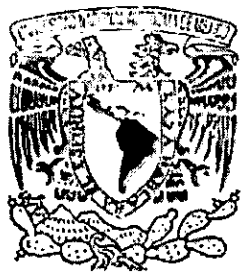
24



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
"ACATLAN"

TUTORIAL DE MÉTODOS NUMÉRICOS EN DELPHI

TESINA

QUE PARA OBTENER EL TÍTULO DE

LICENCIADO EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA:

BERNARDINO PEREA GUZMÁN

ASESOR: M. EN S.I. ALMA LÓPEZ BLANCO

ABRIL 2000

Dedicatoria

A María Agustina Guzmán Nevárez

Si de alguna forma se pudieran pagar esas noches de desvelo esperándome de regreso de la escuela, o por las mañanas el recordatorio de un nuevo día para ir a trabajar, esos gustos que no te pudiste dar por comprarme un libro o tomar un curso, ese soporte de ser padre y madre a la vez, o las palabras de apoyo para terminar un trabajo final. Ese esfuerzo es de los dos y está reflejado en este trabajo.

Gracias mamá

Agradecimientos

A mis padres, que me dieron la vida y la oportunidad de estudiar.

A mi hermano Luis, por su apoyo y paciencia, por estar ahí cuando más lo necesite, porque siempre tuvo tiempo para enseñar y explicar.

A Nancy, mi esposa, por su ayuda y comprensión, por esas palabras de aliento cuando algo iba mal.

A mi asesor Alma López por su tiempo y comentarios para llevar adelante este trabajo

A mis sinodales Sara Camacho, Georgina Eslava, Andrés Hernández y Omar Espinosa por sus acertadas observaciones.

Y a todos los profesores que a lo largo de mi vida estudiantil han sido mi guía, por entregar sin condición sus conocimientos y experiencias.

Contenido

Introducción

Los métodos numéricos han formado parte del desarrollo matemático en toda la historia. La mayor parte de los problemas matemáticos no cuentan con soluciones analíticas, y en general, es más fácil aproximarlos con algún algoritmo. Recordemos que antes del desarrollo de las computadoras, se contrataban personas para desarrollar una serie de operaciones a mano para aproximar funciones, por ejemplo la curva normal, y con ellas se escribían grandes tomos conteniendo las tablas de esos valores aproximados.

En la actualidad, se siguen presentando problemas sin solución exacta, y se requiere de métodos que los aproximen con la mayor precisión posible. Afortunadamente, contamos cada vez con mejores y más rápidas computadoras y lenguajes de programación.

De entre las ventajas de los nuevos lenguajes de programación, hay una que sobresale: el hecho de poder reutilizar el código. Es decir, una persona escribe un programa para una solución en especial, pero lo deja parametrizado de tal modo que otra persona puede tomarlo, agregarle funciones si así lo desea, y utilizarlo para resolver otros problemas.

A lo largo de nuestras vidas, como programadores, desarrollamos un hábito o estilo de programación. Generalmente reciclamos los mejores programas, los que nunca "tronaron", los de mejor presentación, aquellas funciones que siempre se utilizan; y cada vez las mejoramos aún más. Si reunimos estos programas y los preparamos para volverlos a usar en cualquier momento, entonces estamos creando una biblioteca de funciones, librerías y programas.

El objetivo del presente trabajo es desarrollar una biblioteca de componentes que faciliten la creación de programas tanto de métodos numéricos como de índole general, utilizando el lenguaje de programación Delphi 3 bajo ambiente Windows. Adicionalmente se desea que éstos sirvan como base para el desarrollo de nuevos componentes.

Los métodos numéricos que se eligieron para ser presentados en este trabajo, son métodos que complementan a los enseñados en los temarios del plan de estudios de la licenciatura en Matemáticas Aplicadas y Computación. Se pretende con esto que el lector conozca otros métodos numéricos, además de como programar sus algoritmos en componentes.

Métodos numéricos, como la solución de ecuaciones lineales, operaciones con polinomios, interpolación y evaluación de funciones, integrales numéricas e impropias, entre otras, conformarán esta biblioteca de componentes.

Un tutorial es una iniciación guiada a un tema en específico con el uso de un conjunto de conceptos o una técnica. Proporciona la información práctica sobre un tema específico. Y este trabajo lleva el nombre de tutorial porque en su alcance se desea mostrar el uso de la programación orientada a objetos para el desarrollo de programas reutilizables, o componentes, básicamente en el tema de los métodos numéricos. Aquí se muestra la teoría de los métodos recopilados, junto a sus algoritmos y programas dirigidos a obtener componentes. Finalmente, se desarrolla una aplicación donde se puede apreciar el uso, tanto de los componentes, como de los métodos numéricos.

Se desea que al leer este trabajo puedan basarse nuevos desarrollos en estos ejemplos, como un inicio, listo para ser mejorados y adecuados a las necesidades propias de cada problema.

Los componentes ha desarrollar se tomaron como una muestra pequeña de los principales problemas a los que nos enfrentamos al programar problemas matemáticos y nuestra experiencia en el ámbito de la programación es escasa. Problemas como el manejo de matrices, evaluaciones de funciones y tópicos de cálculo.

La mayoría de los algoritmos, de los métodos numéricos, que aquí se presentan fueron desarrollados por otras personas, estos fueron publicados en su tiempo en lenguajes de programación como Fortran o C, usando la técnica estructurada. La aportación que se hace en este trabajo a estos algoritmos es su actualización al lenguaje Delphi y a la técnica orientada a objetos, además de encapsular su implementación en componentes. Los algoritmos para leer cualquier función y derivarlas es un desarrollo propio.

Delphi es la versión para Windows del lenguaje Pascal orientado a objetos desarrollado por Borland. Pascal se ha convertido en un lenguaje popular entre las comunidades académicas por ser de carácter sencillo, tanto de aprender como de utilizar. Por estas características se eligió para programar los algoritmos de los métodos numéricos.

Este tutorial asume un buen conocimiento de Pascal, así cómo crear programas sencillos en Delphi. Sería un beneficio para el lector tener un conocimiento previo de programación orientada a objetos, pero no es del todo necesario, puesto que la creación de los componentes se lleva a cabo paso a paso y, al final, se muestra el listado completo del componente.

El trabajo se divide en seis capítulos agrupando componentes de un mismo tipo.

En el primer capítulo se explica que son los componentes y de los motivos para usarlos. Asimismo, la importancia del uso de bibliotecas existentes, como los sistemas operativos, y los beneficios que se pueden obtener de la reutilización del código. Finalizando con una explicación de cómo hacer componentes en Delphi.

En el segundo capítulo se desarrolla un componente ejemplificando su uso en operaciones con matrices. Se habla concretamente de la solución de sistemas de ecuaciones lineales, utilizando un método rápido y sencillo: la descomposición LU, análoga a la solución por el método de Gauss Jordan. Este componente puede servir de base para desarrollar otros que también utilicen matrices extensivamente.

En el tercer capítulo se presentan algunos consejos para operar polinomios con pocas operaciones, además, se incluyen componentes para evaluarlos junto con sus derivadas y para dividir dos polinomios usando la división sintética. Por otro lado, la evaluación de funciones tabuladas se ataca usando interpolaciones por dos métodos, por polinomios y por fracciones racionales, para cada uno se desarrolla un componente.

En el capítulo cuarto se habla de los métodos y problemas que se presentan al tratar de implementar sistemas que requieren leer y evaluar cualquier función que desee el usuario. Se plantean algunos métodos para resolver el problema y se muestra la forma de implementar un componente para leer funciones matemáticas, evaluarlas e incluso derivarlas. Adicionalmente se presenta un componente para encontrar la derivada numérica de una función.

En el quinto capítulo se desarrollan diversos componentes para evaluar distintas clases de integrales utilizando métodos numéricos, desde los métodos clásicos de integración cerrada, hasta la solución de varias clases de integrales impropias.

En el último capítulo se encuentran algunos programas que ejemplifican el uso de los componentes desarrollados a lo largo del trabajo. Creados sobre un mismo esquema, estos programas pretenden mostrar una forma sencilla y rápida para crear sistemas basándose en la reutilización de código.

Los programas que en este programa se listan fueron compilados en la versión de Delphi 3 de Borland, utilizando una máquina IBM PC compatible con procesador Intel Pentium a 166 Mhz y 80 MB en RAM, sobre Windows 95.

Capítulo 1

Bases

Es un hecho que el hombre se ha desarrollado tecnológicamente a lo largo de la historia gracias a los descubrimientos. Por lo general, un investigador no realiza todos los experimentos hasta llegar a un punto donde comience algo nuevo, sino simplemente asume como cierto algún punto de investigación de alguien más para comenzar de ahí con la suya.

El hombre no está conforme con lo que sabe y siempre anda en busca de algo más. Siempre apoyándose en los conocimientos de sus antepasados. Este concepto aplica también a la historia de la computación, que es otra rama del conocimiento del hombre. Sin embargo, para los que apenas comenzamos, es difícil identificar dónde se localizan esas bases de conocimiento reutilizables. Para otros, que ya cuentan con más experiencia, los conocimientos son su "negocio", lo que les permite sobrevivir en los campos de trabajo, o sea, que se vuelven "egoístas" y protegen sus fuentes de información como un gran tesoro. Un ejemplo de este caso es que los códigos fuente son patentados ante los derechos de autor. El caso del código fuente del sistema operativo UNIX es especial, pues es gratuito y puede ser copiado y modificado por cualquier persona, no así con el de Windows.

Aún así, siendo abiertos o parcialmente cerrados con la información, todavía se puede rescatar algo de ese legado de conocimientos.

Hace unos diez años, las máquinas llevaban por nombre XT 8086 por el tipo de procesador que incluían, contaban con 512KB en RAM, un disco flexible de 5.25 de 360KB y monitor CGA de cuatro tonos de ámbar. El sistema operativo que las hacía funcionar era MS-DOS ver 2.01.

En aquel entonces, antes del primer contacto con una máquina de este tipo, uno podía imaginar a las computadoras como algo tan complejo que podía resolver cálculos complicados con sólo indicárselo, pero la realidad era otra, la máquina sólo respondía con un parpadeo incesante del cursor sobre la pantalla siguiendo a un símbolo A> (ver figura 1.1).

```
Microsoft(R) MS-DOS ver. 2.01
   (C)Copyright Microsoft Corp 1981-1986.

A>_
```

Fig. 1.1 Pantalla de MS-DOS iniciando desde la unidad A>

En estas primeras máquinas no existía esa "magia" deseada, sin embargo, lograr ese parpadeo del cursor esperando instrucciones del usuario para ser ejecutadas, era ya un gigantesco avance tecnológico y de conocimientos que permitiría el increíble desarrollo de la computación en nuestros días.

Los sistemas de cómputo se han desarrollado gracias a que generación tras generación se han desarrollado nuevas técnicas, métodos y tecnologías para lograr ese avance.

El primer adelanto fue poder programar las computadoras basándose en tarjetas perforadas, ya no era necesario modificar los circuitos, ni cambiar los bulbos para cambiar los programas, sólo se necesitaba ingresar un conjunto de tarjetas que representaba el programa y otro tanto los datos. La máquina realizaba las operaciones y entregaba resultados impresos o visuales en tableros especiales.

El segundo paso fue cuando se introdujeron los lenguajes de programación de primer nivel, ya no eran necesarias las tarjetas pues se podía almacenar la información en cintas magnéticas, ahora bastaba con escribir las instrucciones y los datos para que la máquina realizara las operaciones.

Otro avance consistió en mejorar los lenguajes de programación, haciéndolos más cercanos al lenguaje del hombre, el primero se refería al lenguaje de máquina, sólo que no se necesitaban las tarjetas, le siguió el ensamblador, el cual contenía una abstracción mayor de las instrucciones de máquina. Los lenguajes de tercera generación aparecieron y lo hicieron con una semejanza al inglés como nunca, esto permitió que se acelerara el proceso de desarrollo de sistemas, lenguajes como Fortran y Cobol, luego C y Pascal, hicieron una revolución.

Y es en esa revolución donde comenzaremos.

Así como los lenguajes de programación evolucionaban, también lo iban haciendo los sistemas operativos.

La primera forma de biblioteca de utilerías para programar se encuentra precisamente en los sistemas operativos. Gracias a ellos los programadores ya no tenían que preocuparse por desarrollar, cada vez que hacían un sistema, una interfaz con el usuario.

Los sistemas operativos sirven como intermediarios entre el usuario y la máquina, se encargan de traducir las peticiones del usuario en un lenguaje comprensible por él, a instrucciones que puede realizar la máquina.

Debido a que en nuestro país es más común utilizar computadoras personales con sistema operativo DOS es el motivo por el cual el trabajo se enfoca a este sistema operativo en particular

1.1 Sistemas Operativos

Desde el punto de vista de Peter Norton, él define lo que es un sistema operativo a partir de su función dentro de una computadora, detallando y definiendo cada concepto para que así quede más claro el primero.

"Una computadora es una máquina diseñada para realizar distintas tareas (siguiendo una secuencia de instrucciones), a diferencia de la mayoría de las máquinas que ejecutan sólo una acción" [Norton, 1991].

"Una lista de instrucciones que sigue la computadora es llamada un *programa*. Cuando la computadora sigue las instrucciones, decimos que *ejecuta* o *corre* el programa. Podemos correr ciertos programas para que la computadora realice una acción deseada. Estos pueden ir desde los más sencillos como desplegar la fecha, hasta sofisticados procesadores de palabras" [Norton, 1991].

"El *Hardware* se refiere a las partes físicas de una computadora: el monitor, el teclado, los discos, la impresora, el ratón, y otros más. El *Software* consiste de todos los programas que podemos correr en ella. Aunque "hard" y "soft" son opuestos, hardware no es lo contrario de software. De hecho, son complementarios. Es el software –la lista de instrucciones– que traen a la vida al hardware" [Norton, 1991].

"El software y hardware son complejos, tan complejos en sí, que cada computadora requiere un programa especial para manipular sus recursos. Este programa de control maestro es llamado un ***sistema operativo***" [Norton, 1991].

De esta forma, un sistema operativo es el programa que maneja los recursos de un sistema de cómputo.

Otra definición de sistema operativo también de Peter Norton es "lo que hace que una computadora corra. Específicamente, es el sistema de software que opera el hardware de la computadora, permitiéndole realizar las tareas que el usuario requiera" [Norton, 1993].

Para Harry Katzan un sistema operativo "es una colección de programas y datos que son diseñados para manejar los recursos de un sistema de cómputo para facilitar la creación de nuevos programas y controlar su ejecución en el sistema" [Katzan, 1973].

Y, como dice Federico Antolín, "los sistemas operativos actúan como intermediarios entre el usuario y el resto del sistema" [Antolín, 1997].

Como podemos ver, un sistema operativo es un programa que actúa como mediador de los recursos y como una plataforma que ayuda a la ejecución y creación de otros programas.

Podemos agregar que la función principal de cualquier sistema operativo es la de servir como una interfaz entre el hardware de la computadora, el usuario y los programas de aplicación.

Cuando hablamos de las funciones de un sistema operativo, estamos ya tratando de definir lo que un sistema operativo hace. Al respecto tenemos que los dos propósitos principales de un sistema operativo son:

1. Manejar los recursos del hardware de la computadora eficientemente
2. Hacer *fácil* el uso de la computadora para los usuarios.

Para alcanzar estos objetivos, el sistema operativo se involucra con cada faceta de la operación de la computadora. Se encarga, por ejemplo, de ejecutar los programas, del almacenamiento y recuperación de los datos, de la impresión de listados y del despliegue de las imágenes. De hecho, una razón para que los sistemas operativos existan es que ni nosotros ni los programas necesitemos atender cada milisegundo los detalles necesarios para manejar una computadora.

1.2 MS-DOS, una gran biblioteca

Una biblioteca de utilerías de programación la podemos definir como el conjunto de programas que cuentan con una interfaz definida y que son reutilizables.

MS-DOS consiste básicamente de ***interrupciones***[1] con las que se puede programar, cada una tiene definida una interfaz con el usuario para realizar tareas del hardware pero sin tener que preocuparse más que por sólo llamarlas. Por ejemplo, la interrupción 21 contiene un extenso número de funciones que tienen que ver con los dispositivos de entrada / salida.

Los primeros programas ejecutados sobre DOS hacían un uso extensivo del lenguaje ensamblador y duplicaban las funciones que el sistema operativo ya incluía, esto ocurría porque no se confiaba en él o porque se desconocían estas funciones.

Pero con el paso del tiempo, los programadores se fueron dando cuenta que eran muy útiles las funciones que venían listas para usar dentro del sistema operativo, lo que permitió que los sistemas se desarrollaran rápidamente.

Un factor que aceleró aún más el proceso de desarrollo, fueron los lenguajes de tercera generación. Estos incluyen un conjunto de palabras que los asemejan al idioma del hombre, en este caso inglés. Estos lenguajes en realidad son otro conjunto de macrobibliotecas de funciones, en su corazón radican miles de microprogramas escritos en ensamblador que se van entrelazando conforme se compilan en un programa de estos lenguajes.

---

[1] Una interrupción es un programa que forma parte del sistema operativo, su característica principal es que pueden ser ejecutados por algún evento de hardware así como por el usuario en cualquier momento.

Por ejemplo, la función ***readln*** o ***scanf*** de Pascal y C, sirven para leer un dato del teclado, el programador no debe preocuparse más si debe mandar llamar una interrupción, capturar la información del teclado o almacenar espacio de memoria, sólo tiene que usarlas y listo, un dato se ha ingresado. Lo mismo ocurre con write, printf, los ciclos, y todas las funciones que vienen con el lenguaje.

Con los años, el sistema operativo fue modificado para hacerlo más sencillo y se ha trasladado a un ambiente más agradable para el usuario, con la aparición de Windows el usuario se enfrentaba a una interfaz más agradable e intuitiva, pero los primeros intentos para programar en él resultaban difíciles, pues no se confiaba en él, y a la larga lista de etcétera que aplicaban a los primeros programas en DOS.

Windows también contiene una serie de macrobibliotecas prefabricadas y listas para usarlas. La más notoria es la interfaz gráfica, ya no es necesario intentar realizar una serie de funciones de inicialización de gráficas, ventanas, manejo del mouse y cursor, pues todo viene incluido con Windows.

En Windows ya no se pueden usar las interrupciones como en DOS, aunque Windows corre sobre DOS, ha bloqueado las interrupciones para sustituirlas con funciones más poderosas. Ahora las funciones reusables se encuentran en los archivos del núcleo del sistema Kernel.exe y User.exe. Aquí se encuentran todas las funciones relativas al manejo de las ventanas y de los eventos, así como el manejo de entradas y salidas.

Una opción más interesante es el uso de los archivos DLL (Dynamic Link Libraries) que son bibliotecas de funciones listas para usarse por cualquier programa que corra sobre Windows.

El objetivo de las librerías es ahorrar tiempo en el proceso de la programación de sistemas al proporcionarnos una base estable, sobre la cual el desarrollo es sencillo.

No por programar en Windows se deja de programar en ensamblador, aún Windows permite la creación de programas en este lenguaje pero ahora se le puede indicar con una instrucción que se llame a una función interna de Windows.

Del mismo modo en que el software se hace más complejo, el uso de componentes se convierte en el método preferido para desarrollar aplicaciones. Por un lado, hay compañías dedicadas al desarrollo de programas para problemas generales, distribuibles en formatos compatibles con muchos lenguajes, como son los componentes OCX o DLL's ; y por el otro, los programadores se dedican al desarrollo particular de la aplicación, pues pueden encontrar mucho del desarrollo en el mercado.

1.3 Componentes

Un *componente* es (en su definición más simple) un programa. Pero no es cualquier clase de programa, sino uno con características muy particulares. Implementado en cada componente hay

algunas características propias del paradigma orientado a objetos, tales como la *encapsulación*, *polimorfismo* y *especialización*.

El enfoque tradicional de construcción de software normalmente implica la captura de decenas hasta miles de líneas de código. Los componentes proveen una manera diferente de construir software.

Un ejemplo:
Pensemos en la manera convencional de construir un sistema que maneje los diferentes tipos de cuentas que se manejan en un banco (tarjeta de crédito, crédito hipotecario, cheques, etc.). Normalmente lo que se haría seria separar el proyecto en módulos, cada uno de los cuales manejaría cada una de las cuentas (esta es solo una de varias posibles soluciones), creando especificaciones y posteriormente programas, que (en el mejor de los casos) utilizarían algún tipo de biblioteca de código o funciones; después, la compilación y las ligas con los programas para obtener el nuevo sistema.

¿Cómo desarrollaríamos el mismo proyecto basado en componentes?
Enfoquémonos en este momento en los detalles de implementación de los componentes.
Con las especificaciones de componentes podríamos construir componentes **pequeños, independientes, encapsulados** (es decir, que esconden los detalles de la implementación física), y que se enlazan *dinámicamente* (en tiempo de ejecución), ver fig 1.2.

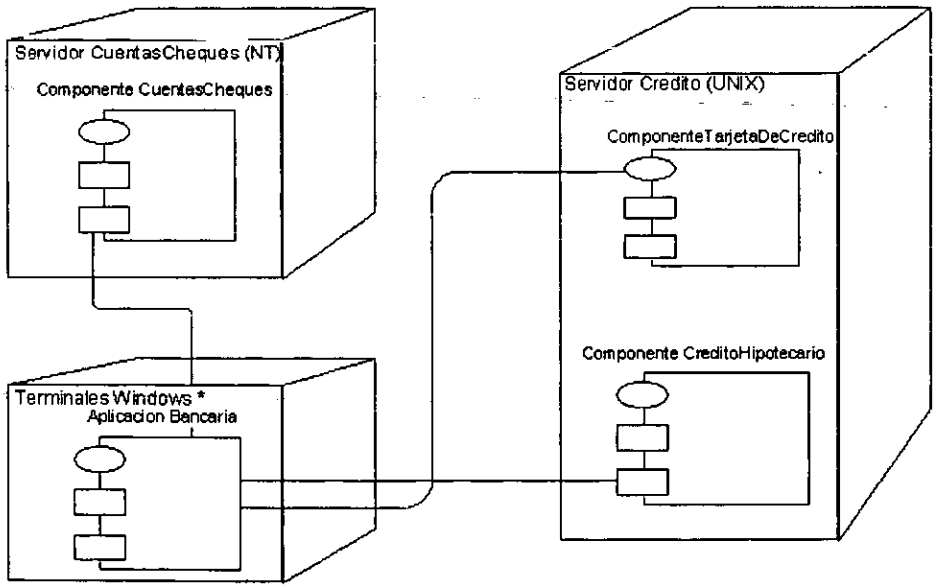


Fig 1.2. Diagrama de implementación de Componentes

Una vez construidos los diferentes componentes (digamos uno para tarjeta de crédito, otro para cuenta de cheques, etc.) utilizamos su funcionalidad por medio de las *interfaces* que cada componente publica.
Hasta este momento el funcionamiento es el mismo (aparentemente). Pero, ¿qué tal si decidimos manejar un nuevo tipo de cuenta?

Con el primer enfoque (tradicional), necesitaríamos hacer un nuevo programa, modificar o agregar (en una situación ideal) algunas líneas del código existente y recompilar nuestro nuevo sistema.

Con el segundo enfoque (componentes), necesitaríamos crear un nuevo componente que manejara el nuevo tipo de cuenta. Después, lo **único** que tendríamos que hacer es el uso de sus nuevas funciones, disponibles a través de una interfaz estándar.

Los componentes son poderosos porque ocultan todos los detalles de la implementación al usuario. Así como un conductor no necesita entender el funcionamiento del motor de combustión para manejar un auto, una aplicación no necesita saber cómo trabaja un componente visual para poder usarlo. Sólo necesita saber como comunicarse con él. Existen varias razones por las que debemos escribir nuestros propios componentes, la principal es la reutilización del código.

Existen varios lenguajes de programación en la actualidad que permiten el desarrollo de componentes. Visual Basic en sus versiones 5 y 6 cuentan con asistentes para crear componentes OCX fáciles de utilizar en otros lenguajes con capacidad OLE. C++ Builder y Delphi pueden crear componentes nativos a estos lenguajes, además de permitir el desarrollo de los mismos en componentes ActiveX. Visual C y C++ además tienen la capacidad de generar archivos DLL para versiones anteriores de Windows. Java también genera código reutilizable por medio de paquetes de clases.

En este trabajo se decidió por utilizar a Delphi por ser de carácter sencillo, tanto de aprender como de utilizar, debido a que es la materialización del lenguaje Pascal orientado a objetos en ambiente gráfico.

Debido a que Delphi es un lenguaje orientado a objetos [Osier, 1997], y un componente visual es un objeto, podemos crear componentes que sean una subclase de otros objetos que ya existen. En otras palabras, mientras desarrollamos componentes ya estamos haciendo uso de código generado por alguien más, ya sea porque comenzamos uno nuevo o porque hacemos más especializado a otro.

Las precursoras de los componentes son las Bibliotecas de Liga Dinámica o DLLs por sus siglas en inglés Dynamic Link Libraries. Estas librerías son poderosas, pero no están orientadas a objetos, su función es permitir crear bibliotecas de funciones y procedimientos en un lenguaje y poder utilizarlas en cualquier otro. Otro rasgo distintivo de las DLLs es que pueden ser cargadas en memoria al momento de la ejecución de modo opuesto a permanecer ligadas a la aplicación. Esto permite dividir el desarrollo y distribución en varios componentes en lugar de desarrollar y ligar estáticamente una aplicación grande.

Para crear un componente en Delphi se puede utilizar el "Component Expert" para generar automáticamente el código base. Todos los componentes deben ser heredados de otros componentes. Si se desea comenzar desde cero, entonces se debe crear una subclase de la clase TComponent. El listado 1.1 muestra el código que el experto produjo para un nuevo componente TNuevo heredado de TComponent.

Listado 1.1. Código básico para crear un componente visual.

```
unit Nuevo;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TNuevo = class(TComponent)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Ejemplos', [TNuevo]);
end;

end.
```

Generalmente debemos modificar el código básico para agregarle funcionalidad al componente. Existen dos partes principales en este código: la clase TNuevo y el procedimiento Register. Delphi usa todo el código asociado a la clase para determinar como funciona el componente. También llama al procedimiento Register para colocar el componente en la paleta de herramientas. En este caso, simplemente instala el componente en el tabulador Ejemplos.

1.4 Escribiendo un componente

Para usar un componente se coloca en una forma o se usa el método ***create***. Después de crear un componente, se puede manipular modificando sus propiedades, llamando a sus métodos, y respondiendo a eventos. No tenemos por que preocuparnos en como funcionan las propiedades y los métodos. El componente llama a su manejador de eventos cuando ocurre un evento en particular.

Entonces, al escribir un componente, debemos describir los detalles de implementación para propiedades y métodos y llamar al manejador de eventos cuando el evento pueda ocurrir. El modo de hacer esto es definir una clase que se convierta en el componente y llenar los detalles

necesarios. Estas propiedades y métodos pueden estar presentes para el usuario o no, para esto deben declararse de una forma en especial indicando su alcance.

Private, Protect, Public y Published

Delphi usa clases de objetos para crear componentes visuales. Diferentes partes de la definición de la clase del objeto son declaradas en diferentes regiones protegidas. Variables, procedimientos y funciones pueden tener cuatro tipos de acceso:

Private (*Privado*). Sólo procedimientos y funciones definidas dentro de la definición de la clase tienen acceso, y sólo rutinas en la misma unidad tienen acceso.

Protected (*Protegido*). Procedimientos y funciones definidas en la definición de la clase tienen acceso, y también procedimientos y funciones de clases descendientes.

Public (*Público*). Todos los procedimientos y funciones tienen acceso.

Published. (*Publicado*). Al igual que *Public*, todos los procedimientos y funciones tienen acceso pero con la cualidad de aparecer en el ambiente de desarrollo (IDE) de Delphi para desplegar información en las páginas de eventos y propiedades.

Propiedades

Un programador en Delphi usa las propiedades de un componente para leer o asignar ciertos atributos de modo similar a los campos almacenados en una clase. Las propiedades, sin embargo, pueden lanzar código para ser ejecutado. Por ejemplo, cuando se cambia la propiedad Shape (figura) en el componente TShape, el componente cambia de figura. Debe existir un mecanismo para informar al componente que debe cambiar la figura al cambiar la propiedad. En otras palabras, una propiedad puede tomar dos personalidades: Puede ser una pieza de información que afecta cómo trabaja un componente, o puede lanzar una acción.

Métodos

Los métodos son funciones y procedimientos que una clase ha hecho públicos. Aunque se pueden usar las propiedades para llamar a una función o a un procedimiento, úselas sólo si es lógico hacerlo. Los métodos, por otro lado, pueden usarse en cualquier momento. Otra diferencia es que las propiedades sólo incluyen una pieza de información. Los métodos pueden tomar múltiples parámetros y pueden regresar múltiples piezas de información a través de la declaración de variables VAR.

Eventos

Los eventos permiten a los programadores o usuarios enriquecer al componente cuando algo sucede –o sea, cuando ocurre un evento. Por ejemplo, el evento OnClick significa en efecto, "Si desea hacer algo cuando el usuario presione el botón del ratón sobre éste componente, dígame que procedimiento ejecutar." Este es el trabajo del diseñador del componente llamar a los eventos del usuario dentro del componente cuando sea necesario.

Agregando propiedades

Las propiedades pueden ser de dos tipos: las que son disponibles mientras se diseña y a las que sólo se puede acceder cuando el programa se está ejecutando. Cual sea el tipo de propiedad que se esté diseñando, cada una esta almacenando información y debe definirse una variable por cada una en la clase. Los usuarios no acceden directamente a las variables, sino a través de una llamada especializada. Las variables que almacenan la información para cada propiedad debe declararse en la sección privada de la definición de la clase –lo que significa que sólo procedimientos y funciones de la clase pueden accederlas. Por convención, los nombres de las variables comienzan con F (que se entiende como "field" campo), seguido del nombre de la propiedad. Por ejemplo:

```
type
   TMult = class(TComponent)
Private
   FVal1 : Integer;
   FVal2 : Integer;
   FRes : Integer;
Protected
Public
Published
end;
```

Ahora debe declarar a las propiedades. Use la palabra reservada ***property*** en la definición de la clase. La definición de una propiedad aparece típicamente en uno de dos lugares. Si una propiedad estará disponible durante el diseño, debe aparecer en la sección published de la declaración. Si estará disponible sólo durante la ejecución del programa, debe colocarse en la sección public. Es posible usar un acceso directo para leer y escribir la propiedad. Con acceso directo puede indicar a Delphi que ponga o regrese el valor de una variable cuando una propiedad es escrita o leída. Los métodos ***read*** y ***write*** definen la variable. A continuación un ejemplo de definición de propiedades:

```
type
   TMult = class(TComponent)
Private
   FVal1 : Integer;
   FVal2 : Integer;
   FRes : Integer;
Protected
Public
   Property Res : Integer read FRes; {propiedad para obtener el resultado, sólo
lectura}
```

```
Published
   Property Val1 : Integer read FVal1 write FVal1 default 1;
   Property Val1 : Integer read FVal1 write FVal1 default 1;
end;
```

Cuando se requiera de propiedades de sólo lectura, no es necesario agregar el método de acceso directo de escritura a esa propiedad.

Agregando el Constructor

Un procedimiento constructor es llamado cuando se crea una clase. Es responsable del almacenamiento dinámico de memoria o garantiza los recursos que una clase necesite. El constructor también asigna valores por defecto a variables en una clase. Cuando un componente es agregado a una forma durante el diseño o la corrida, el constructor es llamado. Para declarar al constructor en la definición de la clase, agregue una línea ***constructor*** en la porción ***public*** de la declaración de la clase. Por convención, ***Create*** es usado como el nombre del procedimiento constructor, como se muestra en este ejemplo:

```
{...}
public
constructor  Create(AOwner : TComponent); override;
{...}
```

Un parámetro es pasado al constructor –el componente que lo adquiere. Es diferente de la propiedad ***parent***. Debe especificar que desea sobreescribir (override) el constructor por defecto de la clase ancestro, TComponent. En la porción de implementación de la unidad, debe agregar el código asociado para el constructor, como en el ejemplo:

```
constructor TMult.Create(Aowner : TComponent);
begin
      inherited Create(AOwner);          {llamar al constructor de la clase padre}
      FVal1 := 1;                        {poner valores por defecto}
end;
```

Para realizar cualquier construcción específica del padre, debe primero llamar al procedimiento ***create*** heredado. En este caso, el único paso adicional fue colocar un valor por defecto.

Agregando un Método

Un método es más fácil de implementar que una propiedad. Para declarar un método, coloque un procedimiento o una función en la parte pública de la definición de la clase, y escriba la función o procedimiento asociado. Por ejemplo, agreguemos el método DoMult como sigue:

```
{...}
```

```
public
   procedure DoMult;   {Método para multiplicar}
{...}

procedure TMult.DoMult;
begin
      FRes := FVal1 * FVal2;
end;
```

Hasta aquí el componente de ejemplo es funcional, un usuario puede agregarlo a una forma, asignar valores para Val1 y Val2 ya sea durante el diseño o en la ejecución de su programa. Durante la ejecución, el método DoMult puede ser llamado para realizar la multiplicación.

Agregando un Evento

Un evento permite al usuario tener ejecutando código especializado cuando algo ocurre. Para el componente, por ejemplo, puede agregar un evento que sea lanzado cada vez que el valor de Val1 o Val2 sea mayor de 100 y se esté ejecutando DoMult. Puede modificar el código de tal forma que Res permanezca sin cambio cuando esto ocurra.

