Math.pow(x,3))-
          (22.4*Math.pow(x,2))-(31.2*x)+57.6;
      return res;
   }
}
```

Al igual que en el ejemplo 1, como cada clase se encuentra en un archivo independiente se deben compilar por separado; y en este ejemplo sólo ejecutar la clase MetodoDeBiseccion, teniendo en el mismo directorio la clase LeeUnDouble ya compilada.

Ejercicios Propuestos

4. Adapte el programa del ejemplo 2, correspondiente al método de bisección, para que resuelva $f(x) = x^3 - 4.0183x^2 + 2.29575x + 2.27745$, en el intervalo [0,2], y una tolerancia de 0.00001. Agregue al programa, las líneas necesarias para que calcule el error relativo, sabiendo que la raíz es $x = 1.5183$

5. Implemente el algoritmo de Newton-Raphson para la solución numérica de ecuaciones. *NOTA: El programa sólo deberá contener un método que resuelva el algoritmo. Este método recibirá los parámetros necesarios para su correcto funcionamiento.*

6. Utilizando la clase anterior, genere otra clase para resolver la ecuación del ejercicio 2; utilice la misma tolerancia.

7. Implemente el algoritmo de Eliminación Gaussiana para sistemas de ecuaciones lineales. *NOTA: Debe ser general, y al ejecutarlo, se le debe indicar al programa de cuántas variables es el sistema. Recuerde que debe considerar que en el caso de que el renglón pivote tenga cero, intercambie el renglón.*

8. Utilice el programa anterior para resolver el siguiente sistema de ecuaciones.

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 2.4 \\ -2x_1 - 4x_2 + 8x_3 &= 6.4 \\ 3x_1 + x_2 + 2x_3 &= 7.6 \end{aligned}$$

9. Implemente el algoritmo de Gauss-Jordan para sistemas de ecuaciones lineales. *NOTA: Debe ser general, y al ejecutarlo, se le debe indicar al programa de cuántas variables es el sistema. . Recuerde que debe considerar que en el caso de que el renglón pivote tenga cero, intercambie el renglón.*

10. Utilice el programa anterior para resolver el siguiente sistema de ecuaciones:

$$\begin{aligned} 4x_1 + 2x_2 + x_3 + 2x_4 + 6x_5 &= 22.26 \\ 5x_1 + x_2 + x_3 - 3x_4 + 8x_5 &= 9.95 \\ 5x_1 + 7x_2 + 2x_3 + 5x_4 + 2x_5 &= 54.81 \\ x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 &= 7.49 \\ 3x_1 + 2x_2 + 6x_3 - 8x_4 - 11x_5 &= 76.81 \end{aligned}$$

11. Implemente el algoritmo de LU para sistemas de ecuaciones lineales.

12. Utilice el programa anterior para resolver el siguiente sistema de ecuaciones.

$$
\begin{aligned}
4x_1 + 2x_2 + x_3 + 2x_4 + 6x_5 &= 22.26\\
5x_1 + x_2 + x_3 - 3x_4 + 8x_5 &= 9.95\\
5x_1 + 7x_2 + 2x_3 + 5x_4 + 2x_5 &= 54.81\\
x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 &= 7.49\\
3x_1 + 2x_2 + 6x_3 - 8x_4 - 11x_5 &= 76.81
\end{aligned}
$$

13. Utilizando el método de Gauss-Seidel, resuelva el siguiente sistema de ecuaciones, utilice una tolerancia de 0.00001.

$$
\begin{aligned}
5x_1 + 3x_2 - 8x_3 &= 85.3\\
-x_1 + 4x_2 - 6x_3 &= 14.32\\
4x_1 - 6x_2 + x_3 &= 17.61
\end{aligned}
$$

14. Implemente el algoritmo de interpolación de Diferencias Divididas de Newton.

15. Utilice el programa anterior para encontrar un polinomio para los siguientes puntos.

x	f(x)
0.1	5.77
1	6
1.5	6.61
2	7.39

Y obtenga el valor de la función en x = 1.1.

16. Implemente el algoritmo de interpolación de Lagrange.

17. Utilice el programa anterior para encontrar un polinomio para los siguientes puntos.

x	f(x)
0	2
1	3
4	18
6	38

Aproximar en x = 2.

DERIVACIÓN E INTEGRACIÓN NUMÉRICA

Objetivos

El alumno conocerá y aplicará diversas técnicas de derivación e integración numérica.

Al final de esta práctica el alumno podrá:

1. *Resolver ejercicios que contengan derivadas e integrales, por medio de métodos numéricos, tales como método de Taylor y Simpson respectivamente.*
2. *Implementar dichos métodos numéricos en lenguaje orientado a objetos*

Antecedentes

1. Haber elaborado programas orientados a objetos en lenguaje Java con aplicación numérica.
2. Manejar soluciones numéricas de derivadas e integrales

Introducción

Derivación numérica

Para diferenciar numéricamente funciones que están definidas mediante datos tabulados o mediante curvas determinadas en forma experimental se usan diferentes procedimientos.

Un método consiste en aproximar la función en la vecindad del punto en que se desea la derivada, mediante una parábola de segundo, tercer o mayor grado, y utilizar entonces la derivada de la parábola en ese punto como la derivada aproximada de la función; este método podría ser el de la Serie de Taylor.

La serie de Taylor para una función $y = f(x)$ en $x_i + \Delta x$, desarrollada con respecto al punto x_i es