En Delphi, un evento es simplemente una propiedad especializada –una propiedad que es un apuntador a una función. Cualquier cosa que aplica a una propiedad también lo hace con las funciones.

Una pieza final de información que el componente debe proveer es cuándo llamar al manejador de eventos del usuario. Esto es simple. Cuando sea necesario lanzar un evento, consultar si el usuario ha definido un manejador para su evento. Si a sí fue, llamar al evento. Y finalmente agregar las declaraciones a la definición de la clase como sigue:

```
{...}
private
   FTooBig : TNotifyEvent;
   {...}
published
   property OnTooBig : TNotifyEvent read FTooBig write FTooBig;  {evento}
   {...}
end;
{...}
```

El tipo TnotifyEvent, que es usado para definir FTooBig y OnTooBig, es un tipo de apuntador a una función genérica que pasa un parámetro de tipo componente –usualmente Self. El paso final es modificar el procedimiento TMult.DoMult para que llame al manejador del evento si cualquiera de los dos números es muy grande. Antes de llamar al manejador del evento, se debe verificar que el evento está definido. Para hacer esto, se utiliza la función *assigned*, que regresa verdadero si un evento está definido para el manejador de eventos y falso si no.

```
{...}
```

```
procedure TMult.DoMult;
begin
     if (Val1 < 100) y (Val2 < 100) then
       FRes := FVal1 * FVal2
     else
       if assigned(FTooBig) then OnTooBig(Self);
end;

{...}
```

Capítulo 2

Solución de Sistemas de Ecuaciones Lineales usando la descomposición LU

2.1 La descomposición LU y sus aplicaciones

Supongamos que podemos escribir una matriz A como el producto de dos matrices,

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{2.1.1}$$

donde **L** es una *matriz triangular inferior* (tiene elementos sólo en la diagonal y hacia abajo) y **U** es una *matriz triangular superior* (con elementos en la diagonal y hacia arriba). Para el caso de una matriz de 4 x 4, por ejemplo, la ecuación 2.1.1 se vería así:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{2.1.2}$$

Podemos usar una descomposición como (2.1.1) para resolver la ecuación lineal

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \tag{2.1.3}$$

Resolviendo primero para el vector **y**, tal que

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{2.1.4}$$

y después resolviendo

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{2.1.5}$$

¿Cuál es la ventaja de separar una ecuación lineal en dos? La ventaja es que la solución de un conjunto de ecuaciones triangular es casi trivial. Esto es, la ecuación (2.1.4) puede resolverse con sustitución hacia adelante como sigue,

$$\begin{aligned} y_1 &= \frac{b_1}{\alpha_{11}} \\ y_i &= \frac{1}{\alpha_{ii}}\left[b_i - \sum_{j=1}^{i-1}\alpha_{ij}y_j\right] \qquad i = 2,3,\ldots,N \end{aligned} \tag{2.1.6}$$

mientras que (2.1.5) puede resolverse con sustitución hacia atrás,

$$\begin{aligned} x_N &= \frac{y_N}{\alpha_{11}} \\ x_i &= \frac{1}{\beta_{ii}}\left[y_i - \sum_{j=i+1}^{N}\beta_{ij}x_j\right] \qquad i = N-1, N-2,\ldots,1 \end{aligned} \tag{2.1.7}$$

Note que, una vez obtenida la descomposición LU de una matriz **A**, podemos resolver tantas ecuaciones como necesitemos, sólo cambiando los valores del vector **b**. Esta es una ventaja sobre otros métodos, como el de Gauss-Jordan que necesita operar nuevamente con todos los elementos de la matriz para resolver el sistema de ecuaciones con otro vector **b**.

2.2 Desarrollando la Descomposición LU

¿Cómo encontrar las matrices **L** y **U** a partir de una matriz **A** dada? Primero escribamos el i, j-ésimo componente de la ecuación (2.1.1) o (2.1.2). Ese componente es siempre la única suma que inicia con

$$\alpha_{i1}\beta_{1j} + \cdots = a_{ij}$$

El número de términos en la suma depende, sin embargo, en cual de las i o las j es el número más pequeño. De hecho existen tres casos,

$$i < j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{2.1.8}$$

$$i = j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{jj} = a_{ij} \tag{2.1.9}$$

$$i > j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ij}\beta_{jj} = a_{ij} \tag{2.1.10}$$

Estas ecuaciones involucran un total de N^2 ecuaciones para las $N^2 + N$ α's y β's incógnitas (la diagonal se representa dos veces). Dado que el número de incógnitas es más grande que el de

las ecuaciones, esto nos invita a especificar N de las incógnitas arbitrariamente y entonces intentar resolver las otras. A decir verdad, como veremos a continuación, es posible tomar

$$\alpha_{ii} \equiv 1 \qquad i = 1,\ldots,N$$

Un procedimiento sorprendente, es el *algoritmo de Crout*, el cual resuelve trivialmente el conjunto de las $N^2 + N$ ecuaciones para todas las α's y β's sólo reagrupando las ecuaciones en cierto orden. Este orden es el que sigue:

Sea $\alpha_{ii} = 1,\ i = 1,\ldots,N$

Para cada j = 1, 2, 3, ..., N realice estos dos procedimientos: Primero, para i = 1, 2, ..., j, use (2.1.8), (2.1.9) y (2.1.10) para resolver β_{ij},

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \qquad 2.1.12$$

(Cuando i = 1 el término de la sumatoria se toma como cero)

Segundo, para $i = j + 1,\ j + 2,\ ...,\ N$ use (2.1.10) para resolver α_{ij}

$$\alpha_{ij} = \frac{1}{\beta_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj}\right) \qquad 2.1.13$$

Debemos asegurar quese ejecutan ambos procedimientos antes de continuar con la j siguiente.

Revisando algunas iteraciones de los procedimientos anteriores, se puede ver que las α's y β's que aparecen en los lados derechos de las ecuaciones (2.1.12) y (2.1.13) ya habían sido determinadas para cuando se necesitaron. También se observa que cada a_{ij} es utilizada sólo una vez. Esto significa que las α's y β's correspondientes pueden almacenarse en el mismo lugar que la a usada ocupó: la descomposición puede hacerse "en el mismo lugar" si se desea. [La diagonal de elementos α_{ii} no se almacena]. En resumen, el método de Crout llena la matriz combinada de α's y β's,

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix}$$

por columnas de izquierda a derecha, y cada columna de arriba hacia abajo.

El pivoteo (es decir, la correcta elección del elemento pivote para la división en la ecuación 2.1.13) es absolutamente esencial para la estabilidad del método de Crout. Sólo puede implementarse eficientemente el pivoteo parcial (intercambio de renglones). A diferencia del método de Gauss que también intercambia columnas.

Escogemos el elemento más grande de la diagonal β (el pivote), luego se hacen todas las divisiones por ese elemento. Este es el método de Crout con pivoteo parcial. Esta implementación tiene una novedad adicional: inicialmente encuentra el elemento más grande de cada renglón, y subsecuentemente (cuando se busque el elemento pivote máximo) escala las comparaciones como si inicialmente hubiéramos escalado todas las ecuaciones para hacer su máximo coeficiente igual a la unidad; a esto se le conoce como ***pivoteo implícito***.

Enseguida se muestra la función para implementar la descomposición LU usando el método de Crout.

```
function TDescomposicionLU.Descomposicion_LU( mxA : Variant; n : Integer; Var indx :
Variant; Var d : Double ) : Variant;
{Dada una matriz a[1..n][1..n], esta rutina regresa una matriz de la misma dimensión
pero descompuesta en la forma LU de una permutación de si misma.
 a y n son entradas, indx[1..n] es un vector de salida que almacena las permutaciones
de renglones efectuadas por un pivoteo parcial; d es una salida del tipo ±1
dependiendo si el número de intercambios de renglones fue par o non, respectivamente.
Esta rutina es usada en combinación con SustitucionAtras_LU para resolver ecuaciones
lineales o para invertir una matriz. }

Const TINY = 1.0e-20;              {Un número pequeño}
Var i, imax, j, k : Integer;
    big, dum, sum, temp : Double ;
    vv : Variant;        {vv Almacena el escalamiento implícito de cada renglón}
    a  : Variant;
begin
     vv := varArrayCreate([1,n], varDouble );
     a := mxA;

     d := 1.0;           {Aún no hay intercambio de renglones}

     for i:=1 to n do begin {recorrer los renglones para obtener la información}
                            {de escalamiento implícito}
         big := 0.0;
         for j:=1 to n do begin
             temp := abs(a[i,j]);
             if temp > big then big := temp;
         end;

         {El elemento más grande no puede ser cero}
         if big = 0.0 Then
             raise EErrorLUDcmp.Create('Se detectó una Matriz Singular al tratar de'
                                       + 'hacer la descomposición LU');
         vv[i] := 1.0 / big;  {guardar el escalamiento}
     end;

     imax := 1;
     {Este es el ciclo sobre columnas del método de Crout, esta es la ecuación 2.1.12
      excepto por i = j}
     for j:=1 to n do begin
         for i:=1 to j-1 do begin
             sum := a[i,j];
             for k:=1 to i-1 do sum := sum - a[i,k] * a[k,j];
```

```
            a[i,j] := sum;
        end;
        big := 0.0; {Inicializar para encontrar el pivote más grande}
        for i:=j to n do begin
            sum := a[i,j];
            for k:=1 to j-1 do sum := sum - a[i,k] * a[k,j];
            a[i,j] := sum;
            dum := vv[i] * abs(sum);
            If dum >= big Then begin
               big := dum;
               imax := i;
            end;
        end;

        if j <> imax Then begin {¿Necesitamos cambiar los renglones?}
           for k:=1 to n do begin
               dum := a[imax,k];
               a[imax,k] := a[j,k];
               a[j,k] := dum;
           end;
           d := -d;                    { ... y cambiamos la paridad de d}
           vv[imax] := vv[j];          { también cambiamos el factor de escala}
        end;
        indx[j] := imax;
        if a[j,j] = 0.0 Then a[j,j] := TINY;
        {Si el elemento pivote es cero la matriz es singular (al menos
         para la precision del algoritmo). Para algunas aplicaciones en
         matrices singulares, es preferible sustituir TINY por cero}

         if j<>n then begin {Finalmente, dividir entre el elemento pivote}
            dum := 1.0 / a[j,j];
            for i:=j+1 to n do a[i,j] := a[i,j] * dum;
         end;
     end;

     result := a;
end;
```

A continuación la rutina para sustitución hacia atrás y hacia adelante, implementando las ecuaciones (2.1.6) y (2.1.7)

```
function TDescomposicionLU.SustitucionAtras_LU( a : Variant; n : Integer; indx, mxB
:variant ) : Variant;
{ Resuelve el conjunto de n ecuaciones lineales A·X = B. Aquí a[1..n,1..n] es una
entrada, no como la matriz A sino como su descomposición LU, determinada por la rutina
Descomposicion_LU. indx[1..n] es entrada como el vector permutado regresado por
Descomposicion_LU. mxB[1..n] es entrada como el vector del lado derecho B, y regresa
el vector solución X. a, n e indx no son modificados por esta rutina y pueden
reutilizarse para llamadas sucesivas con diferentes vectores mxB del lado derecho.
Esta rutina toma en cuenta la posibilidad de que mxB pueda comenzar con muchos ceros,
así que es eficiente para el uso de la inversión de matrices}

var i, ii, ip, j : Integer;
    sum : Double;
    b : Variant;
begin
     ii:=0;
     b := mxB;
```

```
      for i:=1 to n do begin
          ip := indx[i];
          sum := b[ip];
          b[ip] := b[i];
          if ii<> 0 then
             for j:=ii to i-1 do sum := sum - a[i,j] * b[j]
          else if sum <> 0.0 then ii:=i;
          b[i] := sum;
      end;

      for i:=n downto 1 do begin
          sum := b[i];
          for j:= i+1 to n do sum := sum - a[i,j] * b[j];
          b[i] := sum / a[i,i];
      end;
      result := b;
end;
```

2.3 Mejoramiento Iterativo de una Solución de Ecuaciones Lineales

Obviamente no es fácil obtener una precisión muy grande para la solución de un conjunto de ecuaciones lineales que la precisión del punto flotante de la máquina. Desgraciadamente, para conjuntos muy grandes de ecuaciones lineales, no es siempre fácil obtener precisión igual, o cercano, al límite de la computadora. En métodos directos para encontrar la solución, los errores de redondeo se acumulan, y se magnifican al grado de hacer la matriz casi singular.

Si esto sucede, hay un truco para restaurar la precisión completa de la máquina, llamado *mejoramiento iterativo* de la solución. La teoría es simple: suponga que un vector **x** es la solución exacta de un conjunto lineal de ecuaciones

$$A \cdot x = b \qquad 2.3.1$$

Sin embargo, se desconoce **x**. Sólo conocemos una aproximación a la solución $x + \delta x$, donde δx es un error desconocido. Cuando es multiplicado por la matriz **A**, la aproximación da como resultado un producto significativamente discrepante del vector solución deseado **b**,

$$A \cdot (x + \delta x) = b + \delta b \qquad 2.3.2$$

restando (2.3.1) de (2.3.2) se obtiene

$$A \cdot \delta x = \delta b \qquad 2.3.3$$

Pero (2.3.2) puede resolverse, trivialmente, por δb. Sustituyendo esto en (2.3.3) queda

$$A \cdot \delta x = A \cdot (x + \delta x) - b \qquad 2.3.4$$

En esta ecuación, todo el lado derecho se conoce, dado que $x+\delta x$ es la aproximación inicial que se desea mejorar. Finalmente, solo necesitamos resolver (2.3.4) para el error δx, luego restarlo de la primera aproximación para así obtener una solución mejorada.

Un beneficio adicional ocurre si obtuvimos la solución original por descomposición LU. En ese caso tenemos A descompuesta en su forma LU, y lo único que se necesita para resolver (2.3.4) es calcular el lado derecho y hacer sustitución hacia atrás:

El siguiente procedimiento, es una implementación del mejoramiento iterativo de una solución.

```
procedure TDescomposicionLU.MejorarSolucion( a, alud : Variant; n : Integer; idx, b
:variant; Var x : Variant );
{ Mejora un vector solción x[1..n] de un conjunto de ecuaciones lineales A·X = B. La
matriz a[1..n][1..n], y los vectores b[1..n] y x[1..n] son entradas, así como la
dimensión n.
  También es entrada alud[1..n][1..n], la descomposición LU de A del resultado de
Descomposicion_LU, y el vector indx[1..n] también regresado por esa rutina. Como
salida, sólo x[1..n] es modificado, como un conjunto mejorado de valores.}

var
   j,i : Integer;
   sdp : Double;
   r   : Variant;
begin
     r := varArrayCreate([1,n], varDouble );
     for i:=1 to n do begin  { Calcular el lado derecho, acumulando }
         sdp := -b[i];        { el residuo en doble precisión.}
         for j:=1 to n do sdp := sdp + a[i,j] * x[j];
         r[i] := sdp;
     end;

     r := SustitucionAtras_LU( alud, n, idx, r ); {Resolver para el término error }
     for i:=1 to n do x[i] := x[i] - r[i];      {y restarlo de la solución anterior }
end;
```

Se puede llamar a MejorarSolucion varias veces si es necesario. A menos que se comience lejos de la solución real, una llamada generalmente es suficiente; pero una segunda llamada verifica que la convergencia pueda reasegurarse.

2.4 Listado completo del componente para resolver sistemas de ecuaciones usando descomposición LU

Propiedades

A, LU, IDX, B, X : Son las matrices utilizadas por el método

D : Es el valor utilizado para calcular el determinante de una matriz LU

MatrizA, MatrizLU, VectorB, VectorX : Son del tipo TstringGrid que es un componente estándar de Delphi útil para leer y escribir valores en forma de matriz

AcolIni, ARenIni : Indican la posición donde comienzan los valores para la matriz A a partir del StringGrid de la MatrizA

BcolIni, BRenIni : Indican la posición donde comienzan los valores para el vector B a partir del StringGrid del Vector B

XcolIni, XRenIni : Indican la posición donde comienzan los valores para el vector X a partir del StringGrid del Vector X

LUcolIni, LURenIni : Indican la posición donde comienzan los valores para la matriz LU a partir del StringGrid de la MatrizLU

N : Es el tamaño de la matriz A(n,n) y LU(n,n), así como de los vectores B(n) y X(n)

Decimales : Indica la precisión con que se presentan los datos en los resultados, note que la precisión del cálculo no se altera por esta propiedad, es sólo la salida.

Métodos

Descomponer : Basado en las matrices de forma StringGrid, regresa la descomposición LU de la MatrizA en la MatrizLU

ResolverSistema: Basado en las matrices de forma StringGrid, resuelve el sistema de ecuaciones dado por MatrizA * VectorX = VectorB, dejando el resultado en el VectorX

SolucionMejorada: Incrementa la precisión de una solución dada para un sistema de ecuaciones

Listado

```
unit DescomposicionLU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Grids;

type
  EErrorLUDcmp = class(Exception);
  TDescomposicionLU = class(TComponent)
```

```
  private
    FMatA    : TStringGrid;   {Matriz A original, a operar}
    FMatLU   : TStringGrid;  {Matriz de resultado, contiene la descomposición LU de A}
    FVecB    : TStringGrid;
    FVecX    : TStringGrid;   {Vector resultado X del sistema de ecuaciones}
    FAColIni : Integer;
    FARenIni : Integer;
    FLUColIni : Integer;
    FLURenIni : Integer;
    FBColIni : Integer;
    FBRenIni : Integer;
    FXColIni : Integer;
    FXRenIni : Integer;
    FN       : Integer;
    FDecimales : Integer;  {Indica el formato de salida para llenar el StringGrid}
    FFormato   : String;

    FA, FLU, FIDX, FB, FX : Variant;
    FD : Double;
  protected
    function Descomposicion_LU(  mxa : Variant; n : Integer; Var indx : Variant;
Var d : Double ) : Variant ;
    function SustitucionAtras_LU( a : Variant; n : Integer; indx, mxB :variant )
: Variant;
    procedure MejorarSolucion(  a, alud : Variant; n : Integer; idx, b :variant;
Var x : Variant );
    procedure SetFormato(dec : Integer);
  public
    constructor Create(AOwner : TComponent); override;
    property A : Variant read FA;
    property LU : Variant read FLU;
    property IDX : Variant read FIDX;
    property B : Variant read FB;
    property X : Variant read FX;
    property D : Double read FD;
  published
    property MatrizA  : TStringGrid  read FMatA write FMatA;
    property MatrizLU : TStringGrid  read FMatLU write FMatLU;
    property VectorB  : TStringGrid  read FVecB write FVecB;
    property VectorX  : TStringGrid  read FVecX write FVecX;
    property AColIni : Integer read FAColIni write FAColIni default 1;
    property ARenIni : Integer read FARenIni write FARenIni default 1;
    property BColIni : Integer read FBColIni write FBColIni default 1;
    property BRenIni : Integer read FBRenIni write FBRenIni default 1;
    property XColIni : Integer read FXColIni write FXColIni default 1;
    property XRenIni : Integer read FXRenIni write FXRenIni default 1;
    property LUColIni : Integer read FLUColIni write FLUColIni default 1;
    property LURenIni : Integer read FLURenIni write FLURenIni default 1;
    property N : Integer read FN write FN default 4;
    property Decimales : Integer read FDecimales write SetFormato default 3 ;
    procedure Descomponer;
    procedure ResolverSistema;
    procedure SolucionMejorada;
  end;

procedure Register;

implementation

Uses MACTools;

constructor TDescomposicionLU.Create(AOwner : TComponent);
begin
```

```
    inherited Create(AOwner);
    FN := 4;
    FAColIni := 1; FARenIni := 1;
    FBColIni := 1; FBRenIni := 1;
    FXColIni := 1; FXRenIni := 1;
    FLUColIni := 1; FLURenIni := 1;
    FDecimales := 3;     FFormato := '%.3f';
end;

procedure TDescomposicionLU.SetFormato(Dec : Integer);
begin
     FDecimales := Dec;
     FFormato := '%.' + IntToStr(Dec) + 'f' ;
end;

function TDescomposicionLU.Descomposicion_LU( mxA : Variant; n : Integer; Var indx :
Variant; Var d : Double ) : Variant;
{Dada una matriz a[1..n][1..n], esta rutina regresa una matriz de la misma
 dimensión pero descompuesta en la forma LU de una permutación de si misma.
 a y n son entradas, indx[1..n] es un vector de salida que almacena las
 permutaciones de renglones efectuadas por un pivoteo parcial; d es una
 salida del tipo ±1 dependiendo si el número de intercambios de renglones
 fue par o non, respectivamente. Esta rutina es usada en combinación con
 SustitucionAtras_LU para resolver ecuaciones lineales o para invertir
 una matriz.
}
Const TINY = 1.0e-20;              {Un número pequeño}
Var i, imax, j, k : Integer;
    big, dum, sum, temp : Double ;
    vv : Variant;        {vv Almacena el escalamiento implícito de cada renglón}
    a  : Variant;
begin
     vv := varArrayCreate([1,n], varDouble );
     a := mxA;

     d := 1.0;           {Aún no hay intercambio de renglones}

     for i:=1 to n do begin {recorrer los renglones para obtener}
                            {la información de escalamiento implícito}
         big := 0.0;
         for j:=1 to n do begin
             temp := abs(a[i,j]);
             if temp > big then big := temp;
         end;

         {El elemento más grande no puede ser cero}
         if big = 0.0 Then
             raise EErrorLUDcmp.Create('Se detectó una Matriz Singular al tratar de'
                                       +'hacer la descomposición LU');
         vv[i] := 1.0 / big;  {guardar el escalamiento}
     end;

     imax := 1;
     {Este es el ciclo sobre columnas del método de Crout, esta es la ecuación 2.1.12
      excepto por i = j}
     for j:=1 to n do begin
         for i:=1 to j-1 do begin
             sum := a[i,j];
             for k:=1 to i-1 do sum := sum - a[i,k] * a[k,j];
             a[i,j] := sum;
         end;
         big := 0.0; {Inicializar para encontrar el pivote más grande}
         for i:=j to n do begin
```

```
            sum := a[i,j];
            for k:=1 to j-1 do sum := sum - a[i,k] * a[k,j];
            a[i,j] := sum;
            dum := vv[i] * abs(sum);
            If dum >= big Then begin
               big := dum;
               imax := i;
            end;
         end;

         if j <> imax Then begin {¿Necesitamos cambiar los renglones?}
            for k:=1 to n do begin
               dum := a[imax,k];
               a[imax,k] := a[j,k];
               a[j,k] := dum;
            end;
            d := -d;                    { ... y cambiamos la paridad de d}
            vv[imax] := vv[j];          { también cambiamos el factor de escala}
         end;
         indx[j] := imax;
         if a[j,j] = 0.0 Then a[j,j] := TINY;
         {Si el elemento pivote es cero la matriz es singular (al menos
          para la precisión del algoritmo). Para algunas aplicaciones en
          matrices singulares, es preferible sustituir TINY por cero}

          if j<>n then begin {Finalmente, dividir entre el elemento pivote}
             dum := 1.0 / a[j,j];
             for i:=j+1 to n do a[i,j] := a[i,j] * dum;
          end;
      end;

      result := a;
end;

function TDescomposicionLU.SustitucionAtras_LU( a : Variant; n : Integer; indx, mxB
:variant ) : Variant;
{ Resuelve el conjunto de n ecuaciones lineales A·X = B. Aquí a[1..n,1..n] es
  entrada, no como la matriz A sino como su descomposición LU, determinada por
  la rutina Descomposicion_LU. indx[1..n] es entrada como el vector permutado
  regresado por Descomposicion_LU. mxB[1..n] es entrada como el vector del lado
  derecho B, y regresa el vector solución X. a, n e indx no son modificados por
  esta rutina y pueden reutilizarse para llamadas sucesivas con diferentes vectores
  mxB del lado derecho. Esta rutina toma en cuenta la posibilidad de que mxB pueda
  comenzar con muchos ceros, así que es eficiente para el uso de la inversión
  de matrices
}
var i, ii, ip, j : Integer;
    sum : Double;
    b : Variant;
begin
      ii:=0;
      b := mxB;

      for i:=1 to n do begin
          ip := indx[i];
          sum := b[ip];
          b[ip] := b[i];
          if ii<> 0 then
             for j:=ii to i-1 do sum := sum - a[i,j] * b[j]
          else if sum <> 0.0 then ii:=i;
          b[i] := sum;
      end;
```

```
      for i:=n downto 1 do begin
          sum := b[i];
          for j:= i+1 to n do sum := sum - a[i,j] * b[j];
          b[i] := sum / a[i,i];
      end;
      result := b;
end;


procedure TDescomposicionLU.MejorarSolucion( a, alud : Variant; n : Integer; idx, b
:variant; Var x : Variant );
{ Mejora un vector solción x[1..n] de un conjunto de ecuaciones lineales A·X = B. La
matriz a[1..n][1..n], y los vectores b[1..n] y x[1..n] son entradas, así como la
dimensión n.
  También es entrada alud[1..n][1..n], la descomposición LU de A del resultado de
Descomposicion_LU, y el vector indx[1..n] también regresado por esa rutina. Como
salida, sólo x[1..n] es modificado, como un conjunto mejorado de valores.}
var
   j,i : Integer;
   sdp : Double;
   r   : Variant;
begin
      r := varArrayCreate([1,n], varDouble );
      for i:=1 to n do begin  { Calcular el lado derecho, acumulando }
          sdp := -b[i];        { el residuo en doble precisión.}
          for j:=1 to n do sdp := sdp + a[i,j] * x[j];
          r[i] := sdp;
      end;

      r:= SustitucionAtras_LU( alud, n, idx, r );{Resolver para el término error }
      for i:=1 to n do x[i] := x[i] - r[i];  {y restarlo de la solución anterior }
end;


procedure TDescomposicionLU.Descomponer;
{ Basado en las matrices de forma StringGrid, regresa la descomposición LU
  de la MatrizA en la MatrizLU }
begin
     Flu := varArrayCreate([1,Fn, 1,Fn], varDouble );
     Fidx := varArrayCreate([1,Fn], varDouble );

     Fa := MACTools.SGaMatriz( FMatA, FN, FN, FAColIni, FARenIni);
     Flu := Descomposicion_LU( Fa, Fn, Fidx, Fd);
     MACTools.MatrizASG( Flu, FMatLU, FN, FN, FLUColIni, FLURenIni, FFormato);
end;


procedure TDescomposicionLU.ResolverSistema;
{ Basado en las matrices de forma StringGrid, resuelve el sistema de ecuaciones dado
por MatrizA * VectorX = VectorB, dejando el resultado en el VectorX }
begin
   if VarType(FLU) in [varEmpty, varNull] Then
      raise EErrorLUDcmp.Create('Debe hacer la descomposición LU antes de poder
resolver el sistema');

   Fb := varArrayCreate([1,Fn], varDouble );
   Fb := MACTools.SGaVector( FVecB, FN, FBColIni, FBRenIni);

  {Se sustituye hacia atras usando la Matriz LU no A}
   Fx := SustitucionAtras_LU( Flu, Fn, Fidx, Fb );
   MACTools.VectorASG( Fx, FVecX, FN, FXColIni, FXRenIni, FFormato);

end;
```

```
procedure TDescomposicionLU.SolucionMejorada;
begin
   if VarType(FX) in [varEmpty, varNull] Then
      raise EErrorLUDcmp.Create('Primero debe resolver el sistema para poder mejorar
la solición');

   MejorarSolucion( Fa, Flu, Fn, Fidx, Fb, Fx );
   MACTools.VectorASG( Fx, FVecX, FN, FXColIni, FXRenIni, FFormato);
end;

procedure Register;
begin
  RegisterComponents('MAC', [TDescomposicionLU]);
end;

end.
```

Capítulo 3

Polinomios e Interpolación de Funciones

3.1 Operaciones con Polinomios

Un polinomio de grado N es representado numéricamente como un arreglo de coeficientes, c[j] con j=0,...,N. Siempre se toma c[0] para ser el término constante en el polinomio, c[N] el coeficiente de x^N; pero pueden existir otras convenciones. Hay dos clases de manipulaciones que pueden hacerse con un polinomio: manipulaciones numéricas (como la evaluación), donde se da el valor numérico de su argumento, o manipulaciones algebraicas, donde se desea transformar el arreglo de coeficientes de alguna forma sin escoger un argumento en particular.

Se asume nunca evaluaríamos un polinomio de esta forma:

```
p := c[0] + c[1]*x + c[2]*x*x + c[3]*x*x*x + c[4]*x*x*x*x;
```

o (peor aún),

```
p := c[0] + c[1]*x + c[2]*Power(x,2.0) + c[3]* Power (x,3.0)
      + c[4]* Power (x,4.0);
```

Es mejor escribir

```
p := c[n]
for j =n-1 downto 0 do p := p * x + c[j];
```

Otro truco útil para evaluar un polinomio P(x) y su derivada dP(x)/dx simultáneamente:

```
p := c[n];
dp := 0;
for j:= n-1 downto 0 do begin
      dp := dp*x + p;
      p := p * x + c[j];
```

```
end;
```

El truco anterior, que es básicamente división sintética [Stoer 1980] [Numerical 1992], generaliza a la evaluación del polinomio y nd de sus derivadas simultáneamente:

```
procedure TEvaluaPolinomio.evalpol(c : Variant; nc : integer; x : Double; Var pd :
Variant; nd : integer);
{ Dados los nc+1 coeficientes de un polinomio de grado nc cómo el arreglo c[0..nc] con
c[0] el término constante, y dado un valor de x, y dado un valor nd>1, esta rutina
regresa el polinomio evaluado en x como pd[0] y nd derivadas como pd[1..nd].}
var
   nnd,j,i : Integer;
   cnst : Double;
begin
  cnst:=1.0;
  pd[0]:=c[nc];
  for j:=1 to nd do pd[j]:=0.0;
  for i:=nc-1 downto 0 do begin
      if ( nd < (nc-1) ) then nnd := nd
      else                    nnd := nc-i;
      for j:=nnd downto 1 do
          pd[j]:=pd[j]*x+pd[j-1];
      pd[0]:=pd[0]*x+c[i];
  end;
  for i:=2 to nd do begin  { Después de la primer derivada, }
                           { aparecen constantes factoriales.}
      cnst := cnst * i;
      pd[i] := pd[i] * cnst;
  end;
end;
```

Toca el turno a manipulaciones algebraicas. Para multiplicar un polinomio de grado n-1 (un arreglo de [0..n-1]) por un monomio (x – a) se usa un código como el siguiente,

```
c[n] := c[n-1];
for j:=n-1 downto 1 do c[j] := c[j-1] - c[j]*a;
c[0] := c[0] * (-a);
```

De igual forma, se puede dividir un polinomio de grado n por un monomio (x - a) usando división sintética de nuevo,

```
rem := c[n];
c[n] := 0.0;
for i:=n-1 downto 0 do begin
       swap := c[i]
       c[i] := rem;
       rem := swap+rem*a;
end;
```

que deja un nuevo arreglo para el polinomio y un residuo numérico rem.

La multiplicación de dos polinomios generales involucra directamente la suma de los productos, cada uno involucrando un coeficiente de cada polinomio. La división de dos polinomios generales, mientras que puede hacerse de la forma más lenta usando lápiz y papel, es susceptible de hacerlo mejor. Observe la siguiente rutina basada en el algoritmo de [Numerical, 1992].

```
procedure TDividirPolinomio.poldiv( u : Variant; n : Integer; v : Variant; nv :
Integer; Var q, r : Variant );
{ Dados los n+1 coeficientes de un polinomio de grado n en u[0..n], y los nv+1
coeficientes de otro polinomio de grado nv en v[0..nv], dividir el polinomio u por el
polinomio v ("u"/"v") da como resultado un polinomio cociente cuyos coeficientes se
regresan en q[0..n], y un polinomio residuo cuyos coeficientes se regresan en r[0..n].
Los elementos r[nv..n] y q[n-nv+1..n] se regresan como ceros.}
var
   k,j : Integer;
begin
  if v[nv] = 0 then
     raise EErrorOperaPoli.Create('El coeficiente para el grado del denominador no
puede ser cero');

  for j:=0 to n do begin
      r[j]:=u[j];
      q[j]:=0.0;
  end;
  for k:=n-nv downto 0 do begin
      q[k]:=r[nv+k]/v[nv];
      for j:=nv+k-1 downto k do r[j] := r[j] - q[k]*v[j-k];
  end;
  for j:=nv to n do r[j]:=0.0;
end;
```

3.2 Listado completo de los componentes para operar polinomios

Evaluar Polinomios

Propiedades

Polinomio, Derivadas : Son del tipo TstringGrid que es un componente estándar de Delphi útil para leer y escribir valores en forma de matriz.

PolColIni, PolRenIni : Indican la posición donde comienzan los valores para el vector de coeficientes del polinomio a partir del StringGrid.

DerColIni, DerRenIni : Indican la posición donde comienzan los valores para el vector de coeficientes donde se almacenarán los valores de las derivadas del polinomio.

Grado : Indica el número de coeficientes más uno, que forman al polinomio.

Decimales : Indica la precisión con que se presentan los datos en los resultados, note que la precisión del cálculo no se altera por esta propiedad, es sólo la salida.

Métodos

Evaluar : Dado un valor de x este método regresa el valor de P(x).

EvaluarDerivar : Dado un valor de x este método regresa el valor de P(x) y de las N derivadas (N es el grado del polinomio) evaluadas en ese punto.

Dividir Polinomios

Propiedades

Numerador, Denominador, Resultado, Residuo: Son del tipo TStringGrid que es un componente estándar de Delphi útil para leer y escribir valores en forma de matriz. Numerador y Denominador son las entradas, y Resultado y Residuo las salidas.

NumColIni, NumRenIni, DenColIni, DenRenIni, RespColIni, RespRenIni, ResColIni, ResRenIni : Indican la posición donde comienzan los valores para los vectores de coeficientes de los polinomios de entrada y salida.

GradoNumerador, Grado Denominador : Indican el número de coeficientes más uno, que forman a los polinomios numerador y denominador.

Decimales : Indica la precisión con que se presentan los datos en los resultados, note que la precisión del cálculo no se altera por esta propiedad, es sólo la salida.

Métodos

Dividir : Dados los vectores de coeficientes Numerador y Denominador, este procedimiento regresa la división de los mismos en los vectores Resultado y Residuo.

Listado

```
unit OperarPolinomios;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;

type
  EErrorOperaPoli = class(Exception);
```

```
// ----------------------------------------------------------------------------
// Iniciando la definición del Componente EvaluaPolinomio

  TEvaluaPolinomio = class(TComponent)
  private
    FVecPoli:TStringGrid;{Vector donde se almacenarán los valores del polinomio }
    FVecDeri:TStringGrid;{Vector donde se almacenarán la respuesta de las derivadas }

    FPoliColIni : Integer;   {Indices para ubicar los datos a operar}
    FPoliRenIni : Integer;
    FN          : Integer;   {Grado del polinomio}
    FDeriColIni : Integer;   {Indices para ubicar los datos resultado }
    FDeriRenIni : Integer;

    FDecimales : Integer;  {Indica el formato de salida para llenar el StringGrid}
    FFormato   : String;
    procedure evalpol(c : Variant; nc : integer; x : Double; Var pd : Variant;
                      nd  : integer);
  protected
    procedure SetFormato(dec : Integer);
  public
    constructor Create(AOwner : TComponent); override;
    procedure EvaluarDerivar( x : Double);
    function  Evaluar(x : double) : Double;
  published
    property Polinomio : TStringGrid  read FVecPoli write FVecPoli;
    property Derivadas : TStringGrid  read FVecDeri write FVecDeri;
    property PoliColIni : Integer read FPoliColIni write FPoliColIni default 1;
    property PoliRenIni : Integer read FPoliRenIni write FPoliRenIni default 1;
    property DerColIni : Integer read FDeriColIni write FDeriColIni default 1;
    property DerRenIni : Integer read FDeriRenIni write FDeriRenIni default 1;
    property Decimales : Integer read FDecimales write SetFormato default 3 ;
    property Grado     : Integer read FN write FN default 4;
  end;

// ----------------------------------------------------------------------------
// Iniciando la definición del Componente DividirPolinomio

  TDividirPolinomio = class(TComponent)
  private
    FVecU:TStringGrid; {Vector donde se almacenarán los valores del polinomio U }
    FVecV:TStringGrid; {Vector donde se almacenarán los valores del polinomio V }

    FVecQ:TStringGrid; {Vector donde se almacenarán los valores del polinomio Q }
    FVecR:TStringGrid; {Vector donde se almacenarán los valores del polinomio R }

    FUColIni : Integer;   {Indices para ubicar los datos a operar}
    FURenIni : Integer;
    FVColIni : Integer;
    FVRenIni : Integer;
    FQColIni : Integer;
    FQRenIni : Integer;
    FRColIni : Integer;
    FRRenIni : Integer;

    Fnu : Integer;   {Grado del polinomio U numerador}
    Fnv : Integer;   {Grado del polinomio U denominador}

    FDecimales : Integer;  {Indica el formato de salida para llenar el StringGrid}
    FFormato   : String;
    procedure poldiv( u : Variant; n : Integer; v : Variant; nv : Integer; Var q,
                      r : Variant );  protected
```

```
    procedure SetFormato(dec : Integer);
  public
    constructor Create(AOwner : TComponent); override;
    procedure Dividir;
  published
    property Numerador   : TStringGrid  read FVecU write FVecU;
    property Denominador : TStringGrid  read FVecV write FVecV;
    property Resultado : TStringGrid  read FVecQ write FVecQ;
    property Residuo   : TStringGrid  read FVecR write FVecR;

    property NumColIni : Integer read FUColIni write FUColIni default 1;
    property NumRenIni : Integer read FURenIni write FURenIni default 1;
    property DenColIni : Integer read FVColIni write FVColIni default 1;
    property DenRenIni : Integer read FVRenIni write FVRenIni default 1;

    property RespColIni : Integer read FQColIni write FQColIni default 1;
    property RespRenIni : Integer read FQRenIni write FQRenIni default 1;
    property ResiduoColIni : Integer read FRColIni write FRColIni default 1;
    property ResiduoRenIni : Integer read FRRenIni write FRRenIni default 1;

    property Decimales : Integer read FDecimales write SetFormato default 3 ;
    property GradoNumerador   : Integer read Fnu write Fnu default 4;
    property GradoDenominador : Integer read Fnv write Fnv default 4;
  end;


procedure Register;

implementation

Uses MACTools;

// ------------------------------------------------------------------------
// Iniciando la definición del Componente EvaluaPolinomio

constructor TEvaluaPolinomio.Create(AOwner : TComponent);
begin
   inherited Create(AOwner);
   FPoliColIni := 1; FPoliRenIni := 1;
   FDeriColIni := 1; FDeriRenIni := 1;
   FDecimales := 3;    FFormato := '%.3f';
   FN := 4;
end;

procedure TEvaluaPolinomio.SetFormato(Dec : Integer);
begin
     FDecimales := Dec;
     FFormato := '%.' + IntToStr(Dec) + 'f' ;
end;

procedure TEvaluaPolinomio.evalpol(c : Variant; nc : integer; x : Double; Var pd :
Variant; nd : integer);
{ Dados los nc+1 coeficientes de un polinomio de grado nc como el arreglo c[0..nc] con
c[0] el término constante, y dado un valor de x, y dado un valor nd>1, esta rutina
regresa el polinomio evaluado en x como pd[0] y nd derivadas como pd[1..nd].}
var
   nnd,j,i : Integer;
   cnst : Double;
begin
  cnst:=1.0;
  pd[0]:=c[nc];
  for j:=1 to nd do pd[j]:=0.0;
  for i:=nc-1 downto 0 do begin
```

```
      if ( nd < (nc-1) ) then nnd := nd
      else                    nnd := nc-i;
      for j:=nnd downto 1 do
          pd[j]:=pd[j]*x+pd[j-1];
      pd[0]:=pd[0]*x+c[i];
  end;
  for i:=2 to nd do begin   { Después de la primer derivada, aparecen constantes
factoriales.}
      cnst := cnst * i;
      pd[i] := pd[i] * cnst;
  end;
end;


function TEvaluaPolinomio.Evaluar(x : double) : Double;
var c, pd : Variant;
begin
   if Not assigned(FVecPoli) Then
      raise  EErrorOperaPoli.Create('Debe  asignar  primero  un  vector  para  el
polinomio');

   c := MACTools.SGaVectorHorCero( FVecPoli, FN, FPoliColIni, FPoliRenIni);
   pd := varArrayCreate([0,Fn], varDouble );

   evalpol( c, FN, x, pd, 1 );

   result := pd[0];
end;


procedure TEvaluaPolinomio.EvaluarDerivar( x : Double);
var c, pd : Variant;
begin
   if Not assigned(FVecPoli) Then
      raise  EErrorOperaPoli.Create('Debe  asignar  primero  un  vector  para  el
polinomio');
   if Not assigned(FVecDeri) Then
      raise  EErrorOperaPoli.Create('Debe  asignar  primero  un  vector  para  las
Derivadas');

   {Sólo para revisar que el SG tiene suficiente espacio para almacenar la respuesta}
   pd := MACTools.SGaVectorHorCero( FVecDeri, FN, FDeriColIni, FDeriRenIni);
   pd := varArrayCreate([0,Fn], varDouble );

   c := MACTools.SGaVectorHorCero( FVecPoli, FN, FPoliColIni, FPoliRenIni);
   evalpol( c, FN, x, pd, FN );

   MACTools.VectorASGHorCero(pd, FVecDeri, Fn, FDeriColIni, FDeriRenIni , FFormato);
end;


// ----------------------------------------------------------------------------
// Iniciando la definición del Componente DividirPolinomio

constructor TDividirPolinomio.Create(AOwner : TComponent);
begin
   inherited Create(AOwner);
   FUColIni := 1; FURenIni := 1;
   FVColIni := 1; FVRenIni := 1;
   FQColIni := 1; FQRenIni := 1;
   FRColIni := 1; FRRenIni := 1;
   FDecimales := 3;    FFormato := '%.3f';
   Fnu := 4; Fnv := 4;
```

```
end;

procedure TDividirPolinomio.SetFormato(Dec : Integer);
begin
     FDecimales := Dec;
     FFormato := '%.' + IntToStr(Dec) + 'f' ;
end;

procedure TDividirPolinomio.poldiv( u : Variant; n : Integer; v : Variant; nv :
Integer; Var q, r : Variant );
{ Dados los n+1 coeficientes de un polinomio de grado n en u[0..n], y los nv+1
coeficientes de otro polinomio de grado nv en v[0..nv], dividir el polinomio u por el
polinomio v ("u"/"v") da como resultado un polinomio cociente cuyos coeficientes se
regresan en q[0..n], y un polinomio residuo cuyos coeficientes se regresan en r[0..n].
Los elementos r[nv..n] y q[n-nv+1..n] se regresan como ceros.}
var
   k,j : Integer;
begin
  if v[nv] = 0 then
     raise EErrorOperaPoli.Create('El coeficiente para el grado del denominador no
puede ser cero');

  for j:=0 to n do begin
      r[j]:=u[j];
      q[j]:=0.0;
  end;
  for k:=n-nv downto 0 do begin
      q[k]:=r[nv+k]/v[nv];
      for j:=nv+k-1 downto k do r[j] := r[j] - q[k]*v[j-k];
  end;
  for j:=nv to n do r[j]:=0.0;
end;

procedure TDividirPolinomio.Dividir;
var u, v, q, r : Variant;
begin
   if (Not assigned(FVecU)) or (Not assigned(FVecV)) or
      (Not assigned(FVecQ)) or (Not assigned(FVecR))
   Then raise EErrorOperaPoli.Create('Debe especificar los cuatro vectores para poder
realizar la división');

   u := MACTools.SGaVectorHorCero( FVecU, FNU, FUColIni, FURenIni);
   v := MACTools.SGaVectorHorCero( FVecV, FNV, FVColIni, FVRenIni);
   q := varArrayCreate([0,Fnu], varDouble );
   r := varArrayCreate([0,Fnu], varDouble );

   poldiv( u, Fnu, v, Fnv, q, r );

   MACTools.VectorASGHorCero(q, FVecQ, Fnu, FQColIni, FQRenIni , FFormato);
   MACTools.VectorASGHorCero(r, FVecR, Fnu, FRColIni, FRRenIni , FFormato);
end;

procedure Register;
begin
  RegisterComponents('MAC', [TEvaluaPolinomio, TDividirPolinomio]);
end;

end.
```

3.3 Interpolación

A veces sabemos el valor de una función f (x) en un conjunto de puntos $x_1, x_2, \ldots, x_N$ (con $x_1 < \ldots < x_N$), pero no tenemos una expresión analítica para f (x) que nos permita calcular su valor en un punto arbitrario. Por ejemplo, los valores f(x_i) pueden resultar de algunas mediciones físicas o de un cálculo numérico extenso que no se puede poner en una función simple. A menudo las x_i's se encuentran separadas entre sí en la misma cantidad, pero no necesariamente.

La tarea está ahora en estimar f(x) para una valor x arbitrario para, en algunos casos, dibujar una curva en el valor de x_i (y quizás más allá). Si el valor de x deseado se encuentra entre el más grande y más pequeño de las x_i, el problema se llama interpolación; si el valor de x está fuera de ese rango, se llama extrapolación, que es considerablemente más difícil de calcular (como muchos analistas pueden confirmar).

Los esquemas de interpolación y extrapolación deben modelar la función, entre o más allá de los puntos conocidos, por alguna forma funcional plausible. La forma debe ser suficientemente general para poder aproximar muchas clases de funciones que pueden surgir en la práctica. El uso más común entre las formas funcionales son los polinomios. Funciones racionales (cocientes de polinomios) también llegan a ser sumamente útiles. Funciones trigonométricas, senos y cosenos, dan lugar a interpolaciones trigonométricas y métodos de Fourier relacionados.

Hay una extensa literatura matemática consagrada a teoremas sobre las clases de funciones que pueden ser aproximadas por funciones de interpolación. Estos teoremas son casi completamente inútiles para trabajar todos los días: ¡Si sabemos bastante sobre nuestra función para aplicar un teorema de cualquier potencia, no estamos normalmente en el estado lastimoso de tener que interpolar en una mesa sus valores!. La interpolación se relaciona, pero de distinta forma, con la aproximación de la función. Esa tarea consiste en encontrar una función aproximada (pero fácilmente calculable) para usarla en lugar de una más complicada. En el caso de la interpolación, se da la función f a puntos no de su propia elección. Para el caso de la aproximación de la función, se permite calcular la función f en los puntos deseados con el propósito de desarrollar su aproximación.

Uno puede encontrar fácilmente funciones patológicas que hacen una burla de cualquier esquema de interpolación. Considere, por ejemplo, la función

$$f(x) = 3x^2 + \frac{1}{\pi^4}\ln\left[(\pi - x)^2\right] + 1$$

que se comporta bastante bien excepto en $x = \pi$. Cualquier interpolación basada en los valores x=3.13, 3.14, 3.15, 3.16, seguramente obtendrá una respuesta errónea para el valor x = 3.1416, aunque la gráfica de estos cinco puntos se vea realmente suave. (Inténtelo en su calculadora.)

Dado que las patologías como esta pueden ocurrir en cualquier lugar, es altamente deseable que una rutina de extrapolación e interpolación puedan proveernos de una estimación de su error. La interpolación asume siempre algún grado de suavidad, pero dentro de ésta, cualquier desviación de la suavidad puede detectarse.

Conceptualmente, el proceso de interpolación tiene dos etapas: *(i)* calcular una función para los puntos dados. *(ii)* Evaluar la función de interpolación en un punto x dado.

Sin embargo, este método de dos etapas no es generalmente el mejor modo de proceder en la práctica. Típicamente es computacionalmente menos eficiente, y más susceptible a errores de redondeo, que métodos que construyen una función estimada $f(x)$ directamente de los N valores tabulados cada vez que un valor nuevo es solicitado. Los esquemas más prácticos comienzan en un punto cercano $f(x_i)$, y agregan una secuencia de correcciones decrecientes, como información de otras $f(x_i)$'s incorporadas. El procedimiento toma $O(N^2)$ operaciones. Si todo se comporta bien, la última corrección será la más pequeña, y podrá utilizarse como un límite informal del error.

En el caso de interpolación polinomial, algunas veces ocurre que los coeficientes del polinomio de interpolación son de interés, aunque su uso para evaluar la función de interpolación no lo tenga.

El número de puntos (menos uno) usados en un esquema de interpolación es llamado el *orden* de la interpolación. Incrementar el orden no necesariamente incrementa la precisión, especialmente en la interpolación por polinomios. Si los puntos que se agregan están distantes del punto de interés x, el resultado de una polinomial de un orden más alto, con sus puntos obligatorios tradicionales, tiende a oscilar entre los valores tabulados. Esta oscilación puede no tener relación con el comportamiento de la "verdadera" función. Claro, si agregamos puntos *cerca* del punto deseado esto ayudará generalmente, pero esto implica una tabla de valores más grande, y no siempre está disponible.

A menos que exista una evidencia sólida de que la función de interpolación se parezca a la verdadera función f, es una buena idea tener precaución sobre interpolaciones de mayor orden. Podemos realizar interpolaciones con 3 o 4 puntos, ser tolerantes con 5 o 6 puntos; pero rara vez se debe ir con más sólo que haya un riguroso monitoreo de los errores estimados.

Cuando su tabla de valores contenga muchos más puntos que el orden deseable de interpolación, deberá comenzar cada interpolación con una búsqueda para el lugar "correcto" en la tabla.

Las rutinas dadas para interpolar también son rutinas de extrapolación. Los peligros de la extrapolación no se enfatizan, a menos que se tome especial cuidado en el monitoreo de errores: una función de interpolación se convierte en una de extrapolación cuando el argumento x se encuentra fuera del rango de los valores tabulados por más del espacio típico entre los valores de la tabla.

3.3.1 Interpolación y Extrapolación Polinomial

A través de dos puntos existe una línea única. A través de tres puntos, una única cuadrática. Etcétera. El polinomio de integración de grado N-1 que pasa a través de los N puntos $y_1 = f(x_1)$, $y_2 = f(x_2)$, ..., $y_N = f(x_N)$ está dado por la fórmula clásica de Lagrange,

$$P(x) = \frac{(x-x_2)(x-x_3)\cdots(x-x_N)}{(x_1-x_2)(x_1-x_3)\cdots(x_1-x_N)}y_1 + \frac{(x-x_1)(x-x_3)\cdots(x-x_N)}{(x_2-x_1)(x_2-x_3)\cdots(x_2-x_N)}y_2 + \cdots + \frac{(x-x_2)(x-x_3)\cdots(x-x_{N-1})}{(x_N-x_1)(x_N-x_3)\cdots(x_N-x_{N-1})}y_N \qquad 3.3.1.1$$

Hay N términos, cada uno un polinomio de grado $N-1$ y cada uno construido para ser cero en todas las x_i excepto una, la que se construyó para ser y_i.

No esta mal desarrollar la fórmula de Lagrange tal como se muestra, pero tampoco es lo mejor. El algoritmo resultante no arroja un error estimado y, además, resulta un programa un tanto enredado. Un mejor algoritmo (para construir el mismo, y único, polinomio de interpolación) es el *algoritmo de Neville.*

Sea P_1 el valor de x del polinomio único de grado cero (p.e. una constante) que pasa a través de los puntos (x_1, y_1); entonces $P_1 = y_1$. De la misma forma defina P_2, P_3, ..., P_N. Ahora sea P_{12} el valor de x del único polinomio de grado uno pasando tanto por (x_1, y_1) y por (x_2, y_2). De la misma forma se definen $P_{23}, P_{34}, \ldots, P_{(N-1)N}$. Similarmente, para polinomios de mayor orden, hasta $P_{123\ldots N}$, que es el valor del único polinomio de interpolación que pasa por los N puntos, p.e. la respuesta deseada.

Las varias P's de una "tabla" con "ancestros" en la izquierda arrojan a un sólo "descendiente" en la extrema derecha. Por ejemplo, con $N = 4$,

$$\begin{array}{lllll} x_1: & y_1 = P_1 & & & \\ & & P_{12} & & \\ x_2: & y_2 = P_2 & & P_{123} & \\ & & P_{23} & & P_{1234} \\ x_3: & y_3 = P_3 & & P_{234} & \\ & & P_{34} & & \\ x_4: & y_4 = P_4 & & & \end{array} \qquad 3.3.1.2$$

El algoritmo de Neville es una forma recursiva para llenar los números en la tabla una columna a la vez, de izquierda a derecha. Está basado en la relación entre una P "hija" y sus dos "padres",

$$P_{i(i+1)\ldots(i+m)} = \frac{(x - x_{i+m})P_{i(i+1)\ldots(i+m-1)} + (x_i - x)P_{(i+1)(i+2)\ldots(i+m)}}{x_i - x_{i+m}} \qquad 3.3.1.3$$

Esta recurrencia funciona debido a que los dos padres coinciden en los puntos $x_{i+1} \ldots x_{i+m-1}$.

Una mejora en la recurrencia (3.3.1.3) es guardar las pequeñas *diferencias* entre padres e hijas, las cuales nombraremos para definir (para m = 1, 2, ..., $N-1$),

$$C_{m,i} \equiv P_{i\ldots(i+m)} - P_{i\ldots(i+m-1)}$$
$$D_{m,i} \equiv P_{i\ldots(i+m)} - P_{(i+1)\ldots(i+m)}$$

Entonces podemos derivar fácilmente de (3.3.1.3) las relaciones

$$D_{m+1,i} = \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}}$$
$$C_{m+1,i} = \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}}$$

En cada nivel *m*, las C's y las D's son las correcciones que hacen que la interpolación sea de un orden mayor. La respuesta final $P_{123\ldots N}$ es igual a la suma de *cualquier* y_i, más un conjunto de C's y/o D's que forman el camino del árbol familiar hasta la hija de más a la derecha.

Esta es una rutina para interpolación o extrapolación polinomial de *N* puntos de entrada.

```
procedure InterpolarPolinomio( xxa: Variant; ix : integer; yya : Variant; iy, n :
Integer; x : Double; Var y, dy : Double );
{ Dados los arreglos xa[1..N] y ya[1..N], y dado un valor x, esta rutina regresa un
valor y, y y un error estimado dy. Si P(x) es un polinomio de grado N-1 tal que P(xai)
= yai, i = 1,...,N, el valor resgresado será y = P(x).
   Esta rutina permite tener tablas muy grandes y sólo indicar cuantos puntos se
utilizan y donde comienza la tabla para interpolar}
var i, m, ns : Integer;
    den, dif, dift, ho, hp, w : Extended;
    xa, ya, c, d : Variant;
begin
     ns := 1;

     c := varArrayCreate([1,n], varDouble );
     d := varArrayCreate([1,n], varDouble );
     xa := varArrayCreate([1,n], varDouble );
     ya := varArrayCreate([1,n], varDouble );

     if ix = 1 then xa := xxa
     else           for i:=ix to ix+n-1 do xa[i-ix+1] := xxa[i];

     if iy = 1 then ya := yya
     else           for i:=iy to iy+n-1 do ya[i-iy+1] := yya[i];

     dif := abs(x - xa[1]);
```

```
      {Aquí encontramos el índice ns para el elemento más cercano en la tabla}
      for i:=1 to n do begin
           dift := abs(x - xa[i]);
           if dift < dif then begin
              ns := i;
              dif := dift;
           end;

           {E inicializamos las tablas de c's y d's}
           c[i] := ya[i];
           d[i] := ya[i];
      end;

      y := ya[ns]; {y(ns--)}
      Dec(ns);

      for m:=1 to n-1 do begin
          for i:=1 to n-m do begin
              ho := xa[i] - x ;
              hp := xa[i+m] - x;
              w  := c[i+1] - d[i];
              den := ho-hp;
              if then = 0.0 Then
               raise EErrorInterpolacion.Create('Se detectaron 2 valores de X'
                                  +'idénticos mientras se evaluaba el polinomio');
              den := w /den;
              d[i] := hp * den;
              c[i] := ho * den;
          end;
          if (2*ns) < (n-m) then
             dy := c[ns+1]
          else begin
             dy := d[ns];
             Dec(ns);
          end;

          y := y + dy;
      end;
end;
```

3.3.2 Interpolación y Extrapolación por Funciones Racionales

Algunas funciones no son bien aproximadas por polinomios, pero son mejor aproximadas por funciones racionales, esto es cocientes de polinomios. Denotamos por $R_{i(i+1)\ldots(i+m)}$ una función racional que pasa por los $m+1$ puntos $(x_1, y_1) \ldots (x_{i+m}, y_{i+m})$. Más explícitamente, suponga

$$R_{i(i+1)\ldots(i+m)} = \frac{P_\mu(x)}{Q_\nu(x)} = \frac{p_0 + p_1 x + \cdots + p_\mu x^\mu}{q_0 + q_1 x + \cdots + q_\nu x^\nu} \qquad 3.3.2.1$$

Dado que hay $\mu+\nu+1$ p's y q's desconocidas, debemos tener

$$m + 1 = \mu + \nu + 1$$

Para especificar una función de interpolación por una función racional, debe especificar el orden deseado tanto para el numerador como para el denominador.

Las funciones racionales son en ocasiones superiores a los polinomios, gracias a su habilidad para modelar funciones con polos, esto es, ceros del denominador de la ecuación (3.3.2.1). Estos polos pueden ocurrir para valores reales de x, si la función a ser interpolada tiene polos por sí misma. En otras palabras, la función *f(x)* es finita para todos los valores reales finitos de x, pero tiene una continuación analítica con polos en el plano complejo x. Tales polos pueden por sí solos arruinar una aproximación polinomial. Si dibuja un círculo en el plano complejo alrededor de m puntos tabulados, entonces no espere un buen resultado de la interpolación polinomial a menos que el polo más cercano esté lo suficientemente alejado del círculo. Una función racional, en contraste, continuará dando "buenos" resultados mientras tenga suficientes potencias de x en su denominador contando para cancelar los polos cercanos.

Para el problema de la interpolación, una función racional se construye de tal forma que pasa por los puntos tabulados. Sin embargo, debemos mencionar que las aproximaciones de funciones racionales pueden usarse en trabajos analíticos. Algunas veces construimos aproximaciones a funciones racionales por el criterio de que la función racional de la ecuación (3.3.2.1) por sí misma tiene una expansión de series de poder que satisface con los primeros $m+1$ términos de la expansión de las series de poder de la función *f(x)* deseada, a esto se le llama la *aproximación de Padé.*

Bulirsch y Stoer encontraron un algoritmo del tipo del de Neville que realiza extrapolaciones de funciones racionales sobre valores tabulados. Una tabla como la de la ecuación (3.3.1.2) se construye columna a columna, dejando un resultado y un error estimado. El algoritmo Bulirsh – Stoer produce la llamada *función racional diagonal,* con los grados del numerador y denominador iguales (si m es par) o con el grado del denominador más grande por uno (si m es non). Para ver como se deriva el algoritmo acuda a [Stoer, 1980]. El algoritmo puede sumarizarse por una relación de recurrencia análoga exactamente a la ecuación (3.3.1.3) para la aproximación polinomial:

$$R_{i(i+1)\ldots(i+m)} = R_{(i+1)\ldots(i+m)} + \frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{\left(\frac{x-x_i}{x-x_{i+m}}\right)\left(1-\frac{R_{(i+1)\ldots(i+m)} - R_{i\ldots(i+m-1)}}{R_{(i+1)\ldots(i+m)} - R_{(i+1)\ldots(i+m-1)}}\right)-1} \qquad 3.3.2.3$$

Esta recurrencia genera las funciones racionales a través de $m + 1$ puntos desde los primeros a través de los m y $m - 1$ puntos (el término $R_{(i+1)\ldots(i+m-1)}$ en la ecuación 3.3.2.3). Este comienza con

$$R_i = y_i$$

y con

$$R \equiv \left[R_{i(i+1)\ldots(i+m)} \text{ con } m = -1\right] = 0$$

Ahora, exactamente como en la ecuación (3.3.1.4) y (3.3.1.5), podemos convertir la recurrencia (3.3.2.3) por una que involucre sólo las diferencias pequeñas

$$C_{m,i} \equiv R_{i\ldots(i+m)} - R_{i\ldots(i+m-1)}$$
$$D_{m,i} \equiv R_{i\ldots(i+m)} - R_{(i+1)\ldots(i+m)}$$

Note que estas satisfacen la relación

$$C_{m+1,i} - D_{m+1,i} = C_{m,i+1} - D_{m,i}$$

que es útil para probar las recurrencias

$$D_{m+1,i} = \frac{C_{m,i+1}(C_{m,i+1} - D_{m,i})}{\left(\frac{x - x_i}{x - x_{i+m+1}}\right) D_{m,i} - C_{m,i+1}}$$

$$C_{m+1,i} = \frac{\left(\frac{x - x_i}{x - x_{i+m+1}}\right) D_{m,i}(C_{m,i+1} - D_{m,i})}{\left(\frac{x - x_i}{x - x_{i+m+1}}\right) D_{m,i} - C_{m,i+1}}$$

Esta recurrencia se implementa en la siguiente función, cuyo uso es análogo en muchas formas a InterpolarPolinomio (ver tema Interpolar §3.3.1 de éste capítulo).

```
procedure TInterpolarRacional.InterpolarRacional( xxa: Variant; ix : integer; yya :
Variant; iy, n : Integer; x : Double; Var y, dy : Double );
{ Dados los arreglos xa[1..n] y ya[1..n], y dado un valor de x, esta rutina regresa un
valor de y  y un error estimado dy. El valor regresado es el de la función racional
diagonal, evaluada en x, que pasa por los n puntos (xai, yai ), i = 1...n.
   Esta rutina permite tener tablas muy grandes y sólo indicar cuantos puntos se
utilizan y donde comienza la tabla para interpolar }
const TINY = 1.0e-25;     { Un número pequeño }
var
   m,i,ns : Integer;
   w,t,hh,h,dd : Double;
   xa, ya, c,d : Variant;
begin
   ns := 1;
   c := varArrayCreate([1,n], varDouble );
   d := varArrayCreate([1,n], varDouble );

   xa := varArrayCreate([1,n], varDouble );
   ya := varArrayCreate([1,n], varDouble );

   if ix = 1 then xa := xxa
   else           for i:=ix to ix+n-1 do xa[i-ix+1] := xxa[i];

   if iy = 1 then ya := yya
   else           for i:=iy to iy+n-1 do ya[i-iy+1] := yya[i];

   hh := abs(x-xa[1]);
```

```
    for i:=1 to n do begin
        h := abs(x-xa[i]);
        if (h = 0.0) then begin
           y := ya[i];
           dy:= 0.0;
           exit;
        end
        else if (h < hh) then begin
             ns :=i;
             hh :=h;
        end;
        c[i]:=ya[i];
        d[i]:=ya[i]+TINY;  {La parte TINY es necesaria para prevenir una }
                           { rara condición de cero sobre cero }
    end;

    y := ya[ns]; dec(ns);

    for m:=1 to n-1 do begin
        for i:=1 to n-m do begin
            w := c[i+1]-d[i];
            h := xa[i+m]-x;      {h nunca es cero, dado que fue probado en el ciclo}
                                 { de inicialización}
            t := (xa[i]-x)*d[i]/h;
            dd:= t-c[i+1];
            if (dd = 0.0) Then
              raise EErrorPolInt.Create('Se encontró un "polo" en el valor X'
                                        +'solicitado');
            dd:= w/dd;
            d[i]:=c[i+1]*dd;
            c[i]:=t*dd;
        end;

        if (2*ns) < (n-m) then
           dy := c[ns+1]
        else begin
           dy := d[ns]; dec(ns);
        end;
        y := y + dy;
    end;
end;
```

3.3.3. Coeficientes del Polinomio de Interpolación

Ocasionalmente no deseará saber el valor del polinomio de interpolación que pasa a través de un número (pequeño) de puntos, sino de los coeficientes de ese polinomio. Un uso válido de estos coeficientes puede ser, por ejemplo, calcular valores de la función interpolados simultáneamente y de sus derivadas, o comparar un segmento de la función tabulada como alguna otra función, donde los momentos de esa otra función se conocen analíticamente.

Sin embargo, debe estar seguro para que necesite los coeficientes. Generalmente los coeficientes del polinomio de integración pueden determinarse con mucha menos precisión que sus valores en una abscisa deseada. Además, no es buena idea determinar los coeficientes sólo para usarlos en calcular valores interpolados. Los valores calculados de esta manera no pasarán exactamente

por los puntos tabulados, mientras que los valores calculados por las rutinas anteriores pasan exactamente por tales puntos.

Como antes, tomamos los puntos tabulados para decir $y_i \equiv y(x_i)$. Si el polinomio de interpolación se escribe como

$$y = c_0 + c_1 x + c_2 x^2 + \cdots + c_N x^N \qquad 3.3.3.1$$

Entonces las c_i's se requieren para satisfacer la ecuación lineal

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^N \\ 1 & x_1 & x_1^2 & \cdots & x_1^N \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^N \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{bmatrix} \qquad 3.3.3.2$$

Esta es una *matriz Vandermonde*, como se describe en [Numericals, 1992]. Se puede resolver en principio la ecuación (3.3.3.2) por técnicas estándar de ecuaciones lineales; sin embargo, el método especial que se deriva de [Numericals, 1992] es más eficiente para un factor más grande, de orden N, así que este es mejor.

Los sistemas Vandermonde pueden estar condicionados. En tal caso, ningún método numérico dará una respuesta precisa. Tales casos no implican alguna dificultad en encontrar valores interpolados por los métodos anteriores, sólo es difícil encontrar sus coeficientes.

La siguiente rutina se debe a G.B. Rybucki.

```
function TCoeficientesPolinomio.CoeficientesPolInt( x, y : Variant; n : Integer ) :
Variant;
{ Dados los arreglos x[0..n] y y[0..n] conteniendo una función tabulada yi = f(xi),
esta rutina regresa un arreglo de coeficientes cof[0..n], tal que yi = Sumj cofj xi^j.
Usando el algoritmo de Vandermonde }
var
    k,j,i    : Integer;
    phi,ff,b : Double;
    cof, s   : Variant;
begin
   s := varArrayCreate([0,n], varDouble ); {Note que son arreglos basados en cero}
   cof := varArrayCreate([0,n], varDouble );
   for i:=0 to n do begin
       s[i] := 0.0; cof[i]:=0.0;
   end;
   s[n] := -x[0];
   for i:=1 to n do begin   { Los coeficientes si del polinomio maestro P(x) se
encuentran por recurrencia }
       for j:=n-i to n-1 do s[j] := s[j] - x[i]*s[j+1];
       s[n] := s[n] - x[i];
   end;
   for j:=0 to n do begin
       phi:=n+1;
```

```
      for k:=n downto 1 do {La cantidad phi se encuentra como una derivación de
P(xj)}
          phi := k*s[k]+x[j]*phi;
      ff := y[j]/phi;
      b  := 1.0;  { Los coeficientes polynomiales en cada término de la fórmula}
      for k:=n downto 0 do begin   { de Lagrange se encuentran por división sintética
de P(x) por (x-xj) }
          cof[k] := cof[k] + b*ff; { la solución ck es acumulada }
          b:=s[k]+x[j]*b;
      end;
   end;
   result := cof;
end;
```

Capítulo 4

Evaluación de Funciones y sus Derivadas

4.1 Derivadas Numéricas

Imagine que tiene un procedimiento que calcula una función *f(x)*, y ahora desea calcular su derivada *f'(x)*. La definición de la derivada, el límite cuando $h \to 0$ de

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \qquad 4.1$$

prácticamente sugiere un programa: Tomar un valor pequeño de *h*; evaluarlo en *f(x + h)*; y también en f(x), finalmente, aplicar la ecuación (4.1).