$$y(x_i + \Delta x) = y_i + y'_i(\Delta x) + \frac{y''_i(\Delta x)^2}{2!} + \frac{y'''_i(\Delta x)^3}{3!} + \ldots \qquad (1)$$

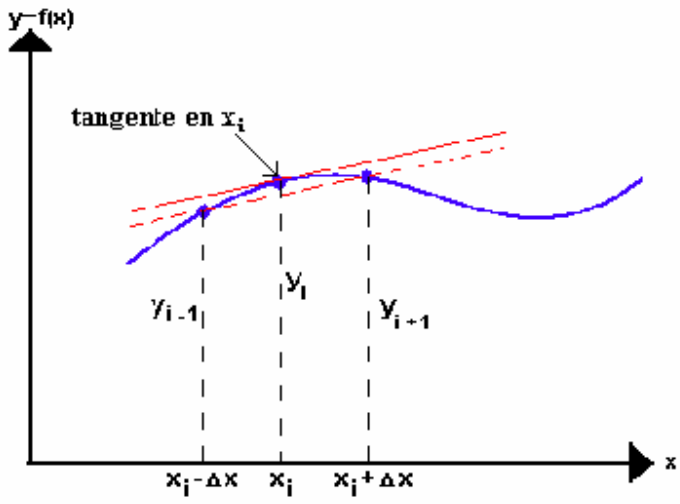
en donde y_i es la ordenada que corresponde a x_i y $(x_i + \Delta x)$ se encuentra en la región de convergencia. La función para $(x_i - \Delta x)$ está dada en forma similar por:

$$y(x_i - \Delta x) = y_i - y'_i(\Delta x) + \frac{y''_i(\Delta x)^2}{2!} - \frac{y'''_i(\Delta x)^3}{3!} + \ldots \qquad (2)$$

Utilizando solamente los tres primeros términos de cada desarrollo, se obtiene una expresión para y_i restando la ec. (2) de la ec. (1),

$$y'_i = \frac{y(x_i + \Delta x) - y(x_i - \Delta x)}{2\Delta x} \qquad (3)$$

Diferencias Centrales, Hacia Adelante y Hacia Atrás



Si se denotan los puntos uniformemente espaciados a la derecha de x_i como x_{i+1}, x_{i+2},… y los puntos a la izquierda de x_i como x_{i-1}, x_{i-2},...; y si se identifican las ordenadas correspondientes como y_{i+1}, y_{i+2}, y_{i-1}, y_{i-2}, respectivamente, la ec. (3) se puede escribir:

$$y'_i = \frac{y_{i+1} - y_{i-1}}{2\Delta x} \qquad (4)$$

La ec. (4) se denomina la primera aproximación, por *Diferencias Centrales* de *y'*, para *x*. La aproximación representa gráficamente la pendiente de la recta discontinua mostrada en la figura de arriba. La derivada real se representa mediante la línea sólida dibujada como tangente a la curva en x_i.

Si sumamos las ecuaciones (1) y (2), y utilizamos la notación descrita previamente, se puede escribir la siguiente expresión para la segunda derivada:

$$y''_i = \frac{y_{i+1} - 2y_i + y_{i-1}}{(\Delta x)^2} \qquad (5)$$

La ec. (5) es la primera aproximación, por *Diferencias Centrales*, de la segunda derivada de la función en x_i. Esta expresión se puede interpretar gráficamente como la pendiente de la tangente a la curva en $x_{i+1/2}$ menos la pendiente de la tangente a la curva en $x_{i-1/2}$ dividida entre Δx, cuando las pendientes de las tangentes están aproximadas mediante las expresiones:

$$y'_{i+\frac{1}{2}} = \frac{y_{i+1} - y_i}{\Delta x} \qquad (6)$$
$$y'_{i-\frac{1}{2}} = \frac{y_i - y_{i-1}}{\Delta x}$$

es decir,

$$y'' = \frac{\frac{y_{i+1} - y_i}{\Delta x} - \frac{y_i - y_{i-1}}{\Delta x}}{\Delta x} = \frac{y_{i+1} - 2y_i + y_{i-1}}{(\Delta x)^2} \qquad (7)$$

Se ha demostrado que las expresiones de *Diferencias Centrales* para las diversas derivadas encierran valores de la función en ambos lados del valor x en que se desea conocer la derivada en cuestión. Se pueden obtener fácilmente expresiones para las derivadas, totalmente en términos de valores de la función en x_i y puntos a la derecha de x_i. Éstas se conocen como expresiones de *Diferencias Finitas Hacia Adelante*. En forma similar, se pueden obtener expresiones para las derivadas que estén solamente en términos de valores de la función en x_i y puntos a la izquierda de x_i. Éstas se conocen como expresiones de *Diferencias Finitas Hacia Atrás*.

EJEMPLO

Usar aproximaciones de *Diferencias Finitas Hacia Adelante, Hacia Atrás* y *Centradas* para estimar la primera derivada de:

$$f(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.25x + 1.2 \text{ en } x = 0.5$$

Utilizando un Δx de 0.5.
Repetir los cálculos usando $\Delta x = 0.25$.

Nótese que la derivada se puede calcular directamente como:

$f'(x) = -0.4x^3 - 0.45x^2 - 1.0x - 0.25$

y evaluando tenemos: $f'(0.5) = -0.9125$

SOLUCIÓN:

Para $\Delta x = 0.5$ se usa la función para determinar:

$X_{i-1} = 0.0$	$Y_{i-1} = 1.200$
$X_i = 0.5$	$Y_i = 0.925$
$X_{i+1} = 1.0$	$Y_{i+1} = 0.200$

Estos datos se utilizan para calcular:

la *Diferencia Hacia Adelante*:

$$y'(0.5) \cong \frac{0.2 - 0.925}{0.5} = -1.45$$

la *Diferencia Dividida Hacia Atrás*:

$$y'(0.5) \cong \frac{0.925 - 1.2}{0.5} = -0.55$$

y la *Diferencia Dividida Central*:

$$y'(0.5) \cong \frac{0.2 - 1.2}{1.0} = -1.0$$

Para $\Delta x = 0.25$, los datos son:

$X_{i-1} = 0.25$	$Y_{i-1} = 1.10351563$
$X_i = 0.50$	$Y_i = 0.92500000$
$X_{i+1} = 0.75$	$Y_{i+1} = 0.63632813$

Por lo que la *Diferencia Dividida Hacia Adelante*:

$$y'(0.5) \cong \frac{0.63632813 - 0.925}{0.25} = -1.15468748$$

la *Diferencia Dividida Hacia Atrás*:

$$y'(0.5) \cong \frac{0.925 - 1.10351563}{0.25} = -0.71406252$$

y la *Diferencia Dividida Central*:

$$y'(0.5) \cong \frac{0.63632813 - 1.10351563}{0.5} = -0.934375$$

Para los dos Δx, las aproximaciones por *Diferencias Centrales* son más exactas que las *Diferencias Divididas Hacia Adelante* o las *Diferencias Divididas Hacia Atrás*.

Codificando este método en Java, se tiene:

```
public class Derivacion{
   double x;
   double deltaX;
   double haciaAdelante;
   double haciaAtras;
   double centradas;

   public static void main(String args[]){
   try{

      Redondear r = new Redondear();
      Derivacion d = new Derivacion();
      d.x = Double.parseDouble(args[0]);
      d.deltaX = Double.parseDouble(args[1]);
      d.haciaAdelante = (d.funcion(r.redondeo(d.x+d.deltaX,2))-
      d.funcion(d.x))/(d.deltaX);

      System.out.println("La derivada de "+d.x+" hacia adelante es:
      "+d.haciaAdelante);

      d.haciaAtras = (d.funcion(d.x)-d.funcion(r.redondeo(d.x-
      d.deltaX,2)))/d.deltaX;

      System.out.println("La derivada de "+d.x+" hacia atras es
      :"+d.haciaAtras);

      d.centradas = (d.funcion(r.redondeo(d.x+d.deltaX,2))-
      d.funcion(r.redondeo(d.x-d.deltaX,2)))/(2*d.deltaX);

      System.out.println("La derivada de "+d.x+" en diferencias centradas
      es: "+d.centradas);

       }

   catch(ArrayIndexOutOfBoundsException aioobe){

       System.out.println("ERROR!!! Faltan parametros");

       System.out.println("Sintaxis: java Derivacion valor_inicial
       incremento");

       }
   }

   public double funcion(double x){

   double f;

   f = -0.1*Math.pow(x,4)-0.15*Math.pow(x,3)-0.5*Math.pow(x,2)-0.25*x+1.2;

   return f;

    }

}
```

Como se puede observar, este código implementa las ecuaciones para las diferencias hacia adelante, hacia atrás, y centradas. Recibe como parámetros el punto en el que se desea calcular la derivada, y el Δx deseado. Se tiene un método que implementa la función de la cual se desea obtener la derivada.

Una de las características principales de este código es que se utiliza un objeto de la clase **Redondear**, la cual es utilizada debido a que en Java, como en otros lenguajes, tiene errores de redondeo y aritmética de computadora. Por ejemplo, se puede dar el caso que si se resta 15 a 15.1, el resultado es 0.0999999999999999999; de aquí el uso de dicha clase ayuda a hacer redondeos para tener valores útiles en estos cálculos.

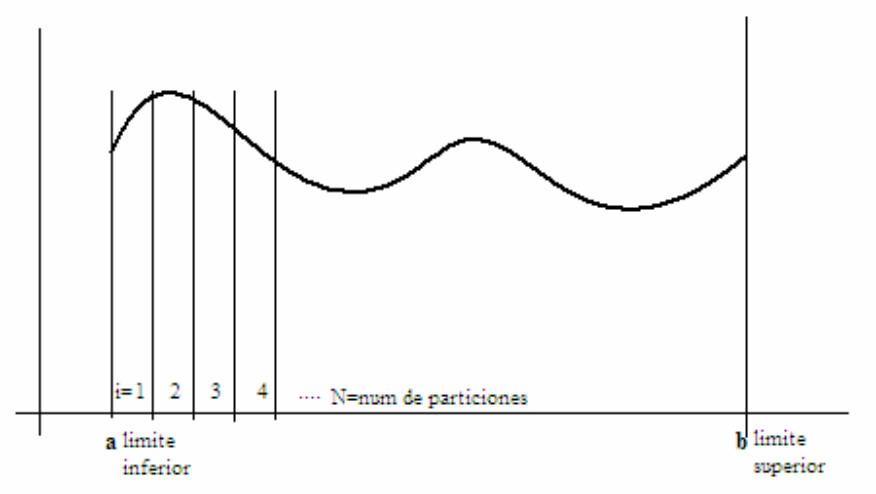
El código de Redondear es:

```
import java.math.*;
public class Redondear{
  public double redondeo(double resultado, int redondeo){
      BigDecimal bd=new BigDecimal(
      Double.toString(resultado)).setScale(redondeo,BigDecimal.ROUND_HALF
      _UP) ;
      return bd.doubleValue();
   }
}
```

Este código crea un objeto de la clase BigDecimal, que ayuda a hacer el redondeo. Se utiliza el método *setScale()*, que recibe como parámetros el número de decimales a los que se redondeará, y un factor, que en este caso, para evitar equivocaciones, se usa ROUND_HALF_UP.

Integración numérica

El principio de los métodos de integración numérica consiste en ajustar un polinomio a un conjunto de puntos y luego integrarlo. Al realizar dichas integrales obtenemos, entre otras, las reglas de trapecio y de Simpson las cuales dan lugar a reglas de integración compuestas que buscan que el error sea cada vez menor.



Regla de Simpson de $\frac{1}{3}$

La regla de Simpson de $\frac{1}{3}$ o simplemente regla de Simpson, consiste en aproximar la curva con polinomios de grado 2, es decir, con parábolas. Omitiendo la deducción, el resultado es

$$S_N = \frac{H}{3}\left(y_0 + 4\sum_{i=1,3,5,\ldots}^{N-1} y_i + 2\sum_{i=2,4,6,..}^{N-2} y_i + y_N \right)$$

La primera sumatoria es para aquellas *i's* que sean impares. La segunda es para las *i's* que sean pares. Dado que para obtener la ecuación de una parábola se requieren 3 puntos, se necesitan 2 particiones, por lo cual la N debe ser par.

Regla de Simpson de $\frac{3}{8}$

La regla de Simpson de $\frac{3}{8}$ consiste en aproximar la función mediante una cúbica:

$$S_{\frac{3}{8}N} = \frac{3H}{3}\left(y_0 + 2\sum_{i=3,6,9,..}^{N-1} y_i + 3\sum_{i\neq \text{múltiplos de } 3}^{N-2} y_i + y_N \right)$$

Esta regla es más complicada. La primera sumatoria sólo incluye aquellas *i's* que sean múltiplos de 3. La segunda el resto, es decir, las *i's* que no sean múltiplos de 3. Entre las 2 cubren desde 1 hasta N-1. Para una cúbica se requieren 4 puntos, por lo cual se utilizan 3 intervalos. Por esta razón N debe ser un múltiplo de 3.

Ejemplo de regla de Simpson

Calcularla siguiente integral con las reglas anteriores. $\int_0^1 \frac{4dx}{1+x^2} = \pi$

Para N=2

$$S_2 = \frac{H}{3}(y_0 + 4y_1 + y_2)$$

H está dada por $H = \frac{b-a}{N} = \frac{1-0}{2} = 0.5$

Sustituyendo $S_2 = \frac{H}{3}(f(0) + 4f(0.5) + f(1)) = \frac{0.5}{3}(4 + 4(3.2) + (2)) = \frac{0.5(18.8)}{3} = \frac{9.4}{3}$

$\therefore S_2 = 3.1333333$

Para N=4

$$S_4 = \frac{H}{3}\left(y_0 + 4\sum_{i=1,3,5,...}^{3} y_i + 2\sum_{i=2,4,6,..}^{2} y_i + y_4\right)$$

$$S_4 = \frac{H}{3}\left(y_0 + 4(y_1 + y_3) + 2y_2 + y_4\right)$$

H está dada por $H = \frac{b-a}{N} = \frac{1-0}{4} = 0.25$

Sustituyendo

$$S_4 = \frac{H}{3}(f(0) + 4\left(f(0.25) + f(0.75)\right) + 2f(0.5) + f(1) =$$

$$= \frac{0.25}{3}(4 + 4(3.7647 + 2.56) + 2(3.2) + (2)) = \frac{0.25(37.6988)}{3} = \frac{9.4247}{3}$$

$\therefore S_4 = 3.141566$

Regla de Simpson $\frac{3}{8}$

Para N=3

$$S_{\frac{3}{8}3} = \frac{3H}{8}\left(y_0 + 2\sum_{i=3,6,9,..}^{0} y_i + 3\sum_{i\neq multiplos\ de\ 3}^{1} y_i + y_3\right)$$

$$S_{\frac{3}{8}3} = \frac{3H}{8}\left(y_0 + 3\left(y_1 + y_2\right)\right) + y_3)$$

H está dada por $H = \frac{b-a}{N} = \frac{1-0}{3} = \frac{1}{3} = 0.33333$

Sustituyendo

$$S_{\frac{3}{8}3} = \left(\frac{3}{8}\right)\left(\frac{1}{3}\right)\left(f(0) + 3\left(f(\tfrac{1}{3}) + f(\tfrac{2}{3})\right) + f(1)\right) = \left(\frac{3}{24}\right)\left(4 + 3\left(\frac{18}{5} + \frac{36}{13}\right) + 2)\right) = \left(\frac{3}{24}\right)\left(\frac{1632}{65}\right)$$

$$S_{\frac{3}{8}3} = \frac{204}{65} = 3.133461$$

Para N=6

$$S_{\frac{3}{8}6} = \frac{3H}{8}\left(y_0 + 2\sum_{i=3,6,9,..}^{3} y_i + 3\sum_{i \neq multiplos\ de\ 3}^{4} y_i + y_6\right)$$

$$S_{\frac{3}{8}6} = \frac{3H}{8}\left(y_0 + 2(y_3) + 3(y_1 + y_2 + y_4 + y_5) + y_6\right)$$

H está dada por $H = \frac{b-a}{N} = \frac{1-0}{6} = \frac{1}{6} = 0.1666666$

Sustituyendo

$$S_{\frac{3}{8}6} = \left(\frac{3}{8}\right)\left(\frac{1}{6}\right)\left(f(0) + 2\left(f(\tfrac{3}{6})\right) + 3\left(f(\tfrac{1}{6}) + f(\tfrac{2}{6}) + f(\tfrac{4}{6}) + f(\tfrac{5}{6})\right) + f(1)\right)$$

$$= \left(\frac{3}{48}\right)\left(4 + 2\left(\frac{16}{5}\right) + 3\left(\frac{144}{37} + \frac{18}{5} + \frac{36}{13} + \frac{144}{61}\right) + 2\right) = \left(\frac{3}{48}\right)(50.265334) = 3.141583375$$

$$\therefore S_{\frac{3}{8}6} = 3.141583375$$

Codificando el método de Simpson $\frac{1}{3}$, se tiene que:

```
public class SimpsonUnTercio{
  double limInf;
  double limSup;
  int n;
  double h;
  double integral;

  public static void main (String args[]){
      LeeUnDouble lud = new LeeUnDouble();
      SimpsonUnTercio sut=new SimpsonUnTercio();
      //Empezamos a obtener datos:
      //Obtenemos el limite inferior
      System.out.println("Dame el valor del limite inferior: ");
      lud.leeNumero();
      sut.limInf=lud.num;

      //Obtenemos el limite superior
      System.out.println("Dame el valor del limite superior: ");
      lud.leeNumero();
      sut.limSup=lud.num;

      //Obtenemos el valor de N
      System.out.println("Dame el valor de N: ");
      lud.leeNumero();
      sut.n=(int)lud.num; //cast
```

```
      sut.h=(sut.limSup-sut.limInf)/sut.n;
      sut.integral=(sut.h/3)*(sut.funcion(sut.limInf)+(4*sut.sumaImpares(
      sut.limInf,sut.h,sut.n))+(2*sut.sumaPares(sut.limInf,sut.h,sut.n))+
      sut.funcion(sut.limSup));
         System.out.println("El resultado de la integral es
         :"+sut.integral);
     }

     public double sumaImpares(double inf, double H, double N){
        double suma=0;
        for(int i=1; i<=(N-1); i=i+2){
           suma=suma+funcion(inf+H*i);
        }
        return suma;
     }

     public double sumaPares(double inf, double H, double N){
        double suma=0;
        for(int i=2; i<=(N-2); i=i+2){
          suma=suma+funcion(inf+H*i);
        }
        return suma;
     }