Aplicado sin cuidado, el procedimiento anterior puede producir resultados incorrectos. Aplicado correctamente, puede ser el camino correcto para calcular una derivada. Hay dos fuentes de error en la ecuación (4.1), error por truncamiento y error por redondeo.

Si no es importante el número de evaluaciones de la función para el cálculo de cada derivada, es significativamente mejor usar la forma simétrica

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \qquad 4.2$$

la cual tiene un error por truncamiento de $e_t \sim h^2 f'''$, mientras que el de la primera es $e_t \sim h^2 f''$, el cual es significativamente menor pues la tercera derivada de la función es, generalmente, más pequeña que la segunda derivada. Aunque el error por redondeo es el mismo.

Es decepcionante, claro esta, que ninguna fórmula simple de diferencias finitas como las de las ecuaciones (4.1) y (4.2) dan una precisión comparable con la de la máquina, o con la menor de las precisiones cuando se evalua f. ¿No existen métodos mejores?

Si existen otros métodos. Todos, sin embargo, involucran la exploración del comportamiento de la función sobre escalas de la curvatura de la función f, $x_c \equiv (f/f'')^{1/2}$, además de asumir un grado de suavidad. Tales métodos también involucran multiples evaluaciones de la función f, así que su precisión mejorada debe pesarse contra el número de evaluaciones por realizar.

La idea general de "la aproximación diferida del límite de Richardson" es particularmente atractiva para estos casos. Para integrales numéricas, esta idea genera la llamada integración de Romberg (vea el capítulo 5). Para derivadas, se busca extrapolar, a $h \to 0$, el resultado del cálculo de diferencias finitas con valores de h finitos cada vez más pequeños. Con el uso del algoritmo de Neville (vea tema §3.3.1), usamos cada nuevo cálculo de diferencias finitas para producir tanto una extrapolación de un orden mayor, como las extrapolaciones de ordenes menores pero con escalas menores de h. [Ridders, 4] ha dado una implementación de esta idea; el siguiente programa, DFRidders, se basa en su algoritmo, modificado para un criterio de finalización mejorado. La entrada de la rutina es una función f, una posición x, y un intervalo grande h (más análogo a lo que llamamos x_c anteriormente que los que hemos llamado h). La salida es el valor de la derivada y un estimado de su error.

```
function TDerivadaNum.DFRidders( func : TUserFunc; x, h : Double; Var err : Double ) :
Double;
{ Regresa la derivada de una función func en el punto x por el método de extrapolación
de polinomios de Ridders. El valor h es una entrada como un tamaño de intervalo
inicial; no necesita ser pequeño, pero deberá ser un incremento en x sobre el cual
func cambie sustancialmente. Un estimado del error en la derivada se regresa como
err.}
const    CON = 1.4;          {El tamaño del intervalo se decrementa CON veces en cada
iteración}
   CON2 = (CON*CON);
   BIG = 1.0e30;
   NTAB = 10;          { Tamaño máximo de la tabla }
   SAFE = 2.0;         { Return when error is SAFE worse than the best so far.}
var
   i,j:Integer;
   errt,fac,hh,ans : Double;
   a : Variant;

   function FMAX(a, b : Double) : Double;
   begin
        result := a;
        if b > a then result := b;
   end;

begin
   if (h = 0.0) then raise EErrorDerivadaNum.Create('h no debe ser cero en la derivada
por el método de Ridders');
   ans := 0;
   a := varArrayCreate([1,NTAB,1,NTAB], varDouble);
   hh:=h;
   a[1,1]:= (func(x+hh)-func(x-hh))/(2.0*hh);
   err:=BIG;
   for i:=2 to NTAB do begin
   { Columnas sucesivas en la tabla de Neville serán de intervalos más pequeños
```

```
      y de ordenes mayores de extrapolación.}
      hh := h / CON;
      a[1,i] := ( func(x+hh) - func(x-hh))/(2.0*hh);
              {Se busca de nuevo en un intervalo más pequeño}
      fac:=CON2;
      for j:=2 to i do begin
       {el cálculo de extrapolaciones de varios ordenes, no requiere nuevas
        evaluaciones de la función}
          a[j,i]:=(a[j-1, i]*fac - a[j-1,i-1])/(fac-1.0);
          fac:=CON2*fac;
          errt:=FMAX( abs(a[j,i]-a[j-1,i]), abs(a[j,i]-a[j-1,i-1]));
          { La estrategia de error es comparar cada nueva extrapolación con una de
orden menor, ambas con el mismo tamaño de intervalo y con el anterior.}

          if (errt <= err) then begin { Si el error decrece, guardar la respuesta
mejorada.}
             err:=errt;
             ans:=a[j,i];
          end;
      end;
      if ( abs(a[i,i]-a[i-1,i-1]) >= SAFE*err) then break;
      {Si el orden mayor es peor que un factor significativo SAFE, entonces salir
antes}
    end;
    result := ans;
end;
```

En DFRidders, el número de evaluaciones de la función func es típicamente de 6 a 12, pero puede ser tan grande como 2xNTAB. Como una función de entrada h, es típico para la precisión que sea mejor entre más grande sea h, hasta un punto donde la extrapolación produzca errores grandes.

Adicionalmente al método de Ridders, existen otras técnicas. Si su función es lo suficientemente suave, y sabe que desea evaluar su derivada muchas veces en puntos arbitrarios en algún intervalo, entonces es mejor construir un polinomio de aproximación de Chebyshev a la función en ese intervalo, y evaluar la derivada directamente de los coeficientes resultantes del polinomio de Chebyshev.

Otra técnica se aplica cuando la función consiste de datos que están tabulados a intervalos igualmente espaciados, puede usarse la técnica de filtros suavizadores de Savitzky-Golay.

4.2 Listado del componente de Derivadas Numéricas

Métodos

Derivar. Dados los valores de x y h, este procedimiento regresa el valor de la derivada de la función definida por el usuario, así como un error estimado de la misma.

Eventos

OnUserFunc. Es en este evento donde el usuario escribe la función que desea derivar.

Listado

```
unit DerivadaNumerica;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  EErrorDerivadaNum = class(Exception);
  TUserFunc = function( x : Double ) : Double of Object;
  TDerivadaNum = class(TComponent)
  protected
    FUserFunc : TUserFunc;
    function DFRidders( func : TUserFunc; x, h : Double; Var err : Double ) : Double;
  public
    procedure Derivar( x, h : Double; Var derivada, error : Double );
  published
    property OnUserFunc : TUserFunc read FUserFunc write FUserFunc;
  end;

procedure Register;

implementation

function TDerivadaNum.DFRidders( func : TUserFunc; x, h : Double; Var err : Double ) :
Double;
{ Regresa la derivada de una función func en el punto x por el método de extrapolación
de polinomios de Ridders. El valor h es una entrada como un tamaño de intervalo
inicial; no necesita ser pequeño, pero deberá ser un incremento en x sobre el cual
func cambie sustancialmente. Un estimado del error en la derivada se regresa como
err.}
const    CON = 1.4;          {El tamaño del intervalo se decrementa CON veces en cada
iteración}
   CON2 = (CON*CON);
   BIG = 1.0e30;
   NTAB = 10;      {Tamaño máximo de la tabla}
   SAFE = 2.0;       { Return when error is SAFE worse than the best so far.}
var
   i,j:Integer;
   errt,fac,hh,ans : Double;
   a : Variant;

   function FMAX(a, b : Double) : Double;
   begin
        result := a;
        if b > a then result := b;
   end;

begin
   if (h = 0.0) then raise EErrorDerivadaNum.Create('h no debe ser cero en la derivada
por el método de Ridders');
   ans := 0;
```

```
   a := varArrayCreate([1,NTAB,1,NTAB], varDouble);
   hh:=h;
   a[1,1]:= (func(x+hh)-func(x-hh))/(2.0*hh);
   err:=BIG;
   for i:=2 to NTAB do begin
   { Columnas sucesivas en la tabla de Neville serán de intervalos más pequeños y de
ordenes
     mayores de extrapolación.}
     hh := h / CON;
     a[1,i] := ( func(x+hh) - func(x-hh))/(2.0*hh); {Se busca de nuevo en un intervalo
más pequeño}
     fac:=CON2;
     for j:=2 to i do begin { el cálculo de extrapolaciones de varios ordenes, no
requiere nuevas evaluaciones de la función}
         a[j,i]:=(a[j-1, i]*fac - a[j-1,i-1])/(fac-1.0);
         fac:=CON2*fac;
         errt:=FMAX( abs(a[j,i]-a[j-1,i]), abs(a[j,i]-a[j-1,i-1]));
         { La estrategia de error es comparar cada nueva extrapolación con una de
orden menor, ambas con el mismo tamaño de intervalo y con el anterior.}

         if (errt <= err) then begin { Si el error decrece, guardar la respuesta
mejorada.}
            err:=errt;
            ans:=a[j,i];
         end;
     end;
     if ( abs(a[i,i]-a[i-1,i-1]) >= SAFE*err) then break;
     {Si el orden mayor es peor que un factor significativo SAFE, entonces salir
antes}
   end;
   result := ans;
end;

procedure TDerivadaNum.Derivar( x, h : Double; Var derivada, error : Double );
begin
     derivada := DFRidders(FUserFunc, x, h, error);
end;

procedure Register;
begin
  RegisterComponents('MAC', [TDerivadaNum]);
end;

end.
```

4.3 Evaluando Funciones

Evaluar una función escrita por el usuario, sin recurrir a métodos numéricos, sino entregar el valor real de esa función como si lo hiciéramos a mano, implica una labor de traducción, del lenguaje matemático (en notación infija) a símbolos dentro de la computadora.

Una forma eficiente para evaluar una expresión matemática (por una computadora) es en notación posfija, ya que no puede existir ambigüedad en la expresión en cuanto a la prioridad de los operadores y se puede seguir un algoritmo sencillo para ello.

Cuando se nos presenta un problema de traducción, el primer paso es recurrir a un diccionario que nos permita conocer las reglas gramaticales de cada uno de los lenguajes que se traducirán.

De igual forma, en la transformación de una cadena escrita en notación infija a otra en notación posfija debemos recurrir a su constitución gramatical, o sea, el lenguaje que reconoce dichas cadenas. Y gracias a que estas notaciones forman parte de los lenguajes regulares, podemos recurrir a sus modelos matemáticos para auxiliarnos en su traducción.

Así, el lenguaje que reconoce expresiones infijas simples se denota por:

expr → *expr* + *término*
expr → *expr* - *término*
expr → *término*
término → 0
...
término → 9

Y el lenguaje que reconoce expresiones posfijas simples se denota por:

expr → *expr término* +
expr → *expr término* -
expr → *término*
término → 0
...
término → 9

Como podemos ver en los dos lenguajes anteriores, la diferencia son únicamente las expresiones regulares que son afectadas por operadores, y esta diferencia radica en la posición del operador.

A menudo, un esquema de traducción dirigida a la sintaxis puede servir como especificación de un traductor. Estos esquemas incluyen instrucciones dentro de la sintaxis como *print* para resaltar que acción se debe tomar en tal o cual expresión. Por ejemplo, un lenguaje que traduce una expresión infija simple a una posfija es el siguiente:

expr → *término resto*
resto → + *término* {*print* ('+')} resto | - *término* {*print* ('-')} resto | ε
término → 0 {*print* ('0')}
...
término → 9 {*print* ('9')}

Este lenguaje se presenta del hecho de que las expresiones infijas y posfijas sólo difieren en la posición del operador. El lenguaje representa a las expresiones infijas, pero las instrucciones {*print*} dan el orden necesario para generar una expresión posfija.

Un lenguaje más complejo, puede reconocer las otras operaciones faltantes; aquí se presenta el problema de la precedencia de los operadores y que los términos pueden ser números de más de un dígito o incluso nombres de variables o funciones. Para el primer problema agregamos otra

producción que nos permita distinguir entre las precedencias de operadores, pero para el segundo problema se requiere agregar un analizador léxico y una tabla de símbolos para poder almacenar las propiedades de cada término, así que el algoritmo que se presenta tiene como alcance el poder traducir cadenas infijas a posfijas con términos de un dígito.

El lenguaje traductor de infijo a posfijo se define como:

$$
\begin{aligned}
expr &\rightarrow expr + término \quad \{print('+')\} \\
&\quad \mid expr - término \quad \{print('-')\} \\
&\quad \mid término \\
término &\rightarrow término * factor \ \{print('*')\} \\
&\quad \mid término / factor \quad \{print('/')\} \\
&\quad \mid factor \\
factor &\rightarrow (expr) \\
&\quad \mid 0 \qquad\qquad\qquad \{print('0')\} \\
&\quad \ldots \\
&\quad \mid 9 \qquad\qquad\qquad \{print('9')\}
\end{aligned}
$$

4.3.1 Algoritmo para pasar una cadena escrita en notación infija a notación posfija (operadores +,-,*,/)

```
/* Notación:
      c = cadena en infijo
      i = posición índice, indica el avance por la cadena infija
      f = fin, es el tamaño de la cadena infija
      p = cadena en posfijo
*/

/* Factor: Escribe en la cadena posfija un número, o si encuentra un paréntesis
abierto ejecuta el traductor de expresiones y verifica que se cierre el paréntesis, si
no es alguno de estos casos regresa error */

Function Factor( c: String; Var i, f : Integer; Var p : String ) : Boolean;
Var err : Boolean;
Begin
     If( (i <= f) and ( c[i] in ['0'..'9']) ) then
     begin
          p := p + c[i];
          i := i+1;
          err := False;                    { no hubo error }
     end
     else
       if ( c[i] = '(' ) Then
       begin
            i := i+1;
            err := Expr( c, i, f, p );
            err := ( c[i] = ')' );       { Error si no se cierra el paréntesis }
            i := i+1;
       end
       else
         err := true;                    { hubo error, carácter no reconocible }

     Factor := err;
End;
```

```
/* Término: Escribe en la cadena posfija un factor, si encuentra un operador * o /
asigna el segundo factor a la cadena posfija y al final escribe el operador, si no es
alguno de estos casos regresa error */

Function Termino( c: String; Var i, f : Integer; Var p : String ) : Boolean;
Var salir, error : Boolean;
    prea  : char ;
Begin
     error := Factor( c, i, f, p);
     salir := False;
     while( (Not Salir) And (Not error) And ( i <= fin ) ) Do
     begin
          prea := c[i];
          if ( (prea = '*') Or (prea = '/') ) then
          begin
             i := i+1;   { siguiente elemento }
             error := Factor( c, i, f, p );
             p := p + prea;
          end
          else
               Salir := True;
     end;
     Termino := error;
End;


/* Infi2Posf: Escribe en la cadena posfija un término, si encuentra un operador + o -
asigna el segundo término a la cadena posfija y al final escribe el operador, si no es
alguno de estos casos regresa error */

Function Infi2Posf( c: String; Var i, f : Integer; Var p : String ) : Boolean;
var error : Boolean;
    prea  : char;
Begin
     p := '';
     error := Termino( c, i, f, p);
     while( (Not error) And ( i <= fin ) ) Do
     begin
          prea := c[i];
          if ( (prea='+') Or ( prea='-') ) then
          begin
             i := i+1;   { siguiente elemento }
             error := Termino( c, i, f, p );
             p := p + prea;
          end
          else
              error := True;
     end;
     Infi2Posf:= Not error;
End;
```

4.3.2 Desarrollando el Evaluador de Funciones

El problema principal es poder reconocer números de más de un dígito, identificar funciones predefinidas en la computadora (ya sea las que proporciona el lenguaje o alguna otra creada por el usuario), como las trigonométricas y los logaritmos.

La primera fase del evaluador de funciones consiste en leer una cadena e identificar cada uno de sus componentes, almacenando sus características en una estructura de datos de tipo lista ligada para su traducción posterior a una expresión posfija.

```
procedure TFuncion.MeterFuncion( cad : String; Var ap_ini, ap_actual : TPLista );
Var primer_num, primer_car, negativo : Boolean;
    anterior, e, f : Char;
    aux, ext, otra : String;
    i, cp, cont_elem, cparent : Integer;
    y : TElemento;
    lp_inicio, lp_actual : TPLista ;
begin
   primer_num := True;
   primer_car := True;
   negativo   := False;
   anterior   := '@';
   cparent    := 0 ;

   i:=1; cont_elem:=0;

   lp_actual := ap_actual;
   lp_inicio := ap_ini;

   while( i <= Length(cad) ) do
   begin
       e := UpCase(cad[i]);

       if ( Not (e in ['A'..'Z', '0'..'9'])) and  {Si no es un número o letra}
          ( Not EsOperador(e) ) and               {Si no es un operador}
          ( Not (e in ['.', ' '])) Then           {Si no es el punto el espacio}
          raise EErrorLeerFuncion.Create('Se detectó un carácater No válido');

       if (e in ['A'..'Z']) and (e <> 'X') Then {Leer funciones}
       begin
            ext := ext + e;
            primer_car := False;
            Inc(i);
            continue ;
       end;

       if (e in ['0'..'9', '.'] ) Then     {Leer números}
       begin
            aux := aux + e;
            primer_num := False;
            {Inc(i);}
            {continue ; {Originalmente no iba pero igual y jala}
       end;

       if ( (e='X') or EsOperador(e) ) Then
       begin
           if ( ( e = '-') and         {Se valida el elemento sea negativo}
                ( (anterior = '@') or
                  (anterior = '^') or
                  (anterior = '(') ) ) Then   (* el menos entra abajo *)
           begin
               Inc(cont_elem);
               y.oper := '(' ;                (* insertar un ( 0 - n ) *)
               Ligar(y, opOperador, lp_inicio, lp_actual);
               y.num := 0.0 ;
               Ligar(y, opConstante, lp_inicio, lp_actual);
               negativo := True ;
               anterior := ' ';
```

```
end;

if ( ((anterior = 'X') or (anterior = 'f') or (anterior = ')'))
   and ((e = 'X') or (e = '(')) ) then  {  agregar un *  }
begin  {Forzar la multiplicación de implícita a explícita}
    y.oper := '*';
    Ligar(y,opOperador, lp_inicio, lp_actual);
    Inc(cont_elem);
end;

if ( Not primer_car) and (e<>'(') then
{si se leyó una función y no hay paréntesis}
    raise EErrorLeerFuncion.Create('Función sin paréntesis abierto');

if (Not primer_num) then  {Si se leyó algún número}
begin
    try
      y.num := StrToFloat(aux);
    except
      raise EErrorLeerFuncion.Create(aux + ' no es un número válido');
    end;
    Ligar(y, opConstante, lp_inicio, lp_actual); {Insertar el número}
    Inc(cont_elem);

    aux := ''; {Preparar para el siguiente número}

    if ( negativo ) then              (* cerrar el ( 0 - n ) *)
    begin
        y.oper := ')' ;
        Ligar(y, opOperador, lp_inicio, lp_actual);
    end;

    if ( (e = 'X') or (e = '(') ) then
    {agregar una multiplicación explicita i.e. 2x => 2 * x}
    begin
        y.oper := '*';
        ligar(y, opOperador, lp_inicio, lp_actual);
        Inc(cont_elem);
    end;

    primer_num := True;
    negativo := False;
end;

if (Not primer_car) and (e = '(') then (* Si se leyó una función *)
begin
    f := IdentificaFuncion( ext );

    cp := 1; otra := '';
    Inc(i);
    e := UpCase(cad[i]);
    while((cp >= 1) and ( i<=length(cad) )) do (* capturar el argumento *)
    begin                                      (* de la función *)
       otra := otra + e;
       Inc(i);
       e := UpCase(cad[i]);
       if (e = '(') then Inc(cp); {contando los paréntesis}
       if (e = ')') then Dec(cp);
    end;

    if ( cp = 0 ) then       (* si se cerraron todos los parentesis *)
    begin                    (* incluido el de la función *)
```

```
                    y.oper := f;        (* Hacer una llamada recursiva para el argumento
*)
                    Ligar( y, opFuncion, lp_inicio, lp_actual );

                    MeterFuncion( otra, lp_actual^.inicio, lp_actual^.actual );

                    primer_car := True; ext := ''; {Listo para leer otra función}

                    Inc(cont_elem) ;
                    anterior := 'f';

                    if ( negativo ) then    { por si se antepuso un signo (-) }
                    begin
                      y.oper := ')' ;
                      Ligar(y,opOperador, lp_inicio, lp_actual);
                      negativo := False ;
                    end;

                    Inc(i);
                    continue ;
                end
                else
                    raise EErrorLeerFuncion.Create('Falta cerrar un paréntesis en el
argumento de la función');

            end; { de si se leyó una función}


            if ( e = '(' ) then Inc(cparent);      (* contar los parentesis *)
            if ( e = ')' ) then Dec(cparent);

            y.oper := e;
            if (e = 'X') then
               Ligar(y, opVariable, lp_inicio, lp_actual )
            else
               Ligar(y, opOperador, lp_inicio, lp_actual );

            Inc(cont_elem);

            if (EsOperador(anterior)) and (Not (anterior in ['(',')'])) and
               (Not (e in ['(', ')', 'X'] ) ) then
               raise EErrorLeerFuncion.Create('Se encontró un doble operador');

        end;                              (* fin del Sí operador *)

        if ( e <> ' ') then anterior := e;

        Inc(i);
    end;                             (* fin del while *)

  if (Not primer_num) and (anterior <> '@') then (* por si se quedo leyendo un número
*)
  begin
        try
          y.num := StrToFloat(aux);
        except
          raise EErrorLeerFuncion.Create(aux + ' no es un número válido');
        end;
        Ligar(y, opConstante, lp_inicio, lp_actual);

        Inc(cont_elem);
        if ( negativo ) Then  {cerrar el ( 0-n }
        begin
```

```
            y.oper := ')' ;
            Ligar(y,opOperador, lp_inicio, lp_actual);
        end;
    end;

    if (Not primer_car) then (* por si se quedo leyendo una función *)
       raise EErrorLeerFuncion.Create('Se encontró una función sin argumentos');

    if (cont_elem mod 2) = 0 then (* hay un número par de elementos (i.e 1- *)
       raise EErrorLeerFuncion.Create('La expresión es inválida, falta un operando');

    if ( cparent <> 0 ) then    (* hay paréntesis abiertos *)
       raise EErrorLeerFuncion.Create('Los paréntesis no están balanceados');

    ap_ini    := lp_inicio;
    ap_actual := lp_actual;
end;
```

Al mismo tiempo que se traduce la cadena escrita en notación infija a una representación de lista ligada, se verifica que la sintaxis de la expresión sea la correcta.

En la segunda fase se traduce la lista ligada a una pila, colocando cada uno de los elementos en su expresión posfija, como se hizo en el 4.3.1, pero ahora en lugar de insertar caracteres se insertan en la pila elementos de la estructura junto con sus atributos.

Las pilas son otro tipo de lista. En una pila los elementos se van insertando por la parte de arriba de la pila, y en cualquier momento se pueden eliminar de ese lugar. Esto se conoce como LIFO último en entrar, primero en salir. Sólo se tienen que hacer dos operaciones sobre la pila:

push *(introduce)* coloca un nuevo elemento arriba de la pila
pop *(extrae)* elimina el elemento de arriba de la pila, exponiendo el anterior, si lo hay.

```
procedure TFuncion.posfija( Var q, inicio, actual : TPLista);
{regresa la expresión posfija de actual en q}
Var op, n : Char;
    t     : TOperandos;
    y     : TElemento;
    e     : TPLista;
    r, p  : TPLista;

    {útil para las funciones}
    ex : TPLista;
    z  : TElemento;
    m  : TOperandos;

begin
   r := Nil; p := Nil; ex := Nil;
   e := inicio;

   y.oper := '(';
   push( y, opOperador, p );

   y.oper :=')'; Ligar(y, opOperador, inicio, actual);

   while (p <> Nil) do
   begin
       if ( ((e^.tipo) = opConstante) or ( (e^.tipo) = opVariable) ) then
          push( e^.elemento, e^.tipo, r );
```

```
        if ( (e^.tipo) = opFuncion ) then
        begin
            push( e^.elemento, e^.tipo, r );
            posfija( ex, e^.inicio, e^.actual);
            while( e^.inicio <> Nil ) do pop( z, m, e^.inicio) ;
            r^.inicio := ex;
            r^.actual := ex;  (* apunta a la pila de notacion de la función *)
        end;
        if ( (e^.tipo) = opOperador ) then
        begin
            op := e^.elemento.oper;

            if (op = '(') then push( e^.elemento, e^.tipo, p );

            if ( (op <> '(') and (op <> ')') ) then
            begin
               n := p^.elemento.oper;

               while( (n <> '(') and ( prioridad(op) <= prioridad(n) ) ) do
               begin
                   pop( y, t, p );
                   push( y,  t, r );
                   n := p^.elemento.oper;
               end;
               push( e^.elemento, e^.tipo, p );
            end;

            if ( op = ')' ) then
            begin
               while( (p^.elemento.oper) <> '(' ) do
               begin
                  pop( y, t, p );
                  push( y,  t, r );
               end;
               pop( y, t, p );  (* vaciarlo *)
            end;
        end;
        e := e^.siguiente;
    end;
    q := r;
end;
```

Los elementos de la expresión deben ser impares, no se permite, por ejemplo, 4 +, dos elementos, 5 / 6 + 7 -, seis elementos, es válido un número negativo, pues la traducción lo convierte en un solo elemento.

Este procedimiento puede ser mejorado basándose en el concepto de las producciones y gramáticas, puede consultar [Aho, 1990] para mayores detalles de cómo construir un analizador sintáctico para identificar si una expresión esta escrita correctamente, y cómo se construye y usa una tabla sintáctica para almacenar las propiedades de los elementos individuales de una expresión.

4.3.3 Evaluación de la derivada analítica de una función

Ahora realizaremos el procedimiento para evaluar la derivada de las pilas que se genera por medio del evaluador de funciones realizado anteriormente.

El procedimiento para derivar analíticamente una pila se sugiere como ejercicio.

Un método sencillo para derivar una función es aplicar las reglas analíticas que para ello existen:

Derivada de una constante = 0
La derivada de X = 1
La derivada de una suma es igual a la suma de la derivada de los términos
La derivada de uv = uv' + vu'
Etcétera

Una estructura de datos que nos permite realizar estas operaciones de forma que su aplicación resulta muy natural es un árbol.

Los árboles son una estructura de datos no lineal que se emplea para representar estructuras jerárquicas y direcciones o etiquetas de forma organizada.

Algunas de sus aplicaciones son en la organización, ordenación y búsqueda. Además de que nos permiten representar situaciones comunes como son los organigramas de empresas u organizaciones de torneos o eventos.

La definición formal de un árbol es:

Un árbol es un conjunto finito T de uno o más nodos de tal forma que existe un nodo especial llamado raíz y los nodos restantes pueden ser particionados en conjuntos disjuntos de tal forma que cada uno de ellos es a su vez un árbol y se denomina subárbol del árbol T.

El número de hijos que se desprenden de un nodo se denomina grado del nodo. El grado de un árbol es el grado máximo de los nodos de un árbol. Un nodo de grado cero es llamado hoja.

Algunos otros ejemplos de árboles son los árboles binarios, árboles binarios de búsqueda y árboles balanceados.

Los árboles binarios son un caso particular de los árboles y se emplean principalmente en el área de computación para poder limitar el número de ligas y la manipulación interna para explotar la información.

Este nuevo concepto de árbol binario rompe con el concepto teórico de árbol, en el que un conjunto T de nodos es llamado árbol binario si y sólo si T es vacío o T esta particionado en 3 conjuntos disjuntos tales que el primero de ellos está compuesto de un sólo nodo llamado raíz y los dos siguientes son llamados a su vez árboles binarios.

Un árbol binario de búsqueda es aquel que es un árbol binario y además existe una llave de búsqueda "n" que es mayor que cero o igual a todas las llaves de búsqueda del subárbol izquierdo y la llave de búsqueda "n" es menor a todas las llaves de búsqueda del subárbol derecho.

Un árbol se dice balanceado si y sólo si el peso de todos sus nodos difieren al menos en uno. A diferencia de los otros tipos de árbol, en el balanceado las operaciones pueden ser realizadas en menor tiempo, para esto se trabaja con una llave dada para la búsqueda, inserción y borrado. Dentro de la estructura del árbol se anota un +1 para identificar que ha sido equilibrado, 0 si el tamaño esta normalizado a la izquierda y -1 cuando se necesite rebalancear. La definición anterior ha sido postulada por Adelson-Velskii y Landis.

En un árbol se puede leer una expresión en tres formas, dependiendo el orden:

Infijo. La forma en la que escribimos las expresiones. Recorrer en Infijo el nodo izquierdo, aplicar el operador y recorrer el nodo derecho en infijo.

Posfijo. Recorrer en Posfijo el nodo izquierdo, recorrer el nodo derecho en Posfijo y aplicar el operador

Prefijo. Aplicar el operador, recorrer el nodo izquierdo en Prefijo y recorrer el lado derecho en prefijo.

Si utilizamos el árbol para evaluar la expresión sólo tenemos que recorrerlo en posfijo y utilizar las técnicas de evaluación que usamos en el evaluador de pilas.

Además, podemos aplicar las reglas de derivación al árbol de modo natural, como lo indican las reglas de derivación, por ejemplo la derivada de la suma se aplicaría como sigue:

```
y' =  derivada( arbol.nodo_izquierdo) + derivada( arbol.nodo_derecho )
```

Y así para cada una de las reglas. Incluso, podemos construir la expresión de la derivada en otro árbol, ya sea que se requiera evaluar constantemente (y no necesitamos recalcular la derivada, sólo se evalúa el árbol de la misma) o se desea calcular derivadas sobre derivadas.

Por lo tanto, el primer paso es convertir la pila construida por el evaluador en un árbol.

```
function TFuncion.agregalarbol( Var p : TPLista ) : TPArbol;
{ Lee los valores de la Pila p y con ellos genera un árbol}
Var arbol : TParbol;
begin
  GetMem(arbol, SizeOf( TArbol ) );
  arbol^.info := p^.elemento ;
  arbol^.tipo := p^.tipo ;
  arbol^.izq  := Nil ;
  arbol^.der  := Nil ;
  arbol^.inicio := p^.inicio;
  arbol^.actual := p^.actual;
  p := p^.siguiente;

  if (arbol^.tipo = opOperador) then   (* Si no es número mete dos valores *)
  begin
   arbol^.der := agregalarbol( p ) ;
   arbol^.izq := agregalarbol( p ) ;
  end;

  if ( arbol^.tipo = opFuncion) then
  begin
      arbol^.der := agregalarbol( arbol^.inicio);
```

```
      arbol^.inicio := arbol^.actual ;
  end;

  result := arbol;
end;
```

Finalmente, desarrollamos un evaluador de una expresión en forma de árbol que nos servirá, tanto para evaluar una función escrita en notación infija por el usuario, como para evaluar la derivada de la misma

```
procedure TFuncion.EvaluaArbol( a : TPArbol; Var p : TPLista; x : double );
var pf : TPLista;
begin
 if (a <> Nil) then
 begin
     if (a^.tipo = opFuncion) then
     begin
        pf := Nil;
        EvaluaArbol( a^.der, pf, x );
        y.num := evalfun_interna( a^.info.oper, pf^.elemento.num );
        push( y, opConstante, p);
        FreeMem(pf, SizeOf(TLista));
        exit;
     end;

     EvaluaArbol( a^.izq, p, x ) ;
     EvaluaArbol( a^.der, p, x ) ;

     if (a^.tipo = opConstante) then
      push( a^.info, opConstante, p );

     if (a^.tipo = opVariable) then
     begin
        y.num := x;
        push( y, opConstante, p );
     end;

     if (a^.tipo = opOperador) then
     begin
        pop( c, t, p );
        pop( b, t, p );
        if (a^.info.oper = '+') then y.num := b.num+c.num;
        if (a^.info.oper = '-') then y.num := b.num-c.num;
        if (a^.info.oper = '*') then y.num := b.num*c.num;
        if (a^.info.oper = '/') then
           if (c.num <> 0.0) then y.num := b.num/c.num
           else raise EErrorLeerFuncion.Create('División entre cero');
        if (a^.info.oper = '^') then
         begin
            if (b.num>0.0) then y.num := exp( c.num * ln(b.num) );

             if (b.num<0.0) then
             begin
               if (c.num - int(c.num)) = 0 then      (* c.num es entero *)
                begin
                     if (trunc(c.num) mod 2 ) = 0.0 then  (* si es par   *)
                     y.num := exp( c.num * ln( abs(b.num)) )
                     else
                     y.num := -exp( c.num * ln( abs(b.num)) );
```

```
                end
                else raise EErrorLeerFuncion.Create('No se puede elevar un número
negativo a una potencia racional');
             end;

             if (b.num = 0) then
             begin
                if (c.num > 0.0) then y.num := 0.0
                else raise EErrorLeerFuncion.Create('No se puede elevar Cero a una
potencia menor o igual a Cero');
             end;
          end;
          push( y , opConstante, p );
       end;
 end;
end;
```

Ahora, sólo necesitamos aplicar las reglas de la derivada a nuestro árbol y simplemente evaluarlo para tener, de esta forma, un resultado preciso sobre el valor de la derivada obviamente mejor que utilizando un método de derivadas numéricas.

4.3.4 Listado del componente para Evaluar una Función escrita en infijo, y su derivada

Propiedades

Funcion. Es un componente del tipo Edit estándar de Delphi para leer cadenas, aquí podrá el usuario escribir en tiempo real una función para ser evaluada.

Métodos

ValidarFuncion. Cuando el usuario ha indicado que ha terminado de escribir la función, entonces es necesario validarla y prepararla para sus posteriores evaluaciones.

AsignarFuncion. Si no se desea utilizar la propiedad Funcion, se puede asignar una cadena por medio de este método.

EvaluaFuncion. Dado un valor x, este procedimiento regresa el valor de la función escrita por el usuario evaluada en ese punto.

EvaluaDerivada. Dado un valor x, este procedimiento regresa el valor de la derivada de la función escrita por el usuario evaluada en ese punto.

Listado

```
unit LeerFunciones;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type

  {Elementos que forman una expresión (los Terminales si hablamos de producciones)}
  TOperandos = (opConstante, opVariable, opOperador, opFuncion);

  TElemento = record
    case boolean of
        True:  (num : Double);
        False: (oper: Char);
  end;

  TPLista = ^TLista ;
  TLista = record
           elemento : TElemento;
           tipo : TOperandos;
           inicio, siguiente, actual : TPLista;
  end;

  EErrorLeerFuncion = class(Exception);

  TArbol = ^TArbol;
  TArbol = Record
          izq, der : TPArbol;
          info : TElemento ;
          tipo : TOperandos;
          inicio, actual : TPLista;
  end;

  TFuncion = class(TComponent)
  private
     Finicio, Factual : TPLista;
     Faf, Fad : TPArbol;
     Ffx, Ffpx: TPlista; {Son las pilas que contienen la función y su derivada}
     FFuncion : TEdit;
     procedure Ligar( y : TElemento; t : TOperandos; Var inicio,
  actual : TPLista);
     function  IdentificaFuncion( fun : String ) : Char;
     function  EsOperador( d : Char ) : Boolean;
     procedure MeterFuncion( cad : String; Var ap_ini, ap_actual : TPLista );
     procedure push( y : TElemento; t : TOperandos; Var tope : TPLista);
     function  prioridad( a : Char ) : Integer;
     procedure pop( Var y : TElemento; Var t : TOperandos; Var tope : TPLista);
     procedure posfija( Var q, inicio, actual : TPLista);
     function  agregalarbol( Var p : TPLista ) : TPArbol;
     function  evalfun_interna( fn : Char; valor :  double ) : double;
     procedure EvaluaArbol( a : TPArbol; Var p : TPLista; x : double );
     procedure RecorreArbolEnPosfijo( arbol : TPArbol; Var p : TPLista );
     procedure fprima( arbol : TPArbol; Var d : TPLista );
     function  Derivar( r : TPLista) : TPLista;
     function  FuncionEvaluar( fx : TPArbol; v : double ) : Double;
     procedure libarbol( a : TPArbol );
     procedure liberarapunt(Var inicio, actual, fun, der : TPLista;
```

```
  Var af, ad : TPArbol);
  public
    constructor Create(AOwner : TComponent); override;
    procedure ValidarFuncion;
    procedure AsignarFuncion( as_func : String );
    function  EvaluaFuncion( x : double ) : Double;
    function  EvaluaDerivada( x : double ) : Double;
  published
    property Funcion : TEdit read FFuncion write FFuncion ;
  end;

procedure Register;

implementation

constructor TFuncion.Create(AOwner : TComponent);
begin
   inherited Create(AOwner);
   Faf := Nil;
   Fad := Nil;
   Ffx := Nil;
   Ffpx := Nil;
   FFuncion.Text := 'x+1';
end;
(****************** Interpretación de la cadena de lectura ******************)

procedure TFuncion.Ligar( y : TElemento; t : TOperandos; Var inicio, actual :
TPLista);
var aux : TPLista;
begin
  GetMem(aux, SizeOf( TLista ) );
  aux^.siguiente := Nil;
  aux^.tipo := t;
  aux^.inicio := Nil ;
  aux^.actual := Nil ;
  aux^.elemento := y;

  if inicio = Nil then
  begin
     inicio := aux;
     actual := aux
  end
  else
  begin
    actual^.siguiente := aux;
    actual := aux;
  end;
end;

function TFuncion.IdentificaFuncion( fun : String ) : Char;
begin
  fun := UpperCase(fun);
  Result := ' ';
  if ( fun = 'SEN') then Result := 's';
  if ( fun = 'COS') then Result := 'c';
  if ( fun = 'E')   then Result := 'e';
  if ( fun = 'LN')  then Result := 'l';
  if ( fun = 'TAN') then Result := 't';

  {Cualquier otra cosa es un error}
  if Result = ' ' then
    raise  EErrorLeerFuncion.Create('Se  detectó  un  nombre  de  función  interna  no
definida');
```

```
end;

function TFuncion.EsOperador( d : Char ) : Boolean;
begin
 if ( (d='+') or (d='-') or (d='*') or (d='/') ) or
    ( (d='^') or (d='(') or (d=')') ) Then Result := True
 else
     Result := False;
End;

procedure TFuncion.MeterFuncion( cad : String; Var ap_ini, ap_actual : TPLista );
{ Dada una función escrita en notación posfija, este procedimiento la convierte en una
lista ligada guardando propiedades adicionales}
Var primer_num, primer_car, negativo : Boolean;
    anterior, e, f : Char;
    aux, ext, otra : String;
    i, cp, cont_elem, cparent : Integer;
    y : TElemento;

    lp_inicio, lp_actual : TPLista ;
begin
   primer_num := True;
   primer_car := True;
   negativo   := False;
   anterior   := '@';
   cparent    := 0 ;

   i:=1; cont_elem:=0;

   lp_actual := ap_actual;
   lp_inicio := ap_ini;

   while( i <= Length(cad) ) do
   begin
       e := UpCase(cad[i]);

       if ( Not (e in ['A'..'Z', '0'..'9'])) and  {Si no es un número o letra}
          ( Not EsOperador(e) ) and               {Si no es un operador}
          ( Not (e in ['.', ' '])) Then           {Si no es el punto el espacio}
          raise EErrorLeerFuncion.Create('Se detectó un carácater No válido');

       if (e in ['A'..'Z']) and (e <> 'X') Then {Leer funciones}
       begin
            ext := ext + e;
            primer_car := False;
            Inc(i);
            continue ;
       end;

       if (e in ['0'..'9', '.'] ) Then     {Leer números}
       begin
            aux := aux + e;
            primer_num := False;
            {Inc(i);}
            {continue ; {Originalmente no iba pero igual y jala}
       end;

       if ( (e='X') or EsOperador(e) ) Then
       begin
           if ( ( e = '-') and         {Se valida el elemento sea negativo}
                ( (anterior = '@') or
                  (anterior = '^') or
                  (anterior = '(') ) ) Then   (* el menos entra abajo *)
```

```
        begin
            Inc(cont_elem);
            y.oper := '(' ;                          (* insertar un ( 0 - n ) *)
            Ligar(y, opOperador, lp_inicio, lp_actual);
            y.num := 0.0 ;
            Ligar(y, opConstante, lp_inicio, lp_actual);
            negativo := True ;
            anterior := ' ';
        end;

        if ( ((anterior = 'X') or (anterior = 'f') or (anterior = ')'))
           and ((e = 'X') or (e = '(')) ) then  {  agregar un *  }
        begin  {Forzar la multiplicación de implícita a explícita}
            y.oper := '*';
            Ligar(y,opOperador, lp_inicio, lp_actual);
            Inc(cont_elem);
        end;

        if ( Not primer_car) and (e<>'(') then
        {si se leyó una función y no hay paréntesis}
            raise EErrorLeerFuncion.Create('Función sin paréntesis abierto');

        if (Not primer_num) then  {Si se leyó algún número}
        begin
            try
              y.num := StrToFloat(aux);
            except
              raise EErrorLeerFuncion.Create(aux + ' no es un número válido');
            end;
            Ligar(y, opConstante, lp_inicio, lp_actual); {Insertar el número}
            Inc(cont_elem);

            aux := ''; {Preparar para el siguiente número}

            if ( negativo ) then                 (* cerrar el ( 0 - n ) *)
            begin
                y.oper := ')' ;
                Ligar(y, opOperador, lp_inicio, lp_actual);
            end;

            if ( (e = 'X') or (e = '(') ) then
            {agregar una multiplicación explicita i.e. 2x => 2 * x}
            begin
                y.oper := '*';
                ligar(y, opOperador, lp_inicio, lp_actual);
                Inc(cont_elem);
            end;

            primer_num := True;
            negativo := False;
        end;

        if (Not primer_car) and (e = '(') then (* Si se leyó una función *)
        begin
            f := IdentificaFuncion( ext );

            cp := 1; otra := '';
            Inc(i);
            e := UpCase(cad[i]);

(* capturar el argumento *)
            while((cp >= 1) and ( i<=length(cad) )) do
            begin                                    (* de la función *)
```

```
                    otra := otra + e;
                    Inc(i);
                    e := UpCase(cad[i]);
                    if (e = '(') then Inc(cp); {contando los paréntesis}
                    if (e = ')') then Dec(cp);
                end;

                if ( cp = 0 ) then       (* si se cerraron todos los parentesis *)
                begin                    (* incluido el de la funci¢n *)
                    y.oper := f;
                    (* Hacer una llamada recursiva para el argumento *)
                    Ligar( y, opFuncion, lp_inicio, lp_actual );

                    MeterFuncion( otra, lp_actual^.inicio, lp_actual^.actual );
                    primer_car := True; ext := ''; {Listo para leer otra función}

                    Inc(cont_elem) ;
                    anterior := 'f';

                    if ( negativo ) then   { por si se antepuso un signo (-) }
                    begin
                      y.oper := ')' ;
                      Ligar(y,opOperador, lp_inicio, lp_actual);
                      negativo := False ;
                    end;

                    Inc(i);
                    continue ;
                end
                else
                    raise EErrorLeerFuncion.Create('Falta cerrar un paréntesis en el
argumento de la función');

            end; { de si se leyó una función}

            if ( e = '(' ) then Inc(cparent);     (* contar los parentesis *)
            if ( e = ')' ) then Dec(cparent);

            y.oper := e;
            if (e = 'X') then
               Ligar(y, opVariable, lp_inicio, lp_actual )
            else
               Ligar(y, opOperador, lp_inicio, lp_actual );

            Inc(cont_elem);

            if (EsOperador(anterior)) and (Not (anterior in ['(',')'])) and
               (Not (e in ['(', ')', 'X'] ) ) then
               raise EErrorLeerFuncion.Create('Se encontró un doble operador');

        end;                              (* fin del Sí operador *)

        if ( e <> ' ') then anterior := e;

        Inc(i);
    end;                              (* fin del while *)

  if (Not primer_num) and (anterior <> '@') then (* por si se quedo leyendo un número
*)
  begin
        try
          y.num := StrToFloat(aux);
        except
```

```
        raise EErrorLeerFuncion.Create(aux + ' no es un número válido');
      end;
      Ligar(y, opConstante, lp_inicio, lp_actual);

      Inc(cont_elem);
      if ( negativo ) Then  {cerrar el ( 0-n }
      begin
          y.oper := ')' ;
          Ligar(y,opOperador, lp_inicio, lp_actual);
      end;
  end;

  if (Not primer_car) then (* por si se quedo leyendo una función *)
     raise EErrorLeerFuncion.Create('Se encontró una función sin argumentos');

  if (cont_elem mod 2) = 0 then (* hay un n£mero par de elementos (i.e 1- *)
     raise EErrorLeerFuncion.Create('La expresión es inválida, falta un operando');

  if ( cparent <> 0 ) then    (* hay par,ntesis abiertos *)
     raise EErrorLeerFuncion.Create('Los paréntesis no están balanceados');

  ap_ini    := lp_inicio;
  ap_actual := lp_actual;
end;

 (****************** Notación posfija de la cadena interpretada **************)

procedure TFuncion.push( y : TElemento; t : TOperandos; Var tope : TPLista);
var
  aux : TPLista;
begin
  GetMem(aux, SizeOf( TLista ) );
  aux^.siguiente := tope;
  aux^.tipo := t;
  aux^.elemento := y;
  aux^.inicio := Nil ;
  aux^.actual := Nil ;
  tope := aux;
end;

function TFuncion.prioridad( a : Char ) : Integer;
begin
  Result := -1;
  if ((a = '(') or (a = ')')) then Result := 4;
  if  (a = '^')               then Result := 3;
  if ((a = '*') or (a = '/')) then Result := 2;
  if ((a = '+') or (a = '-')) then Result := 1;
end;

procedure TFuncion.pop( Var y : TElemento; Var t : TOperandos; Var tope : TPLista);
var
  aux : TPLista;
begin

 if (tope = Nil) Then exit; (* esta vacia *)

 y := tope^.elemento;
 t := tope^.tipo;

 aux := tope^.siguiente;

 FreeMem( tope, SizeOf( TLista ) );
 tope := aux;
```

```
end;

procedure TFuncion.posfija( Var q, inicio, actual : TPLista);
{regresa la expresión posfija de actual en q}
Var op, n : Char;
    t     : TOperandos;
    y     : TElemento;
    e     : TPLista;
    r, p  : TPLista;

    {útil para las funciones}
    ex : TPLista;
    z  : TElemento;
    m  : TOperandos;

begin
   r := Nil; p := Nil; ex := Nil;
   e := inicio;

   y.oper := '(';
   push( y, opOperador, p );

   y.oper :=')'; Ligar(y, opOperador, inicio, actual);

   while (p <> Nil) do
   begin
       if ( ((e^.tipo) = opConstante) or ( (e^.tipo) = opVariable) ) then
          push( e^.elemento, e^.tipo, r );
       if ( (e^.tipo) = opFuncion ) then
       begin
           push( e^.elemento, e^.tipo, r );
           posfija( ex, e^.inicio, e^.actual);
           while( e^.inicio <> Nil ) do pop( z, m, e^.inicio) ;
           r^.inicio := ex;
           r^.actual := ex;  (* apunta a la pila de notacion de la función *)
       end;
       if ( (e^.tipo) = opOperador ) then
       begin
           op := e^.elemento.oper;

           if (op = '(') then push( e^.elemento, e^.tipo, p );

           if ( (op <> '(') and (op <> ')') ) then
           begin
              n := p^.elemento.oper;

              while( (n <> '(') and ( prioridad(op) <= prioridad(n) ) ) do
              begin
                  pop( y, t, p );
                  push( y,  t, r );
                  n := p^.elemento.oper;
              end;
              push( e^.elemento, e^.tipo, p );
           end;

           if ( op = ')' ) then
           begin
              while( (p^.elemento.oper) <> '(' ) do
              begin
                 pop( y, t, p );
                 push( y,  t, r );
              end;
              pop( y, t, p );  (* vaciarlo *)
```

```
            end;
        end;
        e := e^.siguiente;
    end;
    q := r;
end;


(****************** Procedimientos con arboles ****************)

function TFuncion.agregalarbol( Var p : TPLista ) : TPArbol;
Var arbol : TParbol;
begin
  GetMem(arbol, SizeOf( TArbol ) );
  arbol^.info := p^.elemento ;
  arbol^.tipo := p^.tipo ;
  arbol^.izq  := Nil ;
  arbol^.der  := Nil ;
  arbol^.inicio := p^.inicio;
  arbol^.actual := p^.actual;
  p := p^.siguiente;

  if (arbol^.tipo = opOperador) then    (* Si no es número mete dos valores *)
  begin
   arbol^.der := agregalarbol( p ) ;
   arbol^.izq := agregalarbol( p ) ;
  end;

  if ( arbol^.tipo = opFuncion) then
  begin
      arbol^.der := agregalarbol( arbol^.inicio);
      arbol^.inicio := arbol^.actual ;
  end;

  result := arbol;
end;

function TFuncion.evalfun_interna( fn : Char; valor :  double ) : double;
{ Regresa el valor para la función predefinida }
begin
  if (fn = 's') then result := sin(valor)
  else if (fn = 'c') then result := cos(valor)
  else if (fn = 'e') then
    if (valor<=709.0) then result := exp(valor)
    else raise EErrorLeerFuncion.Create('Se evaluó la función exponencial más allá de
709')
  else if (fn = 'l') then
    if (valor > 0) then result := ln(valor)
    else raise EErrorLeerFuncion.Create('Se evaluó la función logaritmo en un número
negativo')
  else if (fn = 't') then
    try
       result := sin(valor) / cos(valor) ;
    except
       raise EErrorLeerFuncion.Create('Se evaluó la función tangente en un múltiplo de
PI')
    end
  else result := -1 ;
end;

Var y, b, c : TElemento;
    t       : TOperandos;
```

```
procedure TFuncion.EvaluaArbol( a : TPArbol; Var p : TPLista; x : double );
var pf : TPLista;
begin
 if (a <> Nil) then
 begin

     if (a^.tipo = opFuncion) then
     begin
        pf := Nil;
        EvaluaArbol( a^.der, pf, x );
        y.num := evalfun_interna( a^.info.oper, pf^.elemento.num );
        push( y, opConstante, p);
        FreeMem(pf, SizeOf(TLista));
        exit;
     end;

     EvaluaArbol( a^.izq, p, x ) ;
     EvaluaArbol( a^.der, p, x ) ;

     if (a^.tipo = opConstante) then
      push( a^.info, opConstante, p );

     if (a^.tipo = opVariable) then
     begin
        y.num := x;
        push( y, opConstante, p );
     end;

     if (a^.tipo = opOperador) then
     begin
        pop( c, t, p );
        pop( b, t, p );
        if (a^.info.oper = '+') then y.num := b.num+c.num;
        if (a^.info.oper = '-') then y.num := b.num-c.num;
        if (a^.info.oper = '*') then y.num := b.num*c.num;
        if (a^.info.oper = '/') then
           if (c.num <> 0.0) then y.num := b.num/c.num
           else raise EErrorLeerFuncion.Create('División entre cero');
        if (a^.info.oper = '^') then
         begin
            if (b.num>0.0) then y.num := exp( c.num * ln(b.num) );

              if (b.num<0.0) then
              begin
                if (c.num - int(c.num)) = 0 then      (* c.num es entero *)
                 begin
                      if (trunc(c.num) mod 2 ) = 0.0 then  (* si es par   *)
                      y.num := exp( c.num * ln( abs(b.num)) )
                      else
                      y.num := -exp( c.num * ln( abs(b.num)) );
                 end
                 else raise EErrorLeerFuncion.Create('No se puede elevar un número
negativo a una potencia racional');
              end;

              if (b.num = 0) then
              begin
                 if (c.num > 0.0) then y.num := 0.0
                 else raise EErrorLeerFuncion.Create('No se puede elevar Cero a una
potencia menor o igual a Cero');
              end;
         end;
        push( y, opConstante, p );
```

```
    end;
 end;
end;

procedure TFuncion.RecorreArbolEnPosfijo( arbol : TPArbol; Var p : TPLista );
(* Recorre el árbol en posfijo y lo coloca en la pila *)
begin
  if (arbol <> Nil) then
  begin
     if (arbol^.tipo = opFuncion) then
     begin
         push(arbol^.info, arbol^.tipo, p );  (* ya está en posfijo *)
         p^.actual := arbol^.actual;  {(*p) -> act = arbol -> act;}
         exit;
     end;
     RecorreArbolEnPosfijo( arbol^.izq, p ) ;
     RecorreArbolEnPosfijo( arbol^.der, p ) ;
     push(arbol^.info, arbol^.tipo, p );
  end;
end;


procedure TFuncion.fprima( arbol : TPArbol; Var d : TPLista );
(* Calcula la derivada de la raíz del árbol con respecto a x y deja la fórmula de la
derivada en la pila d*)
Var
   deriv : TPLista;
   extra,y : TElemento;
   t : Char;
   s : TOperandos;
begin
  deriv := d;

  s := arbol^.tipo;
  if ((s = opConstante) or (s = opVariable)) then
  begin
      if (s = opVariable) then
          extra.num := 1.0
      else
          extra.num := 0.0;
      push(extra,opConstante,deriv);
  end;

  if (s = opOperador) then
  begin
     extra := arbol^.info ;
                                    (* Los resultados se escriben en posfijo *)
     t := extra.oper;
     if ( (t = '+') or (t = '-') ) then
     begin
         fprima( arbol^.izq, deriv );
         fprima( arbol^.der, deriv );
         push( extra, opOperador, deriv );
     end;

     if ( t = '*' ) then
     begin
         fprima( arbol^.izq, deriv );    { (uv)' = u'v + v'u }
         RecorreArbolEnPosfijo( arbol^.der, deriv );
         push( extra, opOperador, deriv );

         RecorreArbolEnPosfijo( arbol^.izq, deriv );
         fprima( arbol^.der, deriv );
```

```
        push( extra, opOperador, deriv );

        y.oper := '+';
        push( y, opOperador, deriv );
    end;

    if ( t = '/') then          { y = u/v ; y' = ( u'v - v'u ) / v^2 }
    begin                       { En posfijo :   u'v * v'u * - v 2 ^ / }
        y.oper := '*';
        fprima( arbol^.izq, deriv );
        RecorreArbolEnPosfijo( arbol^.der, deriv );
        push( y, opOperador, deriv );

        RecorreArbolEnPosfijo( arbol^.izq, deriv );
        fprima( arbol^.der, deriv );
        push( y, opOperador, deriv );

        y.oper := '-'; push( y, opOperador, deriv );

        RecorreArbolEnPosfijo( arbol^.der, deriv );
        y.num  := 2.0; push( y, opConstante, deriv );
        y.oper := '^'; push( y, opOperador,  deriv );

        push( extra, opOperador, deriv );
    end;

    if (t = '^') then       { y = u^n ; y' = ( n * u ^ (n - 1) ) * u' }
    begin                   { En posfijo :    n u n 1 - ^ * u' *  }
       RecorreArbolEnPosfijo( arbol^.der, deriv );   {n}
       RecorreArbolEnPosfijo( arbol^.izq, deriv );   {u}
       RecorreArbolEnPosfijo( arbol^.der, deriv );   {n}
       y.num  := 1.0; push( y, opConstante, deriv ); {1}
       y.oper := '-'; push( y, opOperador,  deriv ); {-}
       y.oper := '^'; push( y, opOperador,  deriv ); {^}
       y.oper := '*'; push( y, opOperador,  deriv ); {*}
       fprima( arbol^.izq, deriv );                  {u'}
       y.oper := '*'; push( y, opOperador,  deriv ); {*}
    end;
end;

if ( s = opFuncion) then
begin
    extra := arbol^.info ;
    t := extra.oper;

    if ((t = 's') or (t = 'c')) then    { y = sen( u ); y' = cos(u)*u'; cos(u)u'* }
    begin
        if (t = 'c') then
        begin
            y.num := 0.0; push( y, opConstante, deriv );
        end;

        if t = 's' then extra.oper := 'c'
        else            extra.oper := 's' ;
        push( extra, opFuncion, deriv );

        deriv^.inicio := arbol^.inicio;
        deriv^.actual := arbol^.actual;
        fprima( arbol^.der, deriv );
        y.oper := '*'; push( y, opOperador, deriv );

        if (t = 'c') then       { y = cos( u ); y'= (0-sen(u))*u'; 0sen(u)u'*- }
        begin
```

```
            y.oper := '-'; push( y, opOperador, deriv );
        end;
    end;

    if (t = 't') then  { y = tan(u); y'= u'/(cos(u)^2); u'cos(u)2^/ }
    begin
        fprima(arbol^.der, deriv );        {u'}
        extra.oper := 'c';                 {cos}
        push( extra, opFuncion, deriv );

        deriv^.inicio := arbol^.inicio;
        deriv^.actual := arbol^.actual;    {(u)}
        y.num  := 2.0; push( y, opConstante, deriv ); {2}
        y.oper := '^'; push( y, opOperador,  deriv ); {^}
        y.oper := '/'; push( y, opOperador,  deriv ); {/}
    end;

    if (t = 'e') then { y = e(u); y' = u'e(u);  u'e(u)* }
    begin
        fprima(arbol^.der, deriv );
        push( extra, opFuncion, deriv );
        deriv^.inicio := arbol^.inicio;
        deriv^.actual := arbol^.actual;
        y.oper := '*'; push( y, opOperador, deriv );
    end;

    if (t = 'l') then  {* y = ln(u); y' = u'/u;  u'u/ }
    begin
        fprima(arbol^.der, deriv );
        RecorreArbolEnPosfijo( arbol^.der, deriv );
        y.oper := '/'; push( y, opOperador, deriv );
    end;
  end;

  d := deriv;
end;

function TFuncion.Derivar( r : TPLista) : TPLista;
{ Dada una pila r, regresa la función derivada de r como otra pila}
var deriv : TPLista;
    a : TParbol ; {Falta ponerlo Nil}
begin
  deriv := Nil;
  a := agregalarbol( r ) ; {Primero se convierte en un árbol}
  fprima( a, deriv ) ;     {Se deriva el árbol}
  result := deriv;
end;

function TFuncion.FuncionEvaluar( fx : TPArbol; v : double ) : Double;
Var q : TPLista;
begin
  q := Nil;
  EvaluaArbol( fx, q, v );
  if q <> Nil Then
  begin
    result := q^.elemento.num;
    FreeMem(q, SizeOf(TArbol));
  end
  else result := 0;
end;

procedure TFuncion.libarbol( a : TPArbol );
begin
```

```
 if (a <>Nil ) then
    begin
     libarbol( a^.izq );
     libarbol( a^.der );
     FreeMem(a, SizeOf(TArbol));
    end;
end;

procedure TFuncion.liberarapunt(Var inicio, actual, fun, der : TPLista; Var af, ad :
TPArbol);
var y : TElemento;
    t : Toperandos;
begin

 while(inicio<> Nil) do pop( y, t, inicio ); actual := Nil;
 while(fun<>Nil) do pop( y, t, inicio );
 while(der<>Nil) do pop( y, t, inicio );

 libarbol( af ); libarbol( ad );
 af := Nil; ad := Nil;
end;

//************************ Métodos

procedure TFuncion.AsignarFuncion( as_func : String );
{ Se puede asignar una cadena directamente al componente por sin no se
  desea utilizar la propiedad edit }
begin
     LiberarApunt(Finicio, Factual, Ffx, Ffpx, Faf, Fad);
     MeterFuncion( as_func, Finicio, Factual);
     posfija( Ffx, Finicio, Factual );
     Ffpx := derivar( Ffx );
     Faf  := agregalarbol( Ffx );
     Fad  := agregalarbol( Ffpx );
end;


procedure TFuncion.ValidarFuncion;
{ Verifica que la función escrita en la propiedad Funcion (de tipo TEdit)
  contenga una función válida, y prepara las variables que almacenarán la
  función y su derivada }
begin
  if assigned( FFuncion ) then
     AsignarFuncion( FFuncion.Text )
  else
     raise EErrorLeerFuncion.Create('Necesita asignar un componente TEdit, o utilice
AsignarFuncion para validar una cadena');
end;


function TFuncion.EvaluaFuncion( x : double ) : Double;
{ Regresa el valor de la función que definió el usuario evaluada en x }
begin
   result := FuncionEvaluar( Faf, x );
end;

function TFuncion.EvaluaDerivada( x : double ) : Double;
{ Regresa el valor de la derivada de la función que definió el usuario evaluada en x }
begin
   result := FuncionEvaluar( Fad, x );
end;
```

```
procedure Register;
begin
  RegisterComponents('MAC', [TFuncion]);
end;

end.
```

Capítulo 5

Integración Numérica

La integración numérica, también llamada *cuadratura*, tiene una historia que se remonta hasta la invención del cálculo. El hecho de que las integrales de funciones elementales no puedan resolverse analíticamente, generalmente, mientras que las derivadas si se puede, sirvió para que durante los siglos XVIII y XIX se abriera el campo para el análisis numérico en ésta área.

La cuadratura es simplemente el caso especial de evaluar la integral

$$I = \int_a^b f(x)dx$$

Los métodos de cuadratura en este capítulo están basados, de un modo o de otro, en el mecanismo obvio de agregar el valor del integrando a una secuencia de abscisas dentro del rango de integración. El objetivo es obtener la integral con tanta precisión como sea posible y con el menor número posible de evaluaciones de la función del integrando. Del mismo modo que en el caso de la interpolación, uno tiene la libertad de escoger métodos de varios *ordenes*, algunas veces con mayor orden dando, pero no siempre, mayor precisión. "La integración por el método de Romberg", que se discutirá en el apartado 5.3, es un formalismo general para hacer uso de métodos de integración de una variedad de diferentes ordenes.

Además de los métodos de éste capítulo existen otros métodos para obtener integrales. Una clase importante está basada en la aproximación de funciones, como el uso de la aproximación de Chebyshev (la cuadratura "Clenshaw-Curtis").

Algunas integrales relacionadas con las transformadas de Fourier pueden calcularse con el algoritmo de la rápida transformada de Fourier.

Las Integrales multidimensionales son otro conjunto de problemas que pueden ser atacados usando la importante técnica de la integración de Monte-Carlo.

5.1 Fórmulas clásicas para abscisas igualmente espaciadas

Las fórmulas clásicas para integrar una función cuyo valor se obtiene de intervalos igualmente espaciados tienen algo de elegancia, y permanecen gracias a su asociación histórica. Sin embargo, los tiempos cambian; con la excepción de las dos más modestas fórmulas ("la regla trapezoidal extendida", y "la regla de punto medio extendida") las funciones clásicas ya casi no se utilizan.

Deseamos integrar la función *f(x)* entre un límite inferior *a* y un límite superior *b*. Una fórmula de integración que use los valores de la función en los puntos límite, *f(a)* o *f(b)*, es llamada una fórmula *cerrada*. Ocasionalmente, deseamos integrar una función cuyo valor en uno o ambos de los límites es difícil de calcular (p. e., el cálculo de *f* se va al límite de cero sobre cero ahí, o peor aún tiene una singularidad en ese lugar). En este caso necesitamos una fórmula *abierta*, que estime la integral usando sólo valores entre *a* y *b* pero sin tomar estos puntos.

Los bloques básicos para construir funciones clásicas son reglas para integrar una función sobre un pequeño número de intervalos. Del mismo modo que el número se incrementa, podemos encontrar reglas que son exactas para polinomios de un creciente orden mayor. (Tenga en cuenta que un mayor orden no siempre implica mayor precisión en casos reales). A continuación se presenta una secuencia de fórmulas cerradas.

Fórmulas cerradas Newton-Cotes

Regla Trapezoidal:

$$\int_{x1}^{x2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] + O(h^3 f'') \qquad 5.1.1$$

Aquí el término error O() significa que la respuesta correcta difiere de la estimada por una cantidad que es el producto de algún coeficiente numérico h^3 veces el valor de la segunda derivada de la función en algún lugar del intervalo de integración. El coeficiente puede conocerse, y puede encontrarse en todas las referencias estándar de este tópico. El punto en el cual la segunda derivada debe evaluarse es, sin embargo, desconocido. Si lo conociéramos, podríamos evaluar la función ahí y tendríamos un método de mayor orden. Dado que el producto de algo conocido por algo desconocido es desconocido, continuemos con las fórmulas y escribamos sólo O(), en lugar del coeficiente.

La ecuación (5.1.1) es una fórmula de dos puntos (x_1 y x_2). Es exacta para polinomios hasta grado igual a 1, p. e. *f(x)* = *x*. Podemos anticipar que hay una fórmula de tres puntos exacta hasta 2° grado. Esto es verdad, pero además, por una cancelación de coeficientes debido a la simetría izquierda derecha de la fórmula, ésta es exacta para polinomios hasta de 3er. grado, p. e. $f(x) = x^3$:

Regla de Simpson

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] + O(h^5 f^{(4)}) \qquad 5.1.2$$

Aquí $f^{(4)}$ significa la cuarta derivada de la función f evaluada en un punto desconocido dentro del intervalo. Note también que la fórmula da la integral sobre un intervalo de tamaño 2h, así que los coeficientes se duplican.

Fórmulas Cerradas Extendidas

Si usamos la ecuación (5.1.1) N-1 veces, para calcular la integral en los intervalos (x_1, x_2), (x_2, x_3),..., (x_{N-1}, x_N), y después sumamos los resultados, obtenemos una función "extendida" o "compuesta" para la integral de x_1 a x_N.

Regla trapezoidal extendida:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{N-1} + \frac{1}{2}f_N\right] + O\left(\frac{(b-a)^3 f''}{N^2}\right) \qquad 5.1.3$$

Aquí escribimos la estimación del error en términos del intervalo b – a y el número de puntos N en lugar de en términos de h.

5.2 Algoritmos Elementales

Nuestro punto de partida es la ecuación (5.1.3), la regla trapezoidal extendida. Hay dos hechos importantes acerca de la regla trapezoidal que la hace el comienzo de una variedad de algoritmos. El primero parece obvio, mientras que el segundo es algo "oscuro".