     public double funcion(double x){
        double fX;
        fX=4/(1+x*x);
        return fX;
     }
}
```

Como se puede observar, se utiliza la clase **LeeUnDouble**, vista anteriormente, para obtener los valores de los límites y el valor de N, para realizar la integral de la función. Se implementan los métodos para las sumatorias de los elementos pares, y de los elementos impares, así como la función a la cual se le desea obtener la integral.

Ejercicios Propuestos

1. Implementar en una clase las ecuaciones para obtener la segunda derivada por medio de la serie de Taylor.
   Utilice la misma ecuación vista en el ejemplo, obtenga el valor teórico de la segunda derivada en el mismo punto y calcule los errores absoluto y relativo.
2. Realizar una clase que efectúe la integración numérica mediante el método de Simpson 3/8.
3. Utilice la función del ejemplo para obtener su integral y compare con los resultados del ejemplo.
4. Obtenga la primera y segunda derivada de la siguiente función:

$$f(x) = \frac{e^x}{x^2 - 3}$$

5. Obtenga por medio de la regla de Simpson de 1/3 la siguiente integral:

$$f(x) = \int_0^3 e^x x^2 dx$$

   proponga un valor de N.
6. Obtenga por medio de la regla de Simpson de 3/8 la integral del ejercicio anterior y compare ambos resultados para una misma N.

SOLUCIÓN DE ECUACIONES Y SISTEMAS DE ECUACIONES DIFERENCIALES

Objetivos

El alumno conocerá y aplicará diversos métodos para la resolución de sistemas y ecuaciones diferenciales, implementando programas orientados a objetos.

Al final de esta práctica el alumno podrá:

1. *Resolver ecuaciones diferenciales por diferentes métodos.*
2. *Usar un lenguaje orientado a objetos para implementar dichos métodos*

Antecedentes

3. Haber elaborado programas orientados a objetos en lenguaje Java.
4. Haber manejado ecuaciones diferenciales.

Introducción

MÉTODO DE EULER

Es un método sencillo que se utiliza para la integración de ecuaciones diferenciales de *primer* orden.

Sea $\frac{dy}{dx} = f(x, y)$ con la condición de entorno: $y(x_0) = y_0$

Supongamos que *y(x)* es la solución exacta del problema. Si tomamos un valor de *x* y lo suficientemente próximo a x_0, podemos tomar la siguiente aproximación:

$$y(x) \cong y(x_0) + \left.\frac{dy}{dx}\right|_{X_0} (x - x_0)$$

$$y(x) \cong y(x_0) + \left.\frac{dy}{dx}\right|_{X_0} (x - x_0) = y_0 + f(x - x_0)$$

Así, si por ejemplo, tomamos un valor $x_1 = x_0 + h$, podemos calcular el valor correspondiente $y_1 = y(x_1)$ del siguiente modo:

$$x_1 = x_0 + h$$
$$y_1 \cong y_o + f(x_o, y_o)h$$

Si ahora quisiéramos calcular la solución en un punto superior, partiríamos de:

$\frac{dy}{dx} = f(x, y)$, con la nueva condición (aproximada) de entorno: $y(x_1) \cong y_1$

Si se denotan por $y_1,...,y_N$ los valores aproximados correspondientes a los valores exactos $y(x_1),..., y(x_N)$, entonces quedaría como sigue:

dado y_0 se calcula y_1 mediante la ecuación: $y_1 = y_0 + hf(x_0, y_0)$, y_2 mediante la ecuación $y_2 = y_1 + hf(x_1, y_1)$, y así sucesivamente… $y_N = y_{N-1} + hf(x_{N-1}, y_{N-1})$. De manera general se obtiene la siguiente expresión: $y_{k+1} = y_k + hf(x_k, y_k) \quad k = 0,1,..., N-1$

Ejemplo: Utilizar el método de Euler para aproximar el valor de la solución de la siguiente ecuación diferencial en el punto x = 1, usando h = 0.2 y h = 0.1.

$$f(x,y) = \frac{dy}{dx} = 2x + y; \quad y(0) = 1$$

h=0.2:

$x_1 = x_0 + h = 0 + 0.2 = 0.2$

$y_1 \cong y_0 + f(x_0, y_0)h = 1 + (1 \times 0.2) = 1.2$

$x_2 = x_1 + h = 0.2 + 0.2 = 0.4$

$y_2 \cong y_1 + f(x_1, y_1)h = 1.2 + (1.6 \times 0.2) = 1.52$

$x3 = x_2 + h = 0.4 + 0.2 = 0.6$

$y_3 \cong y_2 + f(x_2, y_2)h = 1.52 + (2.32 \times 0.2) = 1.984$

$x_4 = x_3 + h = 0.6 + 0.2 = 0.8$

$y_4 \cong y_3 + f(x_3, y_3)h = 1.984 + (3.184 \times 0.2) = 2.6208$

$x_5 = x_4 + h = 0.8 + 0.2 = 1.0$

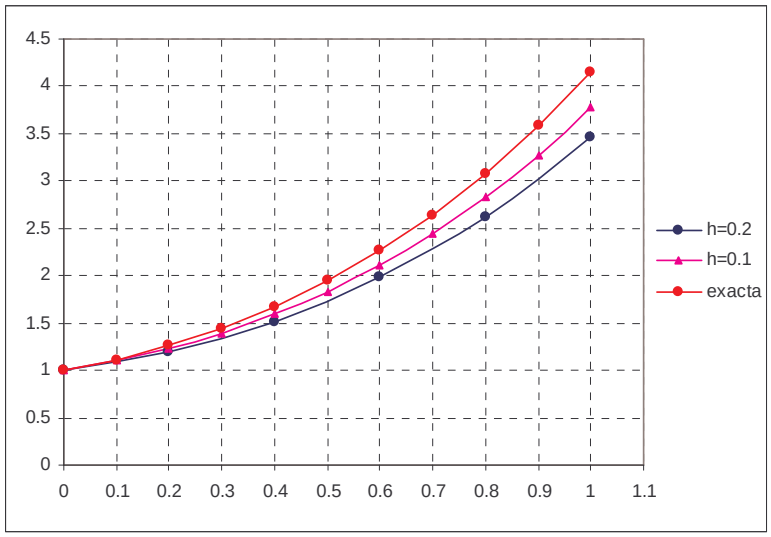
$y_5 \cong y_4 + f(x_4, y_4)h = 2.6208 + (4.2208 \times 0.2) = 3.46496$

h=0.1:

$x_{10} = x_{10} + h = 0.9 + 0.1 = 1.0$

$y_{10} \cong y_{10} + f(x_{10}, y_{10})h = 3.273843 + (5.073843 \times 0.1) = 3.7812273$

Se observa que los valores obtenidos son distintos en ambas aproximaciones, es decir, el resultado varía dependiendo del valor asignado a *h*. Cuanto menor sea *h*, mejor será la aproximación (aunque resulta más largo el procedimiento). Para una *h* constante, el error será tanto mayor cuanto más se aleje del punto inicial, como puede apreciarse en la gráfica siguiente en la que se comparan las dos soluciones aproximadas con la solución exacta.



$$y(x) = -2(x+1) + 3e^{x}$$

Codificando el método anterior para resolver la misma ecuación diferencial, se tiene:

```
public class Euler{
   double h;
   int dec;
   double xo;
   double yo;
   double xi;
   double yi;
   double xii;
   double yii;
   double xf;

   public static void main (String[] args){
      try{
         Redondear r = new Redondear();
         Euler euler = new Euler();
         euler.h = Double.parseDouble( args[0] );
         euler.dec = Integer.parseInt(args[1]);
         euler.xo = Double.parseDouble( args[2] );
         euler.yo = Double.parseDouble( args[3] );
         euler.xf = Double.parseDouble(args[4] );
         euler.xi = euler.xo;
         euler.yi = euler.yo;

//Se inician las iteraciones
         for(int i = 0;euler.xi<euler.xf;i++){
            euler.xii = r.redondeo(euler.xi + euler.h,euler.dec);
            euler.yii = euler.yi + euler.funcion(euler.xi,euler.yi) *
                        euler.h;
            euler.xi = euler.xii;
            euler.yi = euler.yii;
         }
//Se imprime la solución
         System.out.println("La aproximacion de la ecuacion diferencial
```

```
            en "+euler.xii+" es : "+euler.yii);
         }
         catch(ArrayIndexOutOfBoundsException aioobe){
            System.out.println("Faltaron parametros");
            System.out.println("Sintaxis: java Euler h decimales x0 y0 xf");
         }
      }

      public double funcion(double x, double y){
         double temp;
         temp = 2*x+y;
         return temp;
      }
}
```

Como se puede observar, únicamente se utiliza un método que evalúe la función a lo largo de las iteraciones. Del mismo modo, debido a la imprecisión de Java, se vuelve a utilizar la clase Redondear vista en la práctica anterior.
Esta clase recibe como parámetros el valor de *h*, el número de decimales para el redondeo (es recomendable que en este número, si se tiene 0.0035, el número de decimales para el redondeo sea de 4, o si se tiene 0.001, sea de 3), la condición inicial en *x, y*, y el valor final.

MÉTODO DE RUNGE – KUTTA

El método de Runge-Kutta no sigue exactamente la misma línea de los métodos de Euler, ya que está basado únicamente en una aplicación de los polinomios de Taylor.
El método de Runge-Kutta sí contiene como casos especiales los de Euler.
Las fórmulas

$$y_{n+1} = y_n + \frac{1}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right]$$

donde

$$k_1 = h \cdot f(x_n, y_n)$$

$$k_2 = h \cdot f\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1\right)$$

$$k_3 = h \cdot f\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2\right)$$

$$k_4 = h \cdot f\left(x_n + h, y_n + k_3\right)$$

se conocen como las reglas o *fórmulas de Runge-Kutta de orden cuatro* para la ecuación diferencial:

$$y' = f(x, y)$$
$$y(x_0) = y_0$$

Ejemplo
Usar el método de Runge-Kutta para aproximar $y(0.5)$ dada la siguiente ecuación diferencial:

$$y' = 2xy$$
$$y(0) = 1$$

Solución

Primero observamos que esta ecuación sí puede resolverse por métodos tradicionales de ecuaciones diferenciales. Por ejemplo, podemos aplicar el método de separación de variables.

Solución Analítica.

$$\frac{dy}{dx} = 2xy$$
$$\frac{dy}{y} = 2xdx$$
$$\int \frac{dy}{y} = \int 2xdx$$
$$\ln|y| = x^2 + c$$

Sustituyendo la condición inicial:

$$x = 0 \rightarrow y = 1$$
$$\ln 1 = 0^2 + c$$
$$0 = c$$

Por lo tanto, tenemos que la curva solución real está dada:

$$\ln y = x^2$$
$$e^{\ln y} = e^{x^2}$$
$$y = e^{x^2}$$

Y por lo tanto, el valor real que se pide es:

$$y(0.5) = e^{(0.5)^2} = 1.28403$$

Observamos que la distancia entre $x_0 = 0$ y $x_1 = 0.5$ no es lo suficientemente pequeña. Si dividimos esta distancia entre cinco obtenemos un valor de $h = 0.1$ y por lo tanto, obtendremos la aproximación deseada en cinco pasos.

Si contamos con los datos:

$$\begin{cases} x_0 = 0 \\ y_0 = 1 \\ h = 0.1 \\ f(x, y) = 2xy \end{cases}$$

Para poder calcular el valor de y_1, debemos calcular primeros los valores de k_1, k_2, k_3 y k_4. Tenemos entonces que:

$$k_1 = h \cdot f(x_0, y_0) = 0$$

$$k_2 = h \cdot f\left(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_1\right) = 0.1 \cdot (2(0.05)(0.1)) = 0.01$$

$$k_3 = h \cdot f\left(x_0 + \frac{1}{2}h, y_0 + \frac{1}{2}k_2\right) = 0.1 \cdot (2(0.05)(1.005)) = 0.01005$$

$$k_2 = h \cdot f(x_0 + h, y_0 + k_3) = 0.1 \cdot (2(0.1)(1.01005)) = 0.020201$$

$$\therefore y_1 = y_0 + \frac{1}{6}(0 + 2(0.01) + 2(0.01005) + 0.020201) = 1.01005$$

Con el fin de un mayor entendimiento de las fórmulas, veamos la siguiente iteración:

$$x_2 = x_1 + h = 0.2$$

$$k_1 = h \cdot f(x_1, y_1) = 0.1 \cdot (2(0.1)(1.01005)) = 0.020201$$

$$k_2 = h \cdot f\left(x_1 + \frac{1}{2}h, y_1 + \frac{1}{2}k_1\right) = 0.1 \cdot (2(0.15)(1.02010)) = 0.03060$$

$$k_3 = h \cdot f\left(x_1 + \frac{1}{2}h, y_1 + \frac{1}{2}k_2\right) = 0.1 \cdot (2(0.15)(1.02535)) = 0.03076$$

$$k_4 = h \cdot f(x_1 + h, y_1 + k_3) = 0.1 \cdot (2(0.2)(1.04081)) = 0.04163$$

$$\therefore y_2 = y_1 + \frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4] = 1.04081$$

El proceso debe repetirse hasta obtener y_5.Resumimos los resultados en la siguiente tabla:

n	x_n	y_n
0	0	1
1	0.1	1.01005
2	0.2	1.04081
3	0.3	1.09417
4	0.4	1.17351
5	0.5	1.28403

Concluimos que el valor obtenido con el método de Runge-Kutta es: $y(0.5) \approx 1.28403$.

EJERCICIOS PROPUESTOS

1. Ejecutar el ejercicio 1 de la práctica en el programa anterior, utilizando h=0.1, 0.2 y 0.4.
2. Resolver en el programa del ejercicio 1, la siguiente ecuación diferencial

$$y' = 2x + y - 3$$

$$y(2)=1$$

   Aproximar y (2.3) tomando h=0.1 en cada paso del proceso iterativo.
3. Implementar una clase que resuelva el método de Runge-Kutta.
4. Resolver la ecuación diferencial del ejercicio 2 de la práctica.
5. Resolver la siguiente ecuación diferencial por el método de Euler:

$$y' = x + 2xy$$

   Con h=0.1, en un intervalo de $0.5 \le x \le 1$ y con $y(0.5) = 1$

PROGRAMACIÓN ORIENTADA A OBJETOS (ARCHIVOS E HILOS)

Objetivos

El alumno aplicará el concepto de programación orientada a objetos para la realización de programas que involucren el manejo de archivos y creación de hilos.

Al final de esta práctica el alumno podrá:

Implementar programas orientados a objetos, que manipulen archivos; así como, la creación de hilos dentro de una aplicación.

Antecedentes

1. Manejar sentencias de control de flujo en lenguaje de programación Java.
2. Identificar y relacionar varias clases en la programación orientada a objetos empleando la teoría de diseño de jerarquía de clases
3. Haber utilizado archivos en el almacenamiento de datos
4. Manejar los conceptos de proceso y multitareas

Introducción

En Java, para lograr la comunicación entre el programa y la información, ya sea de entrada o salida, ésta se lleva a cabo mediante un flujo, que funge como el intermediario entre el programa y los datos.

En la práctica de *programación orientada a objetos* con *manejo de excepciones*, se aprendió que para hacer la lectura de datos desde el teclado, se tiene que generar un flujo de entrada, es decir, una operación *InputStreamReader*; y el parámetro que recibe es el origen de los datos, en ese caso *System.in*, pero para realizar la lectura de información contenida en archivos o ficheros, se necesita generar un objeto de tipo *File*, igual que como se crea un apuntador *FILE* en C.