El hecho obvio es que, para una función dada *f(x)* a ser integrada entre los límites *a* y *b*, podemos duplicar el número de intervalos en la regla trapezoidal extendida sin perder el beneficio del trabajo anterior. La implementación más cruda de la regla trapezoidal es promediar la función en sus puntos límite. El primer estado de refinamiento es agregar a este promedio el valor de la función en su punto medio. El segundo estado de refinamiento es agregar los valores de los puntos ¼ y ¾. Y así continuar.

Podemos escribir una rutina con esta lógica en mente:

```
function TIntegrar.trapzd( FUNC : TUserFunc; a, b : Double; n : integer) : Double;
{ Esta rutina calcula el n-ésimo estado de refinamiento de una regla trapezoidal
extendida. func es la función a ser integrada entre los límites a y b. Cuando es
```

```
llamada con n=1, la rutina regresa la estimación más cruda de Int(b,a, f(x))dx.
Subsecuentes llamadas con n=2,3,...  (en ese orden secuencial) mejorará la
aproximación agregando 2^(n-2) puntos interiores adicionales.}
var
   x,tnm,sum,del : Double;
   it,j : Integer;
begin
   if (n = 1) then
      FtrzS := 0.5*(b-a)*(FUNC(a)+FUNC(b))
   else begin
      it:=1;
      for j:=1 to n-2 do it := it shl 1;
      tnm := it;
      del := (b-a)/tnm; {Este es el espacio entre los puntos}
      x := a+0.5*del;
      sum := 0.0;
      for j:=1 to it do begin
          sum := sum + FUNC(x);
          x := x + del;
      end;
      FtrzS := 0.5*(FtrzS+(b-a)*sum/tnm); {Esto reemplaza Ftrz_S por su valor
refinado}
   end;
   result := FtrzS;
end;
```

La rutina anterior (trapzd) es una función de trabajo que puede manipularse de varias maneras. La más simple y cruda es la integral de una función por la regla trapezoidal extendida donde se conoce por anticipado el número de pasos deseados. Si desea 2^{M+1}, puede acompañarla con el siguiente fragmento

```
for j:=1 to m+1 do  s := trapzd( func, a, b, j);
```

con la respuesta calculada como s.

Mucho mejor, por supuesto, es refinar la regla trapezoidal hasta alcanzar algún grado de precisión deseado:

```
function TIntegrar.Itrap( func: TUserFunc; a, b : Double ) : Double;
{ Regresa la integral de la función FUNC de a a b. Los parametros EPSy JMAX pueden
ponerse en la precisión deseada, de tal forma que 2 al JMAX-1 potencia sea el número
máximo de pasos permitidos. La integración se realiza por la regla trapezoidal.}
var
   j : Integer;
   s,olds : Double;
begin
   olds := -1.0e30; {Cualquier número que no sea el promedio de la función en sus
puntos finales}
   for j:=1 to FJMAX do begin
       s:=trapzd(func,a,b,j);
       if (abs(s-olds) < EPS*abs(olds)) then begin
          result := s; exit;
       end;
       if (s = 0.0) and (olds = 0.0) and (j > 6) then begin
          result := s; exit;
       end;
       olds :=s;
   end;
```

```
   raise EErrorIntegracion.Create('La función de integración a excedido el límite de
iteraciones de refinamiento');
   result := 0.0; {Nunca llega aqui}
end;
```

Así de simple como se ve, la rutina Itrap es de hecho una forma robusta de realizar integrales de funciones que no son muy suaves, es decir, que tienen cambios inesperados en su trazado. Una sofisticación adicional, en general, la traducirá en un método de un orden mayor cuya eficiencia será mayor sólo para integrandos suficientemente suaves. Itrap, por ejemplo, es el método a elegir para un integrando que es una función de una variable linealmente interpolada entre puntos de datos medidos.

Ahora entramos a la parte "oscura" de la regla trapezoidal extendida. Esto es: el error de la aproximación, el cual comienza con un término de orden $1/N^2$ que viene dado directamente de la fórmula sumatoria de Euler-Maclaurin.

Supongamos que evaluamos (5.1.3) con N pasos, obteniendo un resultado S_N, y lo hacemos nuevamente con 2N pasos, obteniendo un resultado S_{2N}. (Esto se logra con dos llamadas cualquiera consecutivas de Itrap). El término de error restante en la segunda evaluación será de ¼ del tamaño del error en la primera evaluación. Por lo tanto la combinación

$$S = \frac{4}{3}S_{2N} - \frac{1}{3}S_N \qquad 5.2.1$$

cancelará el término del error restante. Pero no hay error de orden $1/N^3$. El error es de orden $1/N^4$, el mismo que para al regla de Simpson. A decir verdad, no le tomará mucho tiempo observar que (5.2.1) es exactamente la regla de Simpson, alternando 2/3's, y 4/3's. Este es el método preferido para evaluar esta regla, y podemos escribir una rutina análoga a Itrap:

```
function TIntegrar.Isimp( func : TUserFunc; a, b : Double ) : Double;
{ Regresa la integral de la función FUNC de a a b. Los parámetros EPSy JMAX pueden
ponerse en la precisión deseada, de tal forma que 2 a la JMAX-1 potencia sea el número
máximo de pasos permitidos. La integración se realiza por la regla de Simpson}
var
   j : Integer;
   s,st,ost,os : Double;
begin
   ost := -1.0e30; os := ost;
   for j:=1 to FJMAX do begin
       st := trapzd(func,a,b,j);
       s  := (4.0*st-ost)/3.0;   {regla de Simpson}
       if (abs(s-os) < EPS*abs(os)) then begin
          result := s; exit;
       end;
       if (s = 0.0) and (os = 0.0) and (j > 6) then begin
          result := s; exit;
       end;
       os  := s;
       ost :=st;
   end;
   raise EErrorIntegracion.Create('Se ha excedido el límite de iteraciones en la
Integración por la Regla de Simpson');
   result := 0.0; {Nunca llega aqui}
end;
```

La rutina Isimp será en general más eficiente que ITrap (p.e. requiere menos evaluaciones de la función) cuando la función a ser integrada tenga una cuarta derivada finita.

5.3 Integración por el método de Romberg

Podemos ver el método de Romberg como la generalización natural de la rutina Isimp de la sección anterior para esquemas de integración que son de orden más alto que la regla de Simpson. La idea básica es usar los resultados de k refinamientos sucesivos de la regla trapezoidal extendida (implementada en trapzd) para eliminar todos los términos en las series de error sin incluir $O(1/N^{2k})$. La rutina ISimp es el caso de k = 2. Este es un ejemplo de una idea muy general que lleva el nombre de *aproximación diferida al límite de Richardson*: algunos algoritmos numéricos para varios valores de un parámetro h, y después se extrapola el resultado al límite continuo h = 0.

La ecuación (5.2.1), que elimina el término de error restante, es un caso especial de extrapolación polinominal. En el caso más general de Romberg, podemos usar el algoritmo de Neville (vea §3.3.1) para extrapolar los refinamientos sucesivos hasta un intervalo cero. El algoritmo de Neville puede, de hecho, codificarse consistentemente dentro de una rutina de integración de Romberg. Por claridad del programa, sin embargo, es mejor hacer la extrapolación haciendo llamadas a InterpolarPolinomio, también dado en §3.3.1.

```
function TIntegrar.Iromb( func : TUserFunc; a, b : Double ) : Double;
{ Regresa la integral de la funcion FUNC de a a b. La integración se realiza por la el
método de Romberg de orden 2k, donde, por ejemplo, K=2 es la regla de Simpson.}
var
   ss,dss : Double;
   s, h : Variant;
   j : Integer;
begin
   { Aqui se almacenan aproximaciones trapezoidales sucesivas y sus intervalos
relativos}
   s := varArrayCreate([1,FJMAX+1], varDouble );
   h := varArrayCreate([1,FJMAX+2], varDouble );

   h[1]:=1.0;
   for j:=1 to FJMAX do begin
       s[j] := trapzd(func,a,b,j);
       if (j > FK) then begin
          MACTools.InterpolarPolinomio(h, j-FK, s, j-FK, FK, 0.0, ss, dss);
          if ( abs(dss) <= EPS*abs(ss)) then begin
             result := ss; exit;
          end;
       end;
       h[j+1]:=0.25*h[j];

{Este es la forma para calcular el siguiente intervalo. El factor es 0.25 dado
que el intervalo sólo se decrementa en 0.5. Esto hace que la extrapolación se
haga en un polinomio en h^2 como se indica en la ecuación 4.2.1, no sólo un
polinomio en h.}
```

```
    end;
    raise EErrorIntegracion.Create('Se ha excedido el límite de iteraciones en la
Integración por el Método de Romberg');
    result := 0.0; {Nunca llega aqui}
end;
```

La rutina IRomb, junto con sus procedimientos necesarios InterpolarPolinomio y trapzd, es suficientemente poderosa para integrandos suaves (p.e. analíticos), integrandos sobre intervalos que no contienen singularidades, y cuando los límites de integración también no son singularidades. IRomb, en tales circunstancias, toma menos evaluaciones de la función que para los métodos anteriores. Por ejemplo, la integral

$$\int_0^2 x^4 \log(x+\sqrt{x^2+1})dx$$

converge en la primera extrapolación, después de tan sólo cinco llamadas a trapzd, mientras que ISimp requiere de 8 llamadas (haciendo 8 evaluaciones más del integrando) e ITrap requiere 13 llamadas (haciendo 256 evaluaciones del integrando).

5.4 Listado de los componentes de Integración Cerrada

Tintegrar es el componente principal del cual se derivan los demás componentes de integración cerrada.

Propiedades

NumMaxIteraciones : Número máximo de iteraciones que realizará el procedimiento de integración, antes de emitir un error.

Eventos

OnUserFunc. Es en este evento donde el usuario escribe la función que desea integrar.

TintegrarTrapecio, TintegrarSimpson y TintegrarRomberg están heredados de Tintegrar, así que tienen tanto sus propiedades como sus eventos, sólo TintegrarRomberg tiene una propiedad adicional y es el Orden de integración, donde Orden = 2 implica la regla de Simpson.

Métodos

Integrar. Dados los límites de integración *a* y *b*, este procedimiento regresa el valor estimado de la integral de la función definida por el usuario.

Listado

```
unit IntegralesCerradas;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

const EPS = 1.0e-5;

type
  EErrorIntegracion = class(Exception);
  TUserFunc = function( x : Double ) : Double of Object;

// ----------------------------------------------------------------------------
// Iniciando la definición del Componente Básico para integrar
 TIntegrar = class(TComponent)
  private
  protected
    FtrzS : Double; {Permite el refinamiento sucesivo de la integración trapezoidal}
    FUserFunc : TUserFunc;
    FJMAX : Integer;
    FK : Integer;     {Número de puntos usados en la extrapolación del método de
Romberg, 2 = Simpson}
    function trapzd( FUNC : TUserFunc; a, b : Double; n : integer) : Double;
    function Itrap( FUNC : TUserFunc; a, b : Double ) : Double;
    function Isimp( func : TUserFunc; a, b : Double ) : Double;
    function Iromb( func : TUserFunc; a, b : Double ) : Double;
  public
    constructor Create(AOwner : TComponent); override;
  published
    property NumMaxIteraciones : integer  read FJMAX write FJMAX default 20;
    property OnUserFunc : TUserFunc read FUserFunc write FUserFunc;
  end; {TIntegrar}

// ----------------------------------------------------------------------------
// La integral por el método del trapecio hereda todas las propiedades de
// Tintegrar y sólo define la función a operar
  TIntegrarTrapecio = class(TIntegrar)
  private
  protected
  public
    function Integrar( a, b : Double ) : Double;
  published
  end;

// ----------------------------------------------------------------------------
// Definiendo el componente para Integrar por la regla de Simpson
  TIntegrarSimpson = class(TIntegrar)
  private
  protected
  public
```

```
    function Integrar( a, b : Double ) : Double;
  published
  end;

// ----------------------------------------------------------------------------
// Definiendo el componente para Integrar por el Método de Romberg
  TIntegrarRomberg = class(TIntegrar)
  private
  protected
  public
    constructor Create(AOwner : TComponent); override;
    function Integrar( a, b : Double ) : Double;
  published
      property Orden : integer  read FK write FK default 5;
  end;

procedure Register;

implementation

uses MACTools;

constructor TIntegrar.Create(AOwner : TComponent);
begin
   inherited Create(AOwner);
   FJMAX := 20;
end;

(*************************)
(* Métodos de integración*)
(*************************)

function TIntegrar.trapzd( FUNC : TUserFunc; a, b : Double; n : integer) : Double;
{ Esta rutina calcula el n-ésimo estado de refinamiento de una regla trapezoidal
extendida. func es la función a ser integrada entre los límites a y b. Cuando es
llamada con n=1, la rutina regresa la estimación más cruda de Int(b,a, f(x))dx.
Subsecuentes llamadas con n=2,3,...   (en ese orden secuencial) mejorará la
aproximación agregando 2^(n-2) puntos interiores adicionales.}
var
   x,tnm,sum,del : Double;
   it,j : Integer;
begin
   if (n = 1) then
      FtrzS := 0.5*(b-a)*(FUNC(a)+FUNC(b))
   else begin
      it:=1;
      for j:=1 to n-2 do it := it shl 1;
      tnm := it;
      del := (b-a)/tnm; {Este es el espacio entre los puntos}
      x := a+0.5*del;
      sum := 0.0;
      for j:=1 to it do begin
          sum := sum + FUNC(x);
          x := x + del;
      end;
      FtrzS := 0.5*(FtrzS+(b-a)*sum/tnm); {Esto reemplaza trz_S por su valor refinado}
   end;
   result := FtrzS;
end;

function TIntegrar.Itrap( func: TUserFunc; a, b : Double ) : Double;
```

```
{ Regresa la integral de la función FUNC de a a b. Los parámetros EPSy JMAX pueden
ponerse en la precisión deseada, de tal forma que 2 al JMAX-1 potencia sea el número
máximo de pasos permitidos. La integración se realiza por la regla trapezoidal.}
var
   j : Integer;
   s,olds : Double;
begin
   olds := -1.0e30; {Cualquier número que no sea el promedio de la función en sus
puntos finales}
   for j:=1 to FJMAX do begin
       s:=trapzd(func,a,b,j);
       if (abs(s-olds) < EPS*abs(olds)) then begin
          result := s; exit;
       end;
       if (s = 0.0) and (olds = 0.0) and (j > 6) then begin
          result := s; exit;
       end;
       olds :=s;
   end;
   raise EErrorIntegracion.Create('La función de integración a excedido el límite de
iteraciones de refinamiento');
   result := 0.0; {Nunca llega aqui}
end;

function TIntegrar.Isimp( func : TUserFunc; a, b : Double ) : Double;
{ Regresa la integral de la función FUNC de a a b. Los parámetros EPSy JMAX pueden
ponerse en la precisión deseada, de tal forma que 2 a la JMAX-1 potencia sea el número
máximo de pasos permitidos. La integración se realiza por la regla de Simpson}
var
   j : Integer;
   s,st,ost,os : Double;
begin
   ost := -1.0e30; os := ost;
   for j:=1 to FJMAX do begin
       st := trapzd(func,a,b,j);
       s  := (4.0*st-ost)/3.0;   {regla de Simpson}
       if (abs(s-os) < EPS*abs(os)) then begin
          result := s; exit;
       end;
       if (s = 0.0) and (os = 0.0) and (j > 6) then begin
          result := s; exit;
       end;
       os  := s;
       ost :=st;
   end;
   raise EErrorIntegracion.Create('Se ha excedido el límite de iteraciones en la
Integración por la Regla de Simpson');
   result := 0.0; {Nunca llega aqui}
end;

function TIntegrar.Iromb( func : TUserFunc; a, b : Double ) : Double;
{ Regresa la integral de la funcion FUNC de a a b. La integración se realiza por la el
método de Romberg de orden 2k, donde, por ejemplo, K=2 es la regla de Simpson.}
var
   ss,dss : Double;
   s, h : Variant;
   j : Integer;
begin
   { Aqui se almacenan aproximaciones trapezoidales sucesivas y sus intervalos
relativos}
   s := varArrayCreate([1,FJMAX+1], varDouble );
   h := varArrayCreate([1,FJMAX+2], varDouble );
```

```
   h[1]:=1.0;
   for j:=1 to FJMAX do begin
       s[j] := trapzd(func,a,b,j);
       if (j > FK) then begin
          MACTools.InterpolarPolinomio(h, j-FK, s, j-FK, FK, 0.0, ss, dss);
          if ( abs(dss) <= EPS*abs(ss)) then begin
             result := ss; exit;
          end;
       end;
       h[j+1]:=0.25*h[j];

{Este es la forma para calcular el siguiente intervalo. El factor es 0.25 dado
que el intervalo sólo se decrementa en 0.5. Esto hace que la extrapolación se
haga en un polinomio en h^2 como se indica en la ecuación 4.2.1, no sólo un
polinomio en h.}

   end;
   raise EErrorIntegracion.Create('Se ha excedido el límite de iteraciones en la
Integración por el Método de Romberg');
   result := 0.0; {Nunca llega aqui}
end;

// ---------------------------------------------------------------------------
// Implementado la Integración por el Método del Trapecio
function TIntegrarTrapecio.Integrar( a, b : Double ) : Double;
begin
     result := Itrap(FUserFunc, a, b);
end;

// ---------------------------------------------------------------------------
// Implementado la Integración por la Regla de Simpson
function TIntegrarSimpson.Integrar( a, b : Double ) : Double;
begin
     result := Isimp(FUserFunc, a, b);
end;

// ---------------------------------------------------------------------------
// Implementado la Integración por el Método de Romberg
constructor TIntegrarRomberg.Create(AOwner : TComponent);
begin
   inherited Create(AOwner);
   FK := 5;
end;

function TIntegrarRomberg.Integrar( a, b : Double ) : Double;
begin
     result := Iromb(FUserFunc, a, b);
end;

procedure Register;
begin
  RegisterComponents('MAC', [TIntegrarTrapecio, TIntegrarSimpson, TIntegrarRomberg]);
end;

end.
```

5.5 Integrales impropias

Para nuestros propósitos, una integral será "impropia" si tiene alguno de los siguientes problemas:

su integrando tiende a un valor finito en límites inferior y superior finitos, pero no puede ser evaluado precisamente en esos límites (p.e. sen(x) / x en x = 0)
su límite superior es ∞, o su límite inferior es $-\infty$
tiene una singularidad integrable en cualquiera de sus límites (p.e. $x^{-1/2}$ en x = 0)
tiene una singularidad integrable en un lugar entre sus límites superior e inferior
tiene una singularidad integrable en un lugar desconocido entre sus límites superior e inferior

Si una integral es infinita (p.e., $\int^{\infty} x^{-1}dx$), o no existe en un sentido de límite (p.e., $\int_{-\infty}^{\infty}\cos x dx$), no es llamada impropia, la nombramos imposible.

Con los procedimientos y técnicas descritos en esta sección puede encontrarse solución para los cuatro primeros problemas.

Necesitamos un procedimiento de partida como en la regla trapezoidal extendida, pero una que sea del tipo fórmula abierta, en el sentido de 5.1, es decir, el integrando no requiere ser evaluado en sus límites.

No es posible duplicar el número de pasos en la regla del punto medio extendida y aún tener el beneficio de las evaluaciones previas de la función. Sin embargo, es posible triplicar el número de pasos. En promedio, triplicar implica un factor de $\sqrt{3}$ de trabajo innecesario, dado que el número "correcto" de pasos para una precisión deseada sólo necesita $\sqrt{2}$, pero perdemos un factor extra de 2 al no poder usar las evaluaciones previas. Dado que $1.732 < 2 \times 1.414$, es mejor triplicar.

Esta es la rutina resultante, comparable directamente con la trapezoidal.

```
function TIntegralesAbiertas.PuntoMedio(FUNC : TUserFunc; a, b : Double; n : integer)
: Double;
{ Esta rutina calcula el n-ésimo estado de refinamiento de una regla de punto medio
extendida. func es la función a ser integrada entre los límites a y b. Cuando es
llamada con n=1, la rutina regresa la estimación más cruda de Int(b,a, f(x))dx.
Subsecuentes llamadas con n=2,3,...  (en ese orden secuencial) mejorará la
aproximación agregando (2/3) x 3^(n-1) puntos interiores adicionales.}
var
  x,tnm,sum,del,ddel : Double;
  it,j : Integer;
begin
     if (n = 1) then
        FmidS := (b-a)*FUNC(0.5*(a+b))
     else begin
        it:=1;
        for j:=1 to n-2 do it := it * 3;
        tnm:=it;
        del:=(b-a)/(3.0*tnm);
```

```
        ddel:=del+del; {Los puntos agregados están alternados entre el espacio de del
y ddel}
        x:=a+0.5*del;
        sum:=0.0;
        for j:=1 to it do begin
            sum := sum + FUNC(x);
            x := x + ddel;
            sum := sum + FUNC(x);
            x := x + del;
        end;
        FmidS := (FmidS+(b-a)*sum/tnm)/3.0; {La nueva suma se combina con la integral
previa para dar una integral refinada}
     end;
     result := FmidS;
end;
```

La rutina PuntoMedio puede reemplazarse exactamente a Trapezoidal en una rutina derivada como ITrap; un simple cambio Trapezoidal(func, a, b, j) por PuntoMedio(func, a, b, j) y quizá reducir el parámetro JMAX dado que 3^{JMAX-1} (de triplicar los pasos) es un número mucho mayor que 2^{JMAX-1} (duplicar los pasos).

La implementación de la fórmula abierta análoga a la regla de Simpson sustituye PuntoMedio a Trapezoidal y decrementa JMAX como se mencionó, pero ahora también se cambia el paso de extrapolación para ser

```
s := (9.0 * st - ost ) / 8.0;
```

dado que, cuando el número de pasos es triplicado, el error se decrementa a 1/9avo de su tamaño, no 1/4to como con los pasos duplicados.

Tanto la modificada ITrap o ISimp pueden resolver el primer problema en la lista del principio de esta sección. Es aún más sofisticado generalizar la integración de Romberg de esta manera:

```
function TIntegralesAbiertas.IRombergAbierta(FUNC : TUserFunc; a, b : Double;
IntegrarPor : TFuncInt ) : Double;
{ Integración de Romberg en un intervalo abirto. Regresa la integral de la función
func de a a b, usando una función de integración especificada en IntegrarPor y el
método de Romberg. Normalmente IntegrarPor será una una fórmula abierta, no se evaluan
los límites de integración. Se asume que IntegrarPor triplica el número de pasos en
cada llavada, y que sus series de error contienen  sólo potencias uniformes del
número de pasos. Las rutinas PuntoMedio, PMInfinito, PMRaizLI, PMRaizLS, PMExp, son
posibles opciones para IntegrarPor. Los parametros tienen el mismo significado que
para IRomb en las integrales cerradas.}
var
   j : Integer;
   ss,dss : Double;
   h, s : Variant;
begin
   s := varArrayCreate([0,FJMAX+1], varDouble );
   h := varArrayCreate([0,FJMAX+2], varDouble );

   h[1]:=1.0;
   for j:=1 to FJMAX do begin
       s[j] := IntegrarPor(func,a,b,j);
       if (j > FK) then begin
          MACTools.InterpolarPolinomio(h, j-FK, s, j-FK, FK, 0.0, ss, dss);
          if ( abs(dss) <= EPS*abs(ss)) then begin
```

```
            result := ss; exit;
          end;
        end;
        h[j+1]:=h[j]/9.0;
        {Dado que el número de pasos es triplicado, el error se decrementa en 1/9 del
tamaño del intervalo, no 1/4 como en los métodos de pasos dobles }
    end;
    raise EErrorIntegracion.Create('Se ha excedido el límite de iteraciones en la
Integración por el Método de Romberg');
    result := 0.0; {Nunca llega aqui}
end;
```

No hay por que preocuparse por la complicada declaración. Una típica invocación es simple (integrando la función de Bessel Y_0(x)de 0 a 2)

```
{...}
var
   respuesta : Double;
{...}
   respuesta := IRombergAbierta( bessy0, 0.0, 2.0, PuntoMedio );
```

Las diferencias entre IRomberg e IRombergAbierta son suficientes para mostrar el listado completo. Esta, sin embargo, es una excelente rutina derivada para resolver todos los problemas de integrales impropias listados al incio de este tema (excepto el quinto), como puede verse a continuación.

El truco básico para resolver integrales impropias es hacer el cambio de variables para eliminar la singularidad, o para mapear un rango infinito de integración a uno finito. Por ejemplo, la identidad

$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right) dt \qquad ab > 0 \qquad 5.5.1$$

Puede ser usada tanto para $b \to \infty$ y a positiva, o con $a \to -\infty$ y b negativa, y trabaja para cualquier función que decrezca hacia el infinito más rápido que $1/x^2$.

Se puede hacer el cambio implicado en 5.5.1 ya sea analíticamente y usar (por ejemplo) IRombergAbierta y PuntoMedio para realizar la evaluación numérica, o se puede dejar que el algoritmo numérico haga el cambio de variable por nosotros. El segundo método es preferible ya que es el más transparente para el usuario. Para implementar la ecuación (5.5.1) simplemente escribimos una versión modificada de PuntoMedio, llamada PMInfinito, que permite que b sea infinita (o, más preciso, un número muy grande para una máquina en particular, como 1×10^{30}), o a ser negativa e infinita.

```
function TIntegralesAbiertas.PMInfinito(funk : TUserFunc; aa, bb : Double; n :
integer) : Double;
{ Esta rutina es un reemplazo exacto de PuntoMedio, p.e., calcula el n-ésimo estado de
refinamiento de la integral de funk de aa a bb, excepto en que la función es evaluada
en 1/x en lugar de x.  Esto permite que el límite superior bb sea tan grande y
positivo como la máquina lo permita, o el límite aa tan grande y negativo, pero no
ambos. aa y bb deben tener el mismo signo.  }
var
```

```
  x,tnm,sum,del,ddel, a, b : Double;
  it,j : Integer;

  function FUNC( x : Double ) : Double;        {efectúa el cambio de variable}
  begin
       result := funk( 1/x ) / (x * x) ;
  end;
begin
     b := 1.0 / aa; {estas dos líneas cambian los límites de integración}
     a := 1.0 / bb;

     if (n = 1) then  {de este punto, la rutina es idéntica a PuntoMedio}
```

Si necesita integrar de un límite inferior negativo a un positivo e infinito, puede hacerlo si separa la integral en dos piezas en algún valor positivo, por ejemplo,

```
      resultado := IRombergAbierta(funk, -10, 4, PuntoMedio)
    + IRombergAbierta (funk, 4, 1e30, PMInfinito);
```

Para trabajar con una integral con una singularidad en su límite inferior, podemos hacer también un cambio de variable. Si el integrando diverge como $(x-a)^{\gamma}$, $0 \le \gamma < 1$, cerca de x = a, use la identidad

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(t^{\frac{1}{1-\gamma}} + a)dt \qquad (b > a) \qquad 5.5.2$$

Si la singularidad está en el límite superior, use la identidad

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(b - t^{\frac{1}{1-\gamma}})dt \qquad (b > a) \qquad 5.5.3$$

Si la singularidad está en ambos límites, divida la integral en un punto interior como en el ejemplo anterior.

Las ecuaciones (5.5.2) y (5.5.3) son particularmente simples en el caso de singularidades de raíces cuadradas inversas, un caso que ocurre frecuentemente en la práctica:

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2tf(a + t^2)dt \qquad (b > a) \qquad 5.5.4$$

para una singularidad en a, y

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2tf(b - t^2)dt \qquad (b > a) \qquad 5.5.6$$

para una singularidad en b. Una vez más, podemos implementar estos cambios de variable de forma transparente para el usuario definiendo rutinas que sustituyen a PuntoMedio y hacen el cambio de variable automáticamente:

```
function TIntegralesAbiertas.PMRaizLI(funk : TUserFunc; aa, bb : Double; n : integer)
: Double;
{ Esta rutina es un reemplazo exacto de PuntoMedio, excepto que permite una
singularidad de raíz cuadrada inversa en el integrando en el límite inferior aa}
var
  x,tnm,sum,del,ddel, a, b : Double;
  it,j : Integer;

  function FUNC( x : Double ) : Double;   {efectúa el cambio de variable}
  begin
       result := 2.0 * x * funk( aa + (x*x) ) ;
  end;

begin
     b := sqrt(bb-aa); {estas dos líneas cambian los límites de integración}
     a := 0;

     if (n = 1) then   {de este punto, la rutina es idéntica a PuntoMedio}
```

De forma similar,

```
function TIntegralesAbiertas.PMRaizLS(funk : TUserFunc; aa, bb : Double; n : integer)
: Double;
{ Esta rutina es un reemplazo exacto de PuntoMedio, excepto que permite una
singularidad de raíz cuadrada inversa en el integrando en el límite superior bb}
var
  x,tnm,sum,del,ddel, a, b : Double;
  it,j : Integer;

  function FUNC( x : Double ) : Double;   {efectúa el cambio de variable}
  begin
       result := 2.0 * x * funk( bb - (x*x) ) ;
  end;
begin
     b := sqrt(bb-aa); {estas dos líneas cambian los límites de integración}
     a := 0;

     if (n = 1) then   {de este punto, la rutina es idéntica a PuntoMedio}
```

Un último ejemplo será suficiente para mostrar cómo estas fórmulas son derivadas en general. Suponga que el límite de integración es infinito, y que el integrando decrece exponencialmente. Entonces necesitamos un cambio de variable que mapee $e^{-x}dx$ en $(\pm)dt$ (con el signo escogido para mantener el límite superior de la nueva variable más grande que el límite inferior). Haciendo la integral da por inspección

$$t = e^{-x} \qquad \text{o} \qquad x = -\log t \tag{5.5.6}$$

así que

$$\int_{x=a}^{x=\infty} f(x)dx = \int_{t=0}^{t=e^{-a}} f(-\log t)\frac{dt}{t} \tag{5.5.7}$$

La implementación transparente al usuario será,

```
function TIntegralesAbiertas.PMExp(funk : TUserFunc; aa, bb : Double; n : integer) :
Double;
{ Esta rutina es un reemplazo exacto de PuntoMedio, excepto que se asume que bb es
infinito (se pasa un valor pero no se usa). Se asume que la función funk decrese
exponencialmente al infinito}
var
  x,tnm,sum,del,ddel, a, b : Double;
  it,j : Integer;

  function FUNC( x : Double ) : Double;       {efectúa el cambio de variable}
  begin
       result := funk(-ln(x)) / x ;
  end;
begin
     b := exp(-aa); {estas dos líneas cambian los límites de integración}
     a := 0;

     if (n = 1) then {de este punto, la rutina es idéntica a PuntoMedio}
⋮
```

Finalmente, el componente que evalúa integrales abiertas IRomberAbiertas incluye un evento manejado por el usuario para que se puedan agregar nuevas funciones de cambio de variable y sustitución de los rangos de integración.

5.6 Cuadraturas Gaussianas y Polinomios Ortogonales

En las fórmulas de la sección anterior (5.5), la integral de una función fue aproximada por la suma de sus valores funcionales en un conjunto de puntos igualmente espaciados, multiplicados por ciertos coeficientes escogidos especialmente. Podemos apreciar que entre más libertad tengamos para escoger los coeficientes, podremos encontrar fórmulas de integración de más alto orden. La idea de la cuadratura Gaussiana es darnos la libertad de escoger no sólo los coeficientes de peso, sino también el lugar donde las abscisas serán evaluadas: ya no estarán espaciadas de igual forma. Esto es, tendremos el doble de grados de libertad a nuestra disposición; podremos ver que con las fórmulas de cuadratura Gaussiana se alcanza un mayor orden, el doble que la fórmula Newton-Cotes con el mismo número de evaluaciones de la función.

Mayor orden no es lo mismo que mayor precisión. Un mayor orden se traduce en mayor precisión sólo cuando el integrando es una curva muy suave, en el sentido de ser "bien aproximada por un polinomio".

Hay, sin embargo, una ventaja adicional de las fórmulas de cuadratura Gaussiana: podemos agrupar la selección de pesos y abscisas para hacer la integral exacta para una clase de integrandos usando polinomios sobre una función conocida W(x). La función W(x) puede escogerse para eliminar singularidades integrables de la integral deseada. Dada W(x), en otras palabras, y dado un entero N, podemos encontrar un conjunto de pesos w_j y abscisas x_j tales que la aproximación

$$\int_a^b W(x)f(x)dx \approx \sum_{j=1}^{N} w_j f(x_j) \qquad 5.6.1$$

es exacta si *f(x)* es un polinomio. Por ejemplo, para hacer la integral

$$\int_{-1}^{1} \frac{\exp(-\cos^2 x)}{\sqrt{1-x^2}}dx$$

podemos interesarnos en la fórmula de cuadratura Gaussiana basada en la siguiente elección

$$W(x) = \frac{1}{\sqrt{1-x^2}}$$

en el intervalo (-1,1), esta elección en particular se llama integración Gauss-Chebyshev, por razones que enseguida se presentan.

Note que la fórmula de integración (5.6.1) puede también escribirse con la función de peso W(x) no visible obviamente: Defina $g(x) \equiv W(x)f(x)$ y $v_j \equiv w_j / W(x_j)$. Entonces (5.6.1) se convierte en

$$\int_a^b g(x)dx \approx \sum_{j=1}^{N} v_j g(x_j) \qquad (5.6.2)$$

Cuando encuentre las tablas de pesos y abscisas para una W(x) dada, debe determinar cuidadosamente como usarlas, ya sea en la forma de (5.6.1) o (5.6.2).

Aquí hay un ejemplo de una rutina de cuadratura que contiene las abscisas y los pesos tabulados para el caso W(x) = 1 y N = 10. Dado que los pesos y las abscisas son, en este caso, simétricos alrededor del punto medio del rango de integración, sólo hay 5 valores distintos para cada uno:

```
function TIntegralesAbiertas.IGauss(FUNC : TUserFunc; a, b : Double) : Double;
{ Regresa la integral de la función func entre a y b, usando diez puntos de
integración Gauss-Legendre: la función es evaluada exactamente diez veces en los
puntos interiores en el rango de integración.}
const
 x: array[1..5] of Double = (                {Abscisas}
      0.1488743389,0.4333953941,
      0.6794095682,0.8650633666,0.9739065285
    );
 w: array[1..5] of Double = (                {Pesos}
      0.2955242247,0.2692667193,
      0.2190863625,0.1494513491,0.0666713443
    );
var
   j : Integer;
   xr,xm,dx,s : Double;
begin
  xm:=0.5*(b+a);
  xr:=0.5*(b-a);
```

```
  s:=0;

  for j:=1 to 5 do begin
      dx := xr*x[j];
      s  := s + w[j]*( func(xm+dx) + func(xm-dx) );
  end;
  s := s * xr; {Escalar la respuesta al rango de integración}
  result := s;
end;
```

La rutina anterior ilustra como se pueden usar las cuadraturas Gaussianas sin necesidad de entender la teoría detrás de ellas: sólo se necesita encontrar en un libro tablas de abscisas y pesos (p.e. [Stoer, 1980] y [Numericals, 1992]).

La teoría detrás de las cuadraturas Gaussianas se remonta a Gauss en 1814, quien uso fracciones continuas para desarrollar este concepto. En 1826 Jacobi rederivó los resultados de Gauss en el sentido de los polinomios ortogonales. El tratamiento de funciones W(x) de peso arbitrarios usando polinomios ortogonales se debe en gran medida a Christoffel en 1877.

Cálculo de Pesos y Abscisas

Un método para evaluar los pesos (cuya prueba está fuera del alcance de este trabajo) es por la fórmula

$$w_j = \frac{\langle p_{N-1} | p_{N-1} \rangle}{p_{N-1}(x_j) p'_{N-1}(x_j)} \qquad 5.6.3$$

donde $p'_N(x_j)$ es la derivada del polinomio ortogonal donde sus x_j son cero.

El cálculo de las reglas de cuadraturas Gaussianas involucran dos fases distintas: (i) la generación de los polinomios ortogonales $p_0,\ldots,p_N$; (ii) la determinación de los ceros de $p_N(x)$, y el cálculo de los pesos asociados.

A continuación se presentan las funciones de peso, intervalos y relaciones de recurrencia que generan los polinomios ortogonales más usados y sus correspondientes formulas de cuadratura Gaussiana.

Gauss-Legendre:

$$W(x)=1 \qquad -1<x<1$$
$$(j+1)P_{j+1} = (2j+1)xP_j - jP_{j-1}$$

Gauss-Chebyshev:

$$W(x) = (1-x^2)^{-1/2} \qquad -1<x<1$$

$$T_{j+1} = 2xT_j - T_{j-1}$$

Gauss-Laguerre:

$$W(x) = x^{\alpha}e^{-x} \qquad 0 < x < \infty$$
$$(j+1)L_{j+1}^{\alpha} = (-x+2j+\alpha+1)L_j^{\alpha} - (j+\alpha)L_{j-1}^{\alpha}$$

Gauss-Hermite:

$$W(x) = e^{-x^2} \qquad -\infty < x < \infty$$
$$H_{j+1} = 2xH_j - 2jH_{j-1}$$

Gauss-Jacobi:

$$W(x) = (1-x)^{\alpha}(1+x)^{\beta} \qquad -1 < x < 1$$
$$c_j P_{j+1}^{(\alpha,\beta)} = (d_j + e_j x)P_j^{(\alpha,\beta)} - f_j P_{j-1}^{(\alpha,\beta)}$$

donde los coeficientes c_j, d_j, e_j y f_j son dados por

$$c_j = 2(j+1)(j+\alpha+\beta-1)(2j+\alpha+\beta)$$
$$d_j = (2j+\alpha+\beta+1)(\alpha^2-\beta^2)$$
$$e_j = (2j+\alpha+\beta)(2j+\alpha+\beta+1)(2j+\alpha+\beta+2)$$
$$f_j = (2j+\alpha)(j+\beta)(2j+\alpha+\beta+2)$$

A continuación se presenta una rutina para calcular las abscisas y los pesos para el primer caso, y la más común, de Gauss-Legendre. Se propone como ejercicio realizar las rutinas para los cuatro casos restantes. La rutina, creada por G.B. Rybicki, usa la ecuación (5.6.3) en la forma especial del caso Gauss-Legendre,

$$w_j = \frac{2}{(1-x_j^2)\left[P'_N(x_j)\right]^2}$$

La rutina también escala el rango de integración de (x_1, x_2) a (-1,1), y provee de abscisas x_j y pesos w_j para la fórmula Gaussiana

$$\int_a^b f(x)dx = \sum_{j=1}^{N} w_j f(x_j)$$

```
procedure TForm1.gauleg(x1, x2 : Double; Var x, w : Variant; n : Integer);
{ Dados los límites de integración inferior y superior x1 y x2, y dada n, esta rutina
regresa los arreglos x[1..n] y w[1..n] de longitud n, conteniendo las abcisas y los
pesos de la fórmula de cuadratura de n puntos de Gauss-Legendre.}
var
```

```
   m,j,i : Integer;
   z1,z,xm,xl,pp,p3,p2,p1 : double ;
begin
     m:=(n+1) div 2; {Como las raíces son simétricas en el intervalo, sólo calculamos
la mitad}
     xm:=0.5*(x2+x1);
     xl:=0.5*(x2-x1);
     for i:=1 to m do begin    { Ciclo sobre las raíces deseadas }
         z:=cos(3.141592654*(i-0.25)/(n+0.5));
         {Comenzando con esta aproximación en la i-ésima raíz, entramos al ciclo
          principal de refinamiento por el método de Newton}

         repeat
            p1:=1.0;
            p2:=0.0;
            for j:=1 to n do begin { Ciclar la relación de recurrencia para obtener }
                p3:=p2;            {el polinomio de Legendre evaluado en z.}
                p2:=p1;
                p1:=((2.0*j-1.0)*z*p2-(j-1.0)*p3)/j;
            end;

            {Ahora p1 es el polinomio de Legendre deseado. Luego calculamos pp, su
derivada,}
            {usando relación estándar involucrando también a p2, el polinomio de un
orden menor.}
            pp:=n*(z*p1-p2)/(z*z-1.0);
            z1:=z;
            z:=z1-p1/pp;              { Método de  Newton. }
         until abs(z-z1) <= EPS;

         x[i]:=xm-xl*z;           {Escalar la raíz al intervalo deseado,}
         x[n+1-i]:=xm+xl*z;       {y ponerla en su contraparte simétrica.}
         w[i]:=2.0*xl/((1.0-z*z)*pp*pp);  {Calcular el peso}
         w[n+1-i]:=w[i];                  {y su contraparte simétrica.}
     end;
end;
```

5.7 Listado de los componentes de Integración Abierta

TintegralesAbiertas es el componente principal del cual se derivan los demás componentes de integración abierta o impropias.

Propiedades

NumMaxIteraciones : Número máximo de iteraciones que realizará el procedimiento de integración, antes de emitir un error.

Orden. Como todas las integrales abiertas se realizan usando el método de Romberg, es necesario especificar el orden de integración.

Eventos

OnUserFunc. Es en este evento donde el usuario escribe la función que desea integrar.

TIAPuntoMedio, TIAInfinito, TIARaizLI, TIARaizLS, TIAExponencial TIAPersonalizada y TIAGauss están heredados de TIntegralesAbiertas, así que tienen tanto sus propiedades como sus eventos.

Métodos

Integrar. Dados los límites de integración *a* y *b*, este procedimiento regresa el valor estimado de la integral de la función definida por el usuario.

Listado

```
unit IntegralesAbiertas;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

const EPS = 1.0e-6;
      TInfinito = 1.0e100;

type
  EErrorIntegracion = class(Exception);
  TUserFunc = function( x : Double ) : Double of Object;
  TUserFunc2v = function( a, b : Double ) : Double of Object;
  TCambiaFunc = function( func : TUserFunc; x : Double ) : Double of Object;

  {Función de tipo Integral}
  TFuncInt = function (FUNC : TUserFunc; a, b : Double; n : integer) : Double of
Object;


//    ------------------------------------------------------------------------//
Iniciando la definición del Componente Básico para Integrales Abiertas

  TIntegralesAbiertas = class(TComponent)
  private
  protected
    FmidS : Double;  {Permite el refinamiento sucesivo de la integración punto medio}
    FJMAX : Integer;
    FK : Integer;     {Número de puntos usados en la extrapolación del método de
Romberg, 2 = Simpson}
    FUserFunc : TUserFunc;
    function PuntoMedio(FUNC : TUserFunc; a, b : Double; n : integer) : Double;
    function PMInfinito(funk : TUserFunc; aa, bb : Double; n : integer) : Double;
    function PMRaizLI(funk : TUserFunc; aa, bb : Double; n : integer) : Double;
    function PMRaizLS(funk : TUserFunc; aa, bb : Double; n : integer) : Double;
```

```
    function PMExp(funk : TUserFunc; aa, bb : Double; n : integer) : Double;
    function IRombergAbierta(FUNC : TUserFunc; a, b : Double; IntegrarPor :
             TFuncInt ) : Double;
    function IGauss(FUNC : TUserFunc; a, b : Double) : Double;
  public
    constructor Create(AOwner : TComponent); override;
  published
    property NumMaxIteraciones : integer  read FJMAX write FJMAX default 20;
    property Orden : integer  read FK write FK default 8;
    property OnUserFunc : TUserFunc read FUserFunc write FUserFunc;
  end;

// ---------------------------------------------------------------------------
// Definiendo el componente para la Integral Abierta por el método del Punto
// Medio
  TIAPuntoMedio = class(TIntegralesAbiertas)
  public
    function Integrar( a, b : Double ) : Double;
  end;

// ---------------------------------------------------------------------------
// Definiendo el componente para la Integral Abierta por el método del Punto
// Medio para poder evaluar la integral hasta el infinito
  TIAInfinito = class(TIntegralesAbiertas)
  public
    function Integrar( a, b : Double ) : Double;
  end;

// ---------------------------------------------------------------------------
// Definiendo el componente para la Integral Abierta por el método del Punto
// Medio para poder eliminar una singularidad de raíz cuadrada en el límite
// inferior
  TIARaizLI = class(TIntegralesAbiertas)
  public
    function Integrar( a, b : Double ) : Double;
  end;

// ---------------------------------------------------------------------------
// Definiendo el componente para la Integral Abierta por el método del Punto
// Medio para poder eliminar una singularidad de raíz cuadrada en el límite
// superior
  TIARaizLS = class(TIntegralesAbiertas)
  public
    function Integrar( a, b : Double ) : Double;
  end;

// ---------------------------------------------------------------------------
// Definiendo el componente para la Integral Abierta por el método del Punto
// Medio con función de reemplazo para el caso en que la función decrese
// exponencialmente
  TIAExponencial = class(TIntegralesAbiertas)
  public
    function Integrar( a : Double ) : Double;
  end;


// ---------------------------------------------------------------------------
// Definiendo la Integración Abierta por el método de cuadratura de Gauss-
// Lagrange
  TIAGuass = class(TIntegralesAbiertas)
  public
    function Integrar( a, b : Double ) : Double;
  end;
```

```
// ------------------------------------------------------------------------
// Definiendo el componente para la Integral Abierta por el método del Punto
// Medio con función de reemplazo definida por el usuario
  TIAPersonalizada = class(TIntegralesAbiertas)
  protected
    {Estas son las funcioes que proporcione el usuario para hacer el cambio de
variable}
    FFReemplazo : TCambiaFunc; {cambia la función a integrar}
       Ffa, Ffb : TUserFunc2v;    {cambian el valor de los límites de integración}
    function PMPersonalizado(funk : TUserFunc; aa, bb : Double; n : integer) :
             Double;
  public
    function Integrar( a, b : Double ) : Double;
    property OnReemplazoFunc : TCambiaFunc read FFReemplazo write FFReemplazo;
    property OnCambiarAFunc : TUserFunc2v read Ffa write Ffa;
    property OnCambiarBFunc : TUserFunc2v read Ffb write Ffb;
  end;


procedure Register;

implementation

Uses MACTools;

constructor TIntegralesAbiertas.Create(AOwner : TComponent);
begin
   inherited Create(AOwner);
   FJMAX := 20;    FK := 8;
end;

function TIntegralesAbiertas.PuntoMedio(FUNC : TUserFunc; a, b : Double; n : integer)
: Double;
{ Esta rutina calcula el n-ésimo estado de refinamiento de una regla de punto medio
extendida. func es   la función a ser integrada entre los límites a y b. Cuando es
llamada con n=1, la rutina regresa la estimación más cruda de Int(b,a, f(x))dx.
  Subsecuentes llamadas con n=2,3,... (en ese orden secuencial) mejorará la
aproximación agregando (2/3) x 3^(n-1) puntos interiores adicionales.}
var
  x,tnm,sum,del,ddel : Double;
  it,j : Integer;
begin
     if (n = 1) then
        FmidS := (b-a)*FUNC(0.5*(a+b))
     else begin
        it:=1;
        for j:=1 to n-2 do it := it * 3;
        tnm:=it;
        del:=(b-a)/(3.0*tnm);
        ddel:=del+del; {Los puntos agregados estan alternados entre el espacio de del
y ddel}
        x:=a+0.5*del;
        sum:=0.0;
        for j:=1 to it do begin
            sum := sum + FUNC(x);
            x := x + ddel;
            sum := sum + FUNC(x);
            x := x + del;
        end;
        FmidS := (FmidS+(b-a)*sum/tnm)/3.0; {La nueva suma se combina con la integral
previa para dar una integral refinada}
     end;
```

```
    result := FmidS;
end;

function TIntegralesAbiertas.PMInfinito(funk : TUserFunc; aa, bb : Double; n :
integer) : Double;
{ Esta rutina es un reemplazo exacto de PuntoMedio, p.e., calcula el n-ésimo estado de
refinamiento de la integral de funk de aa a bb, excepto en que la función es evaluada
en 1/x en lugar de x.
  Esto permite que el límite superior bb sea tan grande y positivo como la máquina lo
permita, o el límite aa tan grande y negativo, pero no ambos. aa y bb deben tener el
mismo signo.  }
var
  x,tnm,sum,del,ddel, a, b : Double;
  it,j : Integer;

  function FUNC( x : Double ) : Double;        {efectúa el cambio de variable}
  begin
       result := funk( 1/x ) / (x * x) ;
  end;
begin
     b := 1.0 / aa;  {estas dos líneas cambian los límites de integración}
     a := 1.0 / bb;
     if (n = 1) then {de este punto, la rutina es idéntica a PuntoMedio}
        FmidS := (b-a)*FUNC(0.5*(a+b))
     else begin
        it:=1;
        for j:=1 to n-2 do it := it * 3;
        tnm:=it;
        del:=(b-a)/(3.0*tnm);
        ddel:=del+del; {Los puntos agregados estan alternados entre el espacio de del
y ddel}
        x:=a+0.5*del;
        sum:=0.0;
        for j:=1 to it do begin
            sum := sum + FUNC(x);
            x := x + ddel;
            sum := sum + FUNC(x);
            x := x + del;
        end;
        FmidS := (FmidS+(b-a)*sum/tnm)/3.0; {La nueva suma se combina con la integral
previa para dar una integral refinada}
     end;
     result := FmidS;
end;

function TIntegralesAbiertas.PMRaizLI(funk : TUserFunc; aa, bb : Double; n : integer)
: Double;
{  Esta rutina es un reemplazo exacto de PuntoMedio, excepto que permite una
singularidad de raíz cuadrada inversa en el integrando en el límite inferior aa}
var
  x,tnm,sum,del,ddel, a, b : Double;
  it,j : Integer;

  function FUNC( x : Double ) : Double;   {efectúa el cambio de variable}
  begin
       result := 2.0 * x * funk( aa + (x*x) ) ;
  end;

begin
     b := sqrt(bb-aa); {estas dos líneas cambian los límites de integración}
     a := 0;

     if (n = 1) then   {de este punto, la rutina es idéntica a PuntoMedio}
```

```
        FmidS := (b-a)*FUNC(0.5*(a+b))
     else begin
        it:=1;
        for j:=1 to n-2 do it := it * 3;
        tnm:=it;
        del:=(b-a)/(3.0*tnm);
        ddel:=del+del; {Los puntos agregados estan alternados entre el espacio de del
y ddel}
        x:=a+0.5*del;
        sum:=0.0;
        for j:=1 to it do begin
            sum := sum + FUNC(x);
            x := x + ddel;
            sum := sum + FUNC(x);
            x := x + del;
        end;
        FmidS := (FmidS+(b-a)*sum/tnm)/3.0; {La nueva suma se combina con la integral
previa para dar una integral refinada}
     end;
     result := FmidS;
end;

function TIntegralesAbiertas.PMRaizLS(funk : TUserFunc; aa, bb : Double; n : integer)
: Double;
{ Esta rutina es un reemplazo exacto de PuntoMedio, excepto que permite una
singularidad de raíz cuadrada inversa en el integrando en el límite superior bb}
var
  x,tnm,sum,del,ddel, a, b : Double;
  it,j : Integer;

  function FUNC( x : Double ) : Double;          {efectúa el cambio de variable}
  begin
       result := 2.0 * x * funk( bb - (x*x) ) ;
  end;
begin
     b := sqrt(bb-aa); {estas dos líneas cambian los límites de integración}
     a := 0;

     if (n = 1) then   {de este punto, la rutina es idéntica a PuntoMedio}
        FmidS := (b-a)*FUNC(0.5*(a+b))
     else begin
        it:=1;
        for j:=1 to n-2 do it := it * 3;
        tnm:=it;
        del:=(b-a)/(3.0*tnm);
        ddel:=del+del; {Los puntos agregados estan alternados entre el espacio de del
y ddel}
        x:=a+0.5*del;
        sum:=0.0;
        for j:=1 to it do begin
            sum := sum + FUNC(x);
            x := x + ddel;
            sum := sum + FUNC(x);
            x := x + del;
        end;
        FmidS := (FmidS+(b-a)*sum/tnm)/3.0; {La nueva suma se combina con la integral
previa para dar una integral refinada}
     end;
     result := FmidS;
end;

function TIntegralesAbiertas.PMExp(funk : TUserFunc; aa, bb : Double; n : integer) :
Double;
```

```
{ Esta rutina es un reemplazo exacto de PuntoMedio, excepto que se asume que bb es
infinito
  (se pasa un valor pero no se usa). Se asume que la función funk decrese
exponencialmente al infinito}
var
  x,tnm,sum,del,ddel, a, b : Double;
  it,j : Integer;

  function FUNC( x : Double ) : Double;      {efectúa el cambio de variable}
  begin
       result := funk(-ln(x)) / x ;
  end;
begin
     b := exp(-aa); {estas dos líneas cambian los límites de integración}
     a := 0;

     if (n = 1) then {de este punto, la rutina es idéntica a PuntoMedio}
        FmidS := (b-a)*FUNC(0.5*(a+b))
     else begin
        it:=1;
        for j:=1 to n-2 do it := it * 3;
        tnm:=it;
        del:=(b-a)/(3.0*tnm);
        ddel:=del+del; {Los puntos agregados estan alternados entre el espacio de del
y ddel}
        x:=a+0.5*del;
        sum:=0.0;
        for j:=1 to it do begin
            sum := sum + FUNC(x);
            x := x + ddel;
            sum := sum + FUNC(x);
            x := x + del;
        end;
        FmidS := (FmidS+(b-a)*sum/tnm)/3.0; {La nueva suma se combina con la integral
previa para dar una integral refinada}
     end;
     result := FmidS;
end;

function TIntegralesAbiertas.IRombergAbierta(FUNC : TUserFunc; a, b : Double;
IntegrarPor : TFuncInt ) : Double;
{ Integración de Romberg en un intervalo abirto. Regresa la integral de la función
func de a a b, usando una función de integración especificada en IntegrarPor y el
método de Romberg. Normalmente IntegrarPor será una una fórmula abierta, no se evaluan
los límites de integración. Se asume que IntegrarPor triplica el número de pasos en
cada llavada, y que sus series de error contienen  sólo potencias uniformes del
número de pasos. Las rutinas PuntoMedio, PMInfinito, PMRaizLI, PMRaizLS, PMExp son
posibles opciones para IntegrarPor. Los parametros tienen el mismo significado que
para IRomb en las integrales cerradas.}
var
   j : Integer;
   ss,dss : Double;
   h, s : Variant;
begin
   s := varArrayCreate([0,FJMAX+1], varDouble );
   h := varArrayCreate([0,FJMAX+2], varDouble );

   h[1]:=1.0;
   for j:=1 to FJMAX do begin
       s[j] := IntegrarPor(func,a,b,j);
       if (j > FK) then begin
          MACTools.InterpolarPolinomio(h, j-FK, s, j-FK, FK, 0.0, ss, dss);
          if ( abs(dss) <= EPS*abs(ss)) then begin
```

```
            result := ss; exit;
          end;
        end;
        h[j+1]:=h[j]/9.0;
        {Dado que el número de pasos es triplicado, el error se decrementa en 1/9 del
tamaño del intervalo, no 1/4 como en los métodos de pasos dobles }
    end;
    raise EErrorIntegracion.Create('Se ha excedido el límite de iteraciones en la
Integración por el Método de Romberg');
    result := 0.0; {Nunca llega aqui}
end;


function TIntegralesAbiertas.IGauss(FUNC : TUserFunc; a, b : Double) : Double;
{ Regresa la integral de la función func entre a y b, usando diez puntos de
integración Gauss-Legendre: la función es evaluada exactamente diez vecesen los puntos
interiores en el rango de integración.}
const
 x: array[1..5] of Double = (                {Abscisas}
      0.1488743389,0.4333953941,
      0.6794095682,0.8650633666,0.9739065285
    );
 w: array[1..5] of Double = (                {Pesos}
      0.2955242247,0.2692667193,
      0.2190863625,0.1494513491,0.0666713443
    );
var
    j : Integer;
    xr,xm,dx,s : Double;
begin
   xm:=0.5*(b+a);
   xr:=0.5*(b-a);
   s:=0;

   for j:=1 to 5 do begin
       dx := xr*x[j];
       s  := s + w[j]*( func(xm+dx) + func(xm-dx) );
   end;
   s := s * xr; {Escalar la respuesta al rango de integración}
   result := s;
end;

// ---------------------------------------------------------------------------
// Implementado la Integración Abierta por el Punto Medio
function TIAPuntoMedio.Integrar( a,b : Double ) : Double;
begin
     if assigned(FUserFunc) then
        result := IRombergAbierta(FUserFunc, a, b, PuntoMedio)
     else result := 0;
end;

// ---------------------------------------------------------------------------
// Implementado la Integración Abierta por el Punto Medio con un extremo en el
// infinito
function TIAInfinito.Integrar( a, b : Double ) : Double;
begin
     if assigned(FUserFunc) then begin
        result := IRombergAbierta(FUserFunc, a, b, PMInfinito)
     end
     else result := 0;
end;
```

```
//    ------------------------------------------------------------------------//
Implementado la Integración Abierta por el Punto Medio con un singularidad de
// raíz cuadrada en el límite inferior
function TIARaizLI.Integrar( a, b : Double ) : Double;
begin
     if assigned(FUserFunc) then begin
        result := IRombergAbierta(FUserFunc, a, b, PMRaizLI)
     end
     else result := 0;
end;

//    ------------------------------------------------------------------------//
Implementado la Integración Abierta por el Punto Medio con un singularidad de
// raíz cuadrada en el límite superior
function TIARaizLS.Integrar( a, b : Double ) : Double;
begin
     if assigned(FUserFunc) then begin
        result := IRombergAbierta(FUserFunc, a, b, PMRaizLS)
     end
     else result := 0;
end;


//    ------------------------------------------------------------------------//
Implementado la Integración Abierta por el Punto Medio con un singularidad con
// reemplazo de función que asume un decrecimiento exponencial del integrando.
function TIAExponencial.Integrar( a : Double ) : Double;
begin
     if assigned(FUserFunc) then begin
        result := IRombergAbierta(FUserFunc, a, TInfinito, PMExp)
     end
     else result := 0;
end;

//    ------------------------------------------------------------------------//
Implementado la Integración Abierta por el método de cuadratura de Gauss-Lagrange
function TIAGuass.Integrar( a, b : Double ) : Double;
begin
     if assigned(FUserFunc) then begin
        result := IGauss(FUserFunc, a, b)
     end
     else result := 0;
end;

//    ------------------------------------------------------------------------//
Implementado la Integración Abierta por el Punto Medio con un singularidad con
// reemplazo definido por el usuario

//*****************************************************
// Punto Medio Definido por el usuario
//*****************************************************

function TIAPersonalizada.PMPersonalizado(funk : TUserFunc; aa, bb : Double; n :
integer) : Double;
{ Esta rutina es un reemplazo exacto de PuntoMedio, excepto que permite al usuario
definir sus propias sustituciones para eliminar las singularidades.}
var
  x,tnm,sum,del,ddel, a, b : Double;
  it,j : Integer;
begin
     b := Ffb(a,b); {estas dos líneas cambian los límites de integración}
     a := Ffa(a,b);
```

```
      if (n = 1) then
         FmidS := (b-a) * FFReemplazo( funk, 0.5*(a+b) )
      else begin
         it:=1;
         for j:=1 to n-2 do it := it * 3;
         tnm:=it;
         del:=(b-a)/(3.0*tnm);
         ddel:=del+del; {Los puntos agregados estan alternados entre el espacio de del
y ddel}
         x:=a+0.5*del;
         sum:=0.0;
         for j:=1 to it do begin
             sum := sum + FFReemplazo( funk, x );
             x := x + ddel;
             sum := sum + FFReemplazo( funk, x );
             x := x + del;
         end;
         FmidS := (FmidS+(b-a)*sum/tnm)/3.0; {La nueva suma se combina con la integral
previa para dar una integral refinada}
      end;
      result := FmidS;
end;

//*****************************************************
function TIAPersonalizada.Integrar( a, b : Double ) : Double;
begin
      if assigned(FUserFunc) and
         assigned(FFReemplazo) and
         assigned(Ffa) and
         assigned(Ffb)
      then begin
         result := IRombergAbierta(FUserFunc, a, b, PMPersonalizado)
      end
      else result := 0;
end;


procedure Register;
begin
  RegisterComponents('MAC',   [TIAPuntoMedio,   TIAInfinito,   TIARaizLI,   TIARaizLS,
TIAExponencial, TIAPersonalizada, TIAGuass]);
end;

end.
```

Capítulo

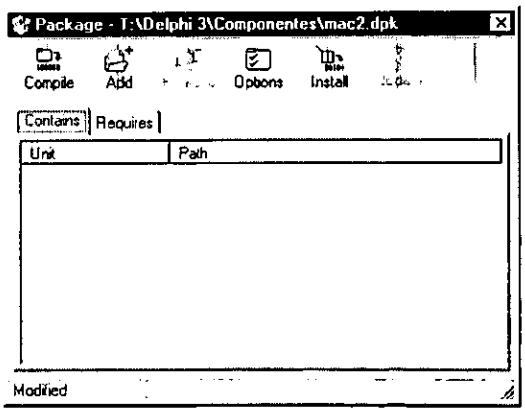
6

Usando los componentes

6.1. Instalando los componentes

Los componentes en Delphi se compilan en paquetes, para crear uno nuevo que contenga los componentes de este trabajo, se elige File | New del menú, el cual mostrará el cuadro de dialogo para nuevos elementos, ahora se elige el icono "package" o paquete. Este pregunta por un nombre de archivo. Para estos ejemplos, se uso el nombre *MAC*. Ahora aparece una página en blanco del Administrador de paquetes, como se muestra en la figura 6.1.1

Fig. 6.1.1
El administrador de paquetes listo para agregar un nuevo componente



Para agregar unidades existentes, ver fig. 6.1.2.:

Presionar sobre el icono Add en el cuadro de dialogo del administrador de paquetes.
Seleccionar la pestaña Add Unit
Elegir el nombre de la unidad que se desea agregar
Presionar OK para guardar los cambios

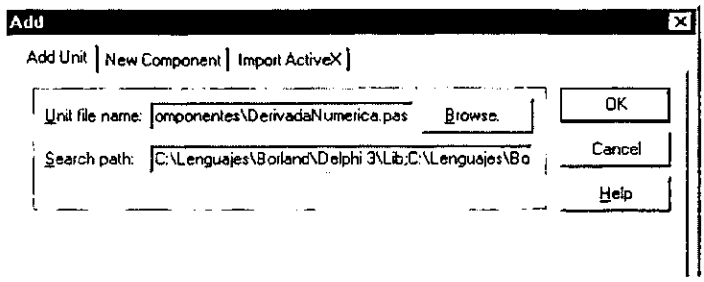


Fig. 6.1.2.
Agregando componentes

Para compilar e instalar los componentes:

En el Administrador de Paquetes, presionar el icono ***Compile***. Esto compila las unidades dentro del archivo del paquete.

En el Administrador de Paquetes, presionar el icono ***Install***. Esto instala el paquete. Si el paquete se instala satisfactoriamente, aparece un cuadro de dialogo informando que la operación fue un éxito, como se muestra en la figura 6.1.3.

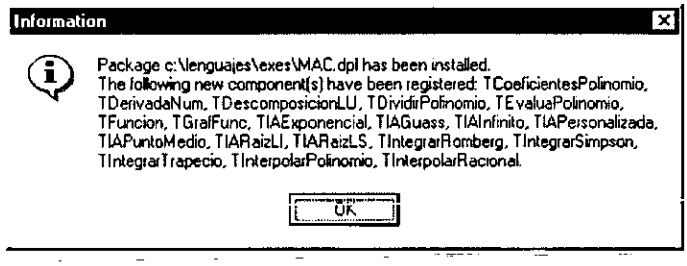


Fig. 6.1.3.
El paquete instalado satisfactoriamente

La barra de herramientas (fig.6.1.4) ha cambiado, ahora aparece una nueva pestaña llamada MAC y en ella aparecen los nuevos componentes.

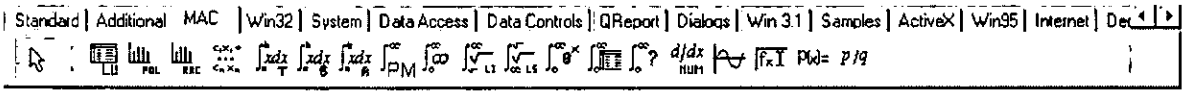


Fig. 6.1.4.
Barra de herramientas con los componentes escritos en este trabajo

Componente	Descripción
	El componente Descomposición LU permite encontrar la solución a sistemas de ecuaciones lineales.
	Interpolación por polinomios.
	Interpolación de funciones usando polinomios racionales.
	Coeficientes del polinomio de interpolación.

Componente	Descripción
	Integración cerrada por el método del Trapecio.
	Integración cerrada por el método de Romberg.
	Integración cerrada por el método de Simpson.
	Integración abierta por el método del Punto Medio.
	Integración abierta de funciones con límites infinitos.
	Integración abierta con sustitución de la singularidad por medio de raíces en el límite superior.
	Integración abierta con sustitución de la singularidad por medio de raíces en el límite inferior.
	Integración abierta por el método de la cuadratura Gaussiana.
	Integración abierta con sustitución de la singularidad del límite usando una función definida por el usuario.
	Derivada numérica de una función.
	Graficar una función.
	El componente TFuncion permite introducir una cadena e interpretarla como una función
	Evaluación de un polinomio y *n* de sus derivadas en un punto dado.
	Este componente permite realizar la división sintética de dos polinomios.

Una vez terminada la instalación del paquete, y la aparición de los nuevos componentes en la barra, ya se puede iniciar el desarrollo de nuevas aplicaciones usando estos componentes.

6.2 Ejemplos de aplicaciones usando los componentes

Para utilizar los componentes dentro de un nuevo proyecto basta con arrastrarlos hacia una forma, cambiar algunas propiedades si es necesario y escribir el código respectivo para usar alguno de los métodos de cada uno.

Los componentes se pueden mezclar para crear diversos tipos de aplicaciones, por ejemplo, podemos utilizar los que se refieren a la integración numérica tradicional para quizás comparar sus resultados con algún método que estemos desarrollando, o quizá para revisar la estimación del error de cada método para situaciones especiales, etcétera; la aplicación de cada uno dependerá de la creatividad y problemas que se presenten y se deseen resolver.

La Inversa de una Matriz

Usando las rutinas de descomposición LU y de sustitución hacia atrás, es realmente sencillo encontrar la inversa de una matriz columna a columna.

El algoritmo consiste en usar la matriz identidad como vector solución, se opera columna a columna y esto nos entrega, a su vez, la matriz inversa, ver fig. 6.2.1.