```
File f = new File(nombreArchivo);
```

Una vez creado el apuntador, se necesita generar el flujo de datos desde un archivo, es decir, crear un objeto de tipo *FileInputStream*, que recibe como parámetro el objeto de tipo *File*.

```
FileInputStream fis = new FileInputStream(f);
```

Con esto, ya se puede crear el *InputStreamReader*, que permitirá leer los datos que se tengan en un archivo; y el *BufferedReader* que hará la lectura de los mismos.

```
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);
```

Ya con estas cuatro líneas se puede hacer la lectura de un archivo.

Ejemplo 1

Elaborar un programa que lea la información contenida en un archivo de texto plano.

```
import java.io.*;

public class LeeArchivo{

   File f;
   FileInputStream fis;
   InputStreamReader isr;
   BufferedReader br;
   String linea;

   public static void main(String args[]){
      try{
         LeeArchivo la = new LeeArchivo();
         //Se crea un objeto de tipo File
         la.f = new File(args[0]);
         //Se genera el flujo de entrada desde archivo
         la.fis = new FileInputStream(la.f);
         //Se crea el flujo de lectura
         la.isr = new InputStreamReader(la.fis);
         //Se crea el objeto que permitirá la lectura
         la.br = new BufferedReader(la.isr);
         //Se hará la lectura hasta que encuentre una línea nula
         while((la.linea=la.br.readLine())!=null){
            System.out.println(la.linea);
         }

        //Se cierran todos los flujos
         la.br.close();
         la.isr.close();
         la.fis.close();
      }
      catch(IOException ioe){
         System.out.println(ioe.getMessage());
      }
   }
}
```

Debido a que todos los flujos de entrada y salida generan una excepción de entrada/salida, es necesario agregar una secuencia de *try-catch*.

Escritura en archivos

Ahora, para realizar la escritura en archivos se debe generar, de la misma manera que para la lectura, flujos de entrada o salida. En este caso, en lugar de utilizar *FileInputStream*, debemos generar un flujo de salida, es decir *FileOutputStream*.

```
FileOutputStream fos = FileOutputStream(f);
```

Debido a que ahora la impresión es en un archivo, se deberá utilizar el comando:

```
PrintStream ps = PrintStream(fos);
```

Ejemplo 2
Elaborar un programa que escriba en archivo una línea que se escriba desde línea de comando.

```
import java.io.*;

public class EscribeArchivo{
   String cadena;
   File f;
   FileOutputStream fos;
   PrintStream ps;
   public static void main(String args[]){
      try{
         EscribeArchivo ea = new EscribeArchivo();
         //Se obtiene la cadena que se escribirá en el archivo
         ea.cadena = args[0];
         //Se crea un objeto de tipo File
         ea.f = new File(args[1]);
         //Se genera el flujo de salida hacia el archivo
         ea.fos = new FileOutputStream(ea.f);
         //Se genera el escritor del flujo
         ea.ps = new PrintStream(ea.fos);
         //Se escribe la línea dentro del archivo
         ea.ps.println(ea.cadena);
         //Se cierran flujos
         ea.ps.close();
         ea.fos.close();
      }
      catch(IOException ioe){
      }
      catch(ArrayIndexOutOfBoundsException aioobe){
         System.out.println("ERROR!!!");
         System.out.println("Sintaxis: java EscribeArchivo
          \"cadena\" archivo");
      }
   }
}
```

Hilos

Cuando se genera un proceso, se crea un hilo primario, que está asociado a un bloque de instrucciones, un conjunto de registros y una pila. Este hilo, puede crear más hilos si es necesario; a estos hilos se les denomina "procesos ligeros" ya que comparten ciertos recursos con el proceso primario que los creó.

Pero, ¿para qué se necesitan varios hilos? Cuando se hace una aplicación robusta en el que tenga que hacer varias acciones al mismo tiempo, en lugar de crear un proceso completo, es más conveniente crear un hilo que pueda llevar a cabo las acciones correspondientes de una forma rápida y eficaz. Por ejemplo, si se tiene un sistema que está en red, para que múltiples usuarios tengan acceso, si se generara un proceso por cada usuario, saldría bastante caro, hablando de recursos máquina, pero si se genera un hilo para que el usuario haga una consulta, o una actualización de un dato, las acciones se realizárían rápidamente, ya que se minimizaría la carga del acceso al sistema. Por esta razón resulta recomendable la utilización de hilos en aplicaciones robustas.

Hay varios estados por los que pasa un hilo, y son:

- *Preparado*.- Cuando se genera un hilo nuevo y está listo para ejecutarse, o cuando ha sido desbloqueado.
- *En ejecución*.- Cuando está haciendo uso de los recurso de la CPU.
- *Bloqueado*.- Cuando está en espera de que se elimine el bloqueo.
    - *Dormido*.- Cuando se bloquea por un cierto tiempo, después del cual despierta, y pasa a preparado.
    - *En espera*.- Cuando está esperando a que ocurra algo, recibe un mensaje y pasa a preparado.
- *Muerto*.- Cuando ha terminado de ejecutarse.

En Java, cuando se trabaja con hilos, se debe indicar que la clase con la que se está trabajando es una subclase de la clase *Thread*, es decir, que se heredarán todas las propiedades que contiene la clase *Thread*.

Ejemplo 3
Elabore un programa que genere 6 hilos.