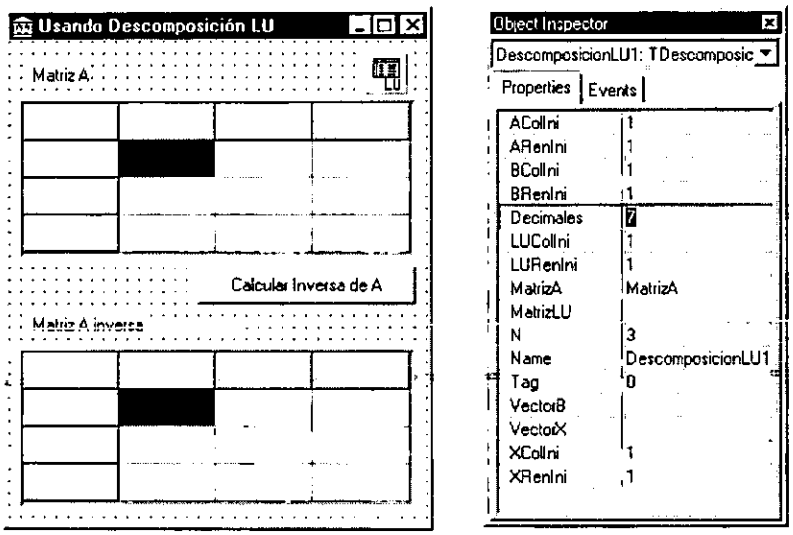


Fig. 6.2.1
Forma y propiedades para la Inversión de una matriz

```
unit InversaMatriz;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, DescomposicionLU, Grids;

type
  TInverisaDeA = class(TForm)
    DescomposicionLU1: TDescomposicionLU;
    Button1: TButton;
    MatrizA: TStringGrid;
    AInversa: TStringGrid;
    Label2: TLabel;
    Label3: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;

var
  InverisaDeA: TInverisaDeA;

implementation

{$R *.DFM}

uses MACTools;

procedure TInverisaDeA.Button1Click(Sender: TObject);
var y, col, xcol : Variant;
    i,j : integer;
begin

    with DescomposicionLU1 do begin
      Descomponer;

      y := varArrayCreate([1,N, 1,N], varDouble );
      col := varArrayCreate([1,N], varDouble );
      xcol    := varArrayCreate([1,N], varDouble );

      {Encontrar la inversa por columnas}
      for j:=1 to N do begin
          for i:=1 to N do col[i] := 0.0;
          col[j]:=1.0;
          xcol := SustitucionAtras_LU( LU, N, IDX, col );
          MejorarSolucion( A, LU, N, IDX, col, xcol );
          MejorarSolucion( A, LU, N, IDX, col, xcol );
          for i:=1 to N do y[i,j] := xcol[i];
      end;

      MACTools.MatrizASG( y, AInversa, N, N, 1, 1, Formato );
    end;
end;

end.
```

La primera aproximación a la solución de la matriz inversa se obtiene de las llamadas a la función SustitucionAtras_LU, sin embargo se puede mejorar ésta ejecutando a la función MejorarSolución una o dos veces. (Ver fig. 6.2.2)

La matriz y contiene al final la inversa de la matriz original **A**.

Incidentalmente, si tiene la necesidad de calcular $A^{-1} \cdot B$ a partir de las matrices A y B, primero debe descomponer en LU la matriz A y después hacer sustitución hacia atrás con las columnas de B en lugar de los vectores unitarios que produjeron la inversa de A. Esto ahorra una multiplicación de matrices, y es mucho más precisa.

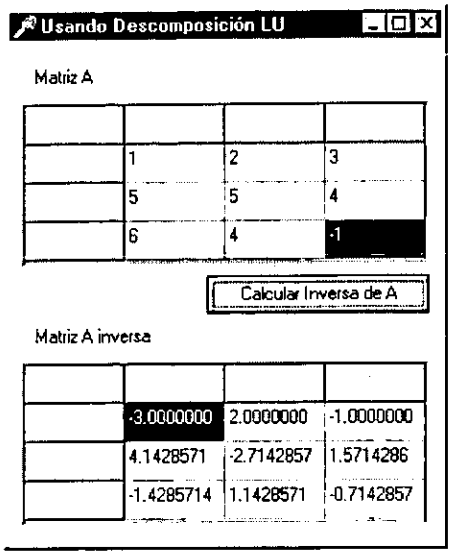


Fig. 6.2.2
Cálculo de la Inversa de una matriz usando el componente de descomposición LU

Determinante de una Matriz

El determinante de una matriz descompuesta LU es simplemente el producto de los elementos de la diagonal,

$$\det = \prod_{j=1}^{N} \beta_{jj}$$

No calculamos la descomposición de la matriz original, sino una descomposición de una permutación por renglones de ella. Afortunadamente, guardamos si el número de renglones intercambiados fue par o impar, así que podemos agregar al producto el signo correspondiente.

El cálculo del determinante sólo necesita una llamada a Descomposición LU, sin subsecuentes llamadas de sustituciones hacia atrás.

```
...
with DescomposicionLU1 do begin
      Descomponer;

      Determinante := D;
      for j:=1 to N do determinante := determinante * a[j,j];
end;
...
```

La variable ***determinante*** ahora contiene el determinante de la matriz original A.

Integrales cerradas

Podemos realizar un comparativo entre las precisiones de los tres métodos de integración cerrada que desarrollamos, usamos un componente TFuncion para que el ejercicio se pueda

realizar con cualquier función y los componentes TIntegrarTrapecio, TIntegrarSimpson y TIntegrarRomberg. (Ver fig. 6.2.3)

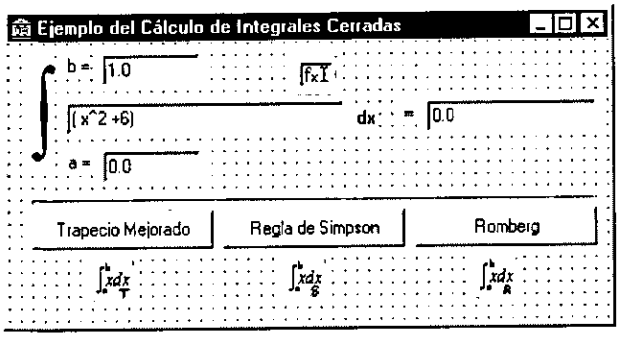


Fig. 6.2.3
Forma con los componentes de integración cerrada y lectura de funciones

Lo único que debemos hacer es llamar a los métodos respectivos de integración y observar los resultados, ver fig. 6.2.4.

```
// Programa que ejemplifica el uso de los componentes de integrales cerradas, y
// leer funciones mostrando la precisión que pueden alcanzar ante diferentes
// funciones que ingrese el usuario

unit EjemploIntegralesCerradas;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, IntegralesCerradas, ExtCtrls, LeerFunciones;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    edA: TEdit;
    edB: TEdit;
    bt_Trapecio: TButton;
    bt_Simpson: TButton;
    edResultado: TEdit;
    Label5: TLabel;
    Label6: TLabel;
    IntegrarTrapecio1: TIntegrarTrapecio;
    bt_Romberg: TButton;
    IntegrarSimpson1: TIntegrarSimpson;
    IntegrarRomberg1: TIntegrarRomberg;
    Funcion1: TFuncion;
    edFuncion: TEdit;
    Bevel1: TBevel;
    procedure bt_TrapecioClick(Sender: TObject);
    function  IntegrarTrapecio1UserFunc(x: Double): Double;
    procedure bt_SimpsonClick(Sender: TObject);
    procedure bt_RombergClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure edFuncionExit(Sender: TObject);
  private
```

```
    { Private declarations }
    procedure LeerAyB(Var a, b : Double);
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

// Dado que la función trae un valor por default, debe ser validado al iniciar
procedure TForm1.FormCreate(Sender: TObject);
begin
     Funcion1.ValidarFuncion;
end;

// Cada vez que se cambie de función debe validarse
procedure TForm1.edFuncionExit(Sender: TObject);
begin
     Funcion1.ValidarFuncion;
end;


procedure TForm1.LeerAyB(Var a, b : Double);
begin
     try
        a := StrToFloat( edA.Text );
     except
        Application.MessageBox('Escriba un número valido en A',
     'Error de captura', IDOK);
        exit;
     end;
     try
        b := StrToFloat( edB.Text );
     except
        Application.MessageBox('Escriba un número valido en B',
     'Error de captura', IDOK);
        exit;
     end;
end;


// Los tres componentes de integración evalúan la misma función
function TForm1.IntegrarTrapecio1UserFunc(x: Double): Double;
begin
   result := Funcion1.EvaluaFuncion(x);  // Funcion1 es lo que el usuario escribe
end;



procedure TForm1.bt_TrapecioClick(Sender: TObject);
var a, b : Double;
begin
     LeerAyB(a, b);
     edResultado.Text := Format('%.6f', [IntegrarTrapecio1.Integrar( a, b )] );
end;

procedure TForm1.bt_SimpsonClick(Sender: TObject);
var a, b : Double;
begin
     LeerAyB(a, b);
```

```
     edResultado.Text := Format('%.6f', [IntegrarSimpson1.Integrar( a, b )] );
end;

procedure TForm1.bt_RombergClick(Sender: TObject);
var a, b : Double;
begin
     LeerAyB(a, b);
     edResultado.Text := Format('%.6f', [IntegrarRomberg1.Integrar( a, b )] );
end;


end.
```

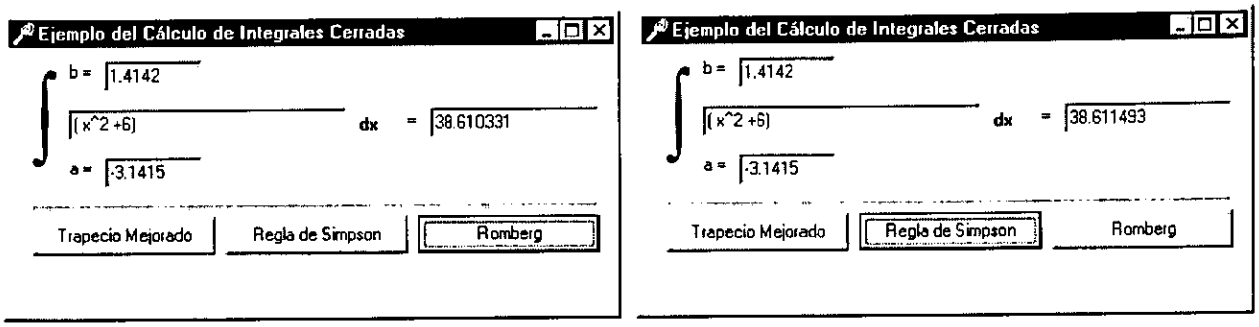


Fig. 6.2.4
Cálculo de una integral, mostrando la diferencia de precisión entre el método de Romberg y el de Simpson

Interpolación de funciones

Usando los componentes de interpolación, podemos convertir una tabla de valores en una función, la cual podemos evaluar en cualquier punto incluso graficarla. El siguiente ejemplo muestra las principales características de estos componentes. (Fig. 6.2.5).

Con este ejercicio se puede observar la diferencia de precisión que se alcanza con una interpolación polinomial y con una racional. Como complemento, se agrega el componente CoeficientesPolinomio para observar el polinomio que esta aproximando la tabla de valores, este polinomio deberá ser muy parecido al que estamos graficando, ver la Fig. 6.2.6.

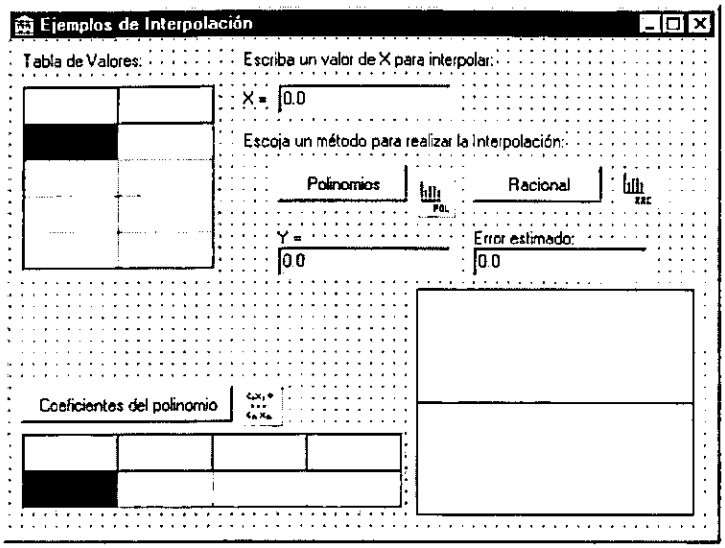


Fig. 6.2.5
Forma con los componentes de interpolación y graficar funciones

```
unit EjemploInterpolar;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Grids, InterpolarPolinomio, StdCtrls, Buttons, GraficaFuncion;

type
  TForm1 = class(TForm)
    InterpolarPolinomio1: TInterpolarPolinomio;
    TablaXY: TStringGrid;
    edX: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    btPoloniomio: TBitBtn;
    Label3: TLabel;
    Label4: TLabel;
    edY: TEdit;
    Label5: TLabel;
    btRacional: TBitBtn;
    edError: TEdit;
    Label6: TLabel;
    InterpolarRacional1: TInterpolarRacional;
    CoefPolinomio: TStringGrid;
    BitBtn1: TBitBtn;
    CoeficientesPolinomio1: TCoeficientesPolinomio;
    GrafFunc1: TGrafFunc;
    procedure FormCreate(Sender: TObject);
    procedure btPoloniomioClick(Sender: TObject);
    procedure btRacionalClick(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
    procedure GrafFunc1UserFunc(x: Double; var y: Double);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
```

```
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
var i : integer;
begin
     TablaXY.Cells[0,0] := '     X'; TablaXY.Cells[1,0] := '     Y';
     for i:= 1 to 4 do begin
          TablaXY.Cells[0,i] := Format('%10.2f', [i*1.0] );
          TablaXY.Cells[1,i] := Format('%10.2f', [0.0] );

          CoefPolinomio.Cells[i-1, 0] := Format('     x^%d', [i-1] );
          CoefPolinomio.Cells[i-1, 1] := Format('     %d', [0] );
     end;
end;


procedure TForm1.btPoloniomioClick(Sender: TObject);
var x, y, dy : Double;
begin
     try
        x := StrToFloat( edX.Text );
     except
        Application.MessageBox('Escriba un número valido en X', 'Error de captura',
IDOK);
        exit;
     end;
     InterpolarPolinomio1.Interpolar( x, y, dy );
     edY.Text := Format('%.6f', [y]);
     edError.Text := Format('%.6f', [dy]);
     GrafFunc1.Repaint;
end;


procedure TForm1.btRacionalClick(Sender: TObject);
var x, y, dy : Double;
begin
     try
        x := StrToFloat( edX.Text );
     except
        Application.MessageBox('Escriba un número valido en X', 'Error de captura',
IDOK);
        exit;
     end;
     InterpolarRacional1.Interpolar( x, y, dy );
     edY.Text := Format('%.6f', [y]);
     edError.Text := Format('%.6f', [dy]);
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
begin
     CoeficientesPolinomio1.CalcularCoeficientes;
end;

procedure TForm1.GrafFunc1UserFunc(x: Double; var y: Double);
var dy : Double;
begin
     InterpolarRacional1.Interpolar( x, y, dy );
end;
```

```
end.
```

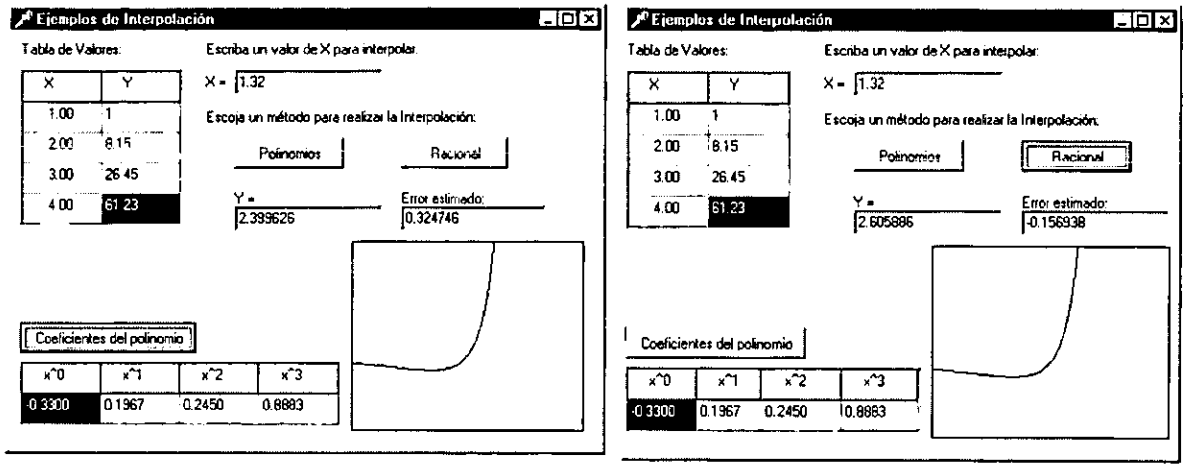


Fig. 6.2.6
Interpolación de una función basada en una tabla de datos, aquí se muestra la diferencia de precisión entre el método de polinomios y por fracciones racionales. Se muestra además la gráfica de la función interpolada por polinomios.

Operador de Matrices

Utilizando el componente de Descomposición LU y agregando algunos algoritmos para hacer operaciones entre matrices, se puede desarrollar una aplicación un poco más compleja, el siguiente es el listado completo, vea las figuras 6.2.7 y 6.2.8 para ver como se ve la forma en la etapa de desarrollo y el comportamiento de la aplicación en una corrida.

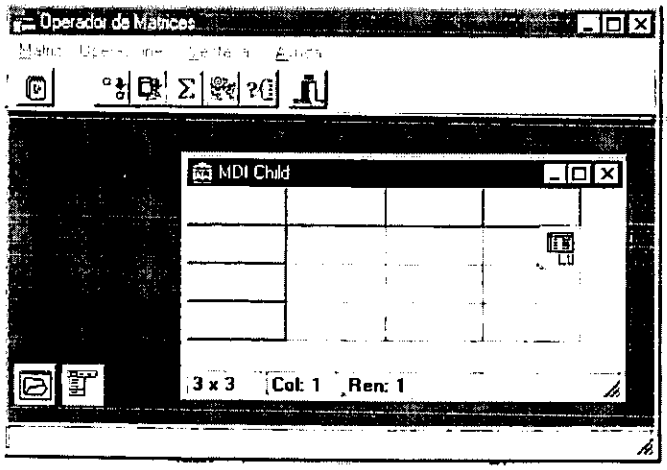


Fig. 6.2.7
Forma MDI principal y forma Child que define a la matriz y sus operaciones, incluye un componente LU

```
// Programa que muestra el uso del componente Descomposición LU en una
// calculadora básica de matrices
```

```
unit Main;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ComCtrls;

type
  TMainForm = class(TForm)
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    FileNewItem: TMenuItem;
    Window1: TMenuItem;
    Help1: TMenuItem;
    N1: TMenuItem;
    FileExitItem: TMenuItem;
    WindowCascadeItem: TMenuItem;
    WindowTileItem: TMenuItem;
    WindowArrangeItem: TMenuItem;
    HelpAboutItem: TMenuItem;
    OpenDialog: TOpenDialog;
    WindowMinimizeItem: TMenuItem;
    SpeedPanel: TPanel;
    InversaBtn: TSpeedButton;
    TranspBtn: TSpeedButton;
    SumarBtn: TSpeedButton;
    MultBtn: TSpeedButton;
    ExitBtn: TSpeedButton;
    StatusBar: TStatusBar;
    NewBtn: TSpeedButton;
    Operaciones1: TMenuItem;
    Inversa: TMenuItem;
    Sumar1: TMenuItem;
    Multiplicar1: TMenuItem;
    N2: TMenuItem;
    Resolversistemadeecuaciones1: TMenuItem;
    N3: TMenuItem;
    Transpuesta1: TMenuItem;
    LUBtn: TSpeedButton;
    procedure FormCreate(Sender: TObject);
    procedure FileNewItemClick(Sender: TObject);
    procedure WindowCascadeItemClick(Sender: TObject);
    procedure UpdateMenuItems(Sender: TObject);
    procedure WindowTileItemClick(Sender: TObject);
    procedure WindowArrangeItemClick(Sender: TObject);
    procedure FileExitItemClick(Sender: TObject);
    procedure WindowMinimizeItemClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure HelpAboutItemClick(Sender: TObject);
    procedure InversaClick(Sender: TObject);
    procedure Sumar1Click(Sender: TObject);
    procedure Multiplicar1Click(Sender: TObject);
    procedure Transpuesta1Click(Sender: TObject);
    procedure Resolversistemadeecuaciones1Click(Sender: TObject);
  private
    { Private declarations }
    procedure CreateMDIChild(const Name: string; col, ren : Integer);
    procedure ShowHint(Sender: TObject);
  public
    { Public declarations }
  end;

var
```

```
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses ChildWin, About, Selector, OperaMatrices, Nueva;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnHint := ShowHint;
  Screen.OnActiveFormChange := UpdateMenuItems;
end;

procedure TMainForm.ShowHint(Sender: TObject);
begin
  StatusBar.SimpleText := Application.Hint;
end;

procedure TMainForm.CreateMDIChild(const Name: string; col, ren : Integer);
var
  Child: TMDIChild;
begin
  { create a new MDI child window }
  Child := TMDIChild.Create(Application);
  Child.Caption := Name;
  Child.DimensionMatriz(col, ren);
end;

procedure TMainForm.FileNewItemClick(Sender: TObject);
begin
     with NewMatForm do
        If ShowModal = mrOK then begin
        CreateMDIChild('Matriz ' + IntToStr(MainForm.MDIChildCount + 1), Columnas,
Renglones);
        end;
end;


procedure TMainForm.FileExitItemClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.WindowCascadeItemClick(Sender: TObject);
begin
  Cascade;
end;

procedure TMainForm.WindowTileItemClick(Sender: TObject);
begin
  Tile;
end;

procedure TMainForm.WindowArrangeItemClick(Sender: TObject);
begin
  ArrangeIcons;
end;

procedure TMainForm.WindowMinimizeItemClick(Sender: TObject);
var
  I: Integer;
```

```
begin
  { Must be done backwards through the MDIChildren array }
  for I := MDIChildCount - 1 downto 0 do
    MDIChildren[I].WindowState := wsMinimized;
end;

procedure TMainForm.UpdateMenuItems(Sender: TObject);
begin
  InversaBtn.Enabled := MDIChildCount > 0;
  TranspBtn.Enabled := MDIChildCount > 0;
  SumarBtn.Enabled := MDIChildCount > 0;
  MultBtn.Enabled := MDIChildCount > 0;
  LUBtn.Enabled := MDIChildCount > 0;

  Operaciones1.Enabled := MDIChildCount > 0;

  WindowCascadeItem.Enabled := MDIChildCount > 0;
  WindowTileItem.Enabled := MDIChildCount > 0;
  WindowArrangeItem.Enabled := MDIChildCount > 0;
  WindowMinimizeItem.Enabled := MDIChildCount > 0;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  Screen.OnActiveFormChange := nil;
end;

procedure TMainForm.HelpAboutItemClick(Sender: TObject);
begin
  AboutBox.ShowModal;
end;


procedure TMainForm.InversaClick(Sender: TObject);
var A, Inversa: TMDIChild;
    y, col, xcol : Variant;
    i, j : integer;
begin
     A := TMDIChild(ActiveMDIChild);
     If Not A.esCuadrada then begin
        ShowMessage('Sólo se puede calcular la inversa a matrices cuadradas');
        exit;
     end;

     inversa := TMDIChild.Create(Application);
     inversa.Caption := 'Matriz ' + IntToStr(MainForm.MDIChildCount + 1) + ': Inversa
de ' + A.caption ;
     inversa.DimensionMatriz(A.columnas, A.renglones);


      with A.DescomposicionLU1 do begin
        Descomponer;

        y := varArrayCreate([1, N, 1, N], varDouble );
        col := varArrayCreate([1,N], varDouble );
        xcol   := varArrayCreate([1,N], varDouble );

        {Encontrar la inversa por columnas}
        for j:=1 to N do begin
            for i:=1 to N do col[i] := 0.0;
            col[j]:=1.0;
            xcol := SustitucionAtras_LU( LU, N, IDX, col );
            MejorarSolucion( A, LU, N, IDX, col, xcol );
```

```
// Con una tiene certeza de 5 decimales
            MejorarSolucion( A, LU, N, IDX, col, xcol );
// Si quiere aumentar la precisión, con 2 es suficiente
            for i:=1 to N do y[i,j] := xcol[i];
        end;

      end;

      matCopia( y, Inversa.laMatriz );
      Inversa.MostrarDatos;

end;

procedure TMainForm.Sumar1Click(Sender: TObject);
Var  ChildSuma: TMDIChild;
     a, b, s : Variant;
begin
     SelectorForm.InicializarCombos;

     with SelectorForm do
        If ShowModal = mrOK then begin
           a :=TMDIChild(Matriz_A.Items.Objects[ Matriz_A.ItemIndex ]).laMatriz ;
           b :=TMDIChild(Matriz_B.Items.Objects[ Matriz_B.ItemIndex ]).laMatriz ;

           s := matSuma(a, b) ;
           ChildSuma := TMDIChild.Create(Application);
           ChildSuma.Caption := 'Matriz ' + IntToStr(MainForm.MDIChildCount + 1) + ':
Suma de ' + Matriz_A.Items.Strings[Matriz_A.ItemIndex] + ' y ' +
Matriz_B.Items.Strings[Matriz_B.ItemIndex] ;
           ChildSuma.DimensionMatriz( matNum_Col(a), matNum_Ren(a) );
           matCopia( s, ChildSuma.laMatriz );
           ChildSuma.MostrarDatos;
        end;
end;

procedure TMainForm.Multiplicar1Click(Sender: TObject);
Var  ChildMult: TMDIChild;
     a, b, z : Variant;
begin
     SelectorForm.InicializarCombos;

     with SelectorForm do
        If ShowModal = mrOK then begin
           a :=TMDIChild(Matriz_A.Items.Objects[ Matriz_A.ItemIndex ]).laMatriz ;
           b :=TMDIChild(Matriz_B.Items.Objects[ Matriz_B.ItemIndex ]).laMatriz ;

           z := matMultiplica(a, b) ;
           ChildMult := TMDIChild.Create(Application);
           ChildMult.Caption := 'Matriz ' + IntToStr(MainForm.MDIChildCount + 1) + ':
Multiplicación de ' + Matriz_A.Items.Strings[Matriz_A.ItemIndex] + ' y ' +
Matriz_B.Items.Strings[Matriz_B.ItemIndex] ;
           ChildMult.DimensionMatriz( matNum_Col(b), matNum_Ren(a) );
           matCopia( z, ChildMult.laMatriz );
           ChildMult.MostrarDatos;
        end;
end;

procedure TMainForm.Transpuesta1Click(Sender: TObject);
var A, Transpuesta: TMDIChild;
    t : Variant;
begin
     A := TMDIChild(ActiveMDIChild);
     Transpuesta := TMDIChild.Create(Application);
```

```
     Transpuesta.Caption := 'Matriz ' + IntToStr(MainForm.MDIChildCount + 1) + ':
Transpuesta de ' + A.caption ;
     Transpuesta.DimensionMatriz(A.renglones, A.columnas);

     t := matTranspuesta( A.laMatriz );
     matCopia( t, Transpuesta.laMatriz );
     Transpuesta.MostrarDatos;
end;

procedure TMainForm.ResolversistemadeEcuaciones1Click(Sender: TObject);
var MatrizA, VectorB, VectorX: TMDIChild;
    vb, vx : Variant;
    i : integer;
begin
     SelectorForm.InicializarCombos;

     with SelectorForm do
        If ShowModal = mrOK then begin
           MatrizA := TMDIChild(Matriz_A.Items.Objects[ Matriz_A.ItemIndex ]) ;
           VectorB := TMDIChild(Matriz_B.Items.Objects[ Matriz_B.ItemIndex ])  ;

           If Not MatrizA.esCuadrada then begin
              ShowMessage('La matriz A de la ecuación debe ser una matriz cuadrada');
              exit;
           end;

           If VectorB.columnas > 1 then begin
              ShowMessage('El vector de resultados B, debe tener una columna');
              exit;
           end;

           If MatrizA.Renglones <> VectorB.Renglones then begin
              ShowMessage('El sistema de ecuaciones A debe tener el mismo numero de
soluciones en B');
              exit;
           end;

           VectorX := TMDIChild.Create(Application);
           VectorX.Caption := 'Matriz ' + IntToStr(MainForm.MDIChildCount + 1) + ':
Solción del sistema de ecuaciones ' + MatrizA.caption + ' y ' + VectorB.caption ;
           VectorX.DimensionMatriz(1, VectorB.renglones);

           vb := MatAVectorCol( VectorB.laMatriz, 1);
           vx := MatAVectorCol( VectorX.laMatriz, 1);


            with MatrizA.DescomposicionLU1 do begin
              Descomponer;

              vx := SustitucionAtras_LU( LU, N, IDX, vb );

              MejorarSolucion( A, LU, N, IDX, vb, vx );
// Con una tiene certeza de 5 decimales
              MejorarSolucion( A, LU, N, IDX, vb, vx );
// Si quiere aumentar la precisión, con 2 es suficiente
            end;

            for i:=1 to VectorX.Renglones do
                VectorX.laMatriz[1,i] := vx[i];

            VectorX.MostrarDatos;

        end; // If ShowModal = mrOK;
```

```
end;


end.



unit Childwin;

interface

uses Windows, Classes, Graphics, Forms, Controls, ComCtrls, Grids, SysUtils,
  Menus, DescomposicionLU;

type
  TMDIChild = class(TForm)
    Matriz: TStringGrid;
    StatusBar1: TStatusBar;
    DescomposicionLU1: TDescomposicionLU;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormCreate(Sender: TObject);
    procedure MatrizSelectCell(Sender: TObject; Col, Row: Integer;
      var CanSelect: Boolean);
    procedure MatrizGetEditText(Sender: TObject; ACol, ARow: Integer;
      var Value: String);
    procedure MatrizSetEditText(Sender: TObject; ACol, ARow: Integer;
      const Value: String);
    procedure Nueva1Click(Sender: TObject);
    procedure Inversa1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    laMatriz : Variant;
    esCuadrada : Boolean;
    Renglones : Integer;
    Columnas : Integer;
    procedure DimensionMatriz ( col, ren : integer );
    procedure MostrarDatos;
  end;

implementation

{$R *.DFM}
Uses Main;


procedure TMDIChild.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;

procedure TMDIChild.FormCreate(Sender: TObject);
begin
     // Por defecto creamos una matriz de 3 x 3
     DimensionMatriz(3,3);
end;


//-----------------------------
//-- Métodos de la matríz
```

```
procedure TMDIChild.DimensionMatriz( col, ren : Integer );
begin
  renglones := ren;
  columnas  := col;

  If renglones <= 0 then renglones := 0;
  If columnas  <= 0 then columnas := 0;

  esCuadrada := (renglones = columnas);
  DescomposicionLU1.N := columnas;

  Matriz.ColCount := columnas + 1;
  MAtriz.RowCount := renglones + 1;
  StatusBar1.Panels[0].Text := IntToStr(ren) + ' x ' + IntToStr(col);

  laMatriz := varArrayCreate([1, columnas, 1,renglones], varDouble );
  MostrarDatos;
end;


procedure TMDIChild.MatrizSelectCell(Sender: TObject; Col, Row: Integer;
  var CanSelect: Boolean);
begin
     StatusBar1.Panels[1].Text := 'Col: ' + IntToStr(col) ;
     StatusBar1.Panels[2].Text := 'Ren: ' + IntToStr(row) ;
end;

procedure TMDIChild.MatrizGetEditText(Sender: TObject; ACol, ARow: Integer;
  var Value: String);
var d : double;
begin
     d := (laMatriz[ACol, ARow]); //FloatToStr(laMatriz[ACol, ARow]);
     Value := Format('%.3f', [d]);
end;

procedure TMDIChild.MatrizSetEditText(Sender: TObject; ACol, ARow: Integer;
  const Value: String);
var ld_val : Double;
begin
     if value = '' then exit;

     try     ld_val := StrToFloat(value)
     except  ld_val := 0.0
     end;
     laMatriz[ACol,ARow] := ld_val;

end;

procedure TMDIChild.MostrarDatos;
var i, j   : Integer;
    ld_val : Double;
begin
   for i:=1 to columnas do
       for j :=1 to renglones do begin
           ld_val := laMatriz[i, j];
           Matriz.Cells[i,j] := Format('%.3f', [ ld_val ]);
       end ;
end;


procedure TMDIChild.Nueva1Click(Sender: TObject);
begin
     MainForm.FileNewItemClick(Sender);
```

Conclusión

Tener programas sencillos de utilizar en forma de componentes, como parte del entorno de desarrollo del lenguaje, permitirá que centremos más nuestra atención en los algoritmos de los problemas que se nos presenten.

Inicialmente, para el desarrollo de las bibliotecas de funciones matemáticas reutilizables, se pensó en un lenguaje que adquiere matices de universal en estos días, Java; también en las nuevas tecnologías de componentes Active X de Microsoft.

Sin embargo, se optó por la opción de hacerlas en Delphi porque éste, además de crear los componentes nativos al lenguaje, puede convertir los componentes en controles Active X, extendiendo la capacidad de usar el código en otros lenguajes.

Los componentes desarrollados en el trabajo ejemplifican el uso de la programación orientada a objetos, sobre todo en los aspectos que se refieren a la encapsulación de la implementación del programa frente a la interfaz del usuario, y el uso de la herencia para crear variantes de un mismo componente alterando algunas propiedades de la clase heredada.

Al comenzar con el desarrollo de los componentes nos encontramos con el siguiente problema, los algoritmos de muchos de ellos tenían un manejo