```
public class GeneraHilo extends Thread{
   //Rutina que se ejecuta cuando se le da start() a un hilo
   public void run(){
   System.out.println("Esta es una instancia de nuestra clase: "
                      +this.getName());
   }
   public static void main(String args[]){
      //Se generan 6 hilos
      GeneraHilo gh1 = new GeneraHilo();
      GeneraHilo gh2 = new GeneraHilo();
      GeneraHilo gh3 = new GeneraHilo();
```

```
      GeneraHilo gh4 = new GeneraHilo();
      GeneraHilo gh5 = new GeneraHilo();
      GeneraHilo gh6 = new GeneraHilo();
      //Se les asigna nombre
      gh1.setName("Hilo1");
      gh2.setName("Hilo2");
      gh3.setName("Hilo3");
      gh4.setName("Hilo4");
      gh5.setName("Hilo5");
      gh6.setName("Hilo6");
      //Se "ejecutan" los hilos
      gh1.start();
      gh2.start();
      gh3.start();
      gh4.start();
      gh5.start();
      gh6.start();
      System.out.println("Hilo Principal:"
                         +Thread.currentThread().getName());
   }
}
```

EJERCICIOS PROPUESTOS

1. El programa del Ejemplo 1, tiene una falla cuando no se le pasan parámetros, arréglelo utilizando un *catch*. Además, cuando se le da el nombre de un archivo no existente aparece la leyenda:

```
(El sistema no puede hallar el archivo especificado)
```

Modifique el programa de tal forma que se indique:

```
ERROR!!!
Archivo no válido, favor de verificar el nombre del archivo.
```

TIP: Utilice una excepción para la modificación de archivo no válido.

2. Modifique el programa del Ejemplo 2 de tal forma que ahora se escriba en archivo todo lo que se escriba mientras está en ejecución el programa.

3. Modifique el programa anterior, para que también se pueda agregar información al final del archivo que se indique.

4. Elabore un programa que lea información de un archivo, y lo escriba en otro.

5. Cuando se ejecuta varias veces seguidas el programa del Ejemplo 3, ¿qué pasa? Ahora en el método run(), agregue la instrucción `sleep((long)Math.random()*2500)`, antes de la impresión del nombre del hilo que se está ejecutando. Haga las modificaciones necesarias para que compile y ejecute correctamente el programa. Si se ejecuta varias veces seguidas el programa, ¿Ahora, qué sucede?

BIBLIOGRAFIA

ANTONAKOS, James L.
MANSFIELD JR, Kenneth C.
Programación estructurada en C
Editorial Pearson/Prentice Hall
Madrid 1997

BISHOP, Judy
Java: Fundamentos de programación
Addison-Wesley Iberoamericana
España 1999. 2ª edición

BOOCH, Grady
JACOBSON, Ivar
RUMBAUGH, James
El lenguaje unificado de modelado
Addison Wesley
Madrid 1999.

BURDEN, L. R.
FAIRES, J.D.
Análisis Numérico
Thomson Internacional
México 2003. 7ª. Edición

CAIRÓ, Osvaldo
Metodología de la programación. Algoritmos, diagramas de flujo y programas.
Editorial Alfaomega
México 2003. 2ª edición.

CEBALLOS, Fco. Javier
El lenguaje de programación C#
Alfaomega
México 2002. 1ª edición

CEBALLOS, Fco. Javier
Java 2
Alfaomega/Ra-Ma
Madrid 2003. 2ª edición.

CHAPRA, Steven C
CANALE, Raymond P.
Métodos Numéricos para Ingenieros
Mc. Graw-Hill,
México, 1999 3ª. Edición

DEITEL, Harvey
DEITEL, Paul J.
Cómo programar en Java
Pearson Educación
México 2004. 5ª edición

DEITEL, Harvey M.
DEITEL, Paul J.
Cómo programar en C/C++ y Java
Pearson Educación
México 2004. 4ª edición

FLANAGAN, David
Java in a nutshell
O'Rilley
USA, 1999 2ª edición

GERALD, Curtis F.
Análisis Numérico
Prentice Hall
México, 2001 6ª Edición

GERALD, Curtis F.
WHEATLEY, Patrick O.
Análisis Numérico con aplicaciones
Pearson Educación
México, 2000 6ª edición

GRAU, Miguel
NOGUERA, Miguel
Cálculo Numérico
UPC Barcelona, 2001 1ª edición

HUERTA, Antonio
SARRATE, Josep
RODRÍGUEZ-FERRAN, Antonio
Métodos Numéricos: Introducción, aplicaciones y programación
UPC
Barcelona, 1998 1

KELLEY, Al
POHL, Ira
Lenguaje C: Introducción a la programación
Addison-Wesley Iberoamericana S. A.
Washington Delaware 1987

KERNIGHAN, Brian W.
RITCHIE, Dennis M.
El lenguaje de programación C
Prentice-Hall Hispanoamericana
México, 1991

MARON, Melvin J.
LOPEZ, Robert J.
Análisis Numérico: Un enfoque práctico
Compañía Editorial Continental
México, 1995 1ª edición

NAKAMURA, Shoichiro
Métodos numéricos aplicados con software
Prentice-Hall Hispanoamericana
México, 1992 1ª edición

NAKAMURA, Shoichiro
Applied Numerical Methods in C
Prentice-Hall
USA, 1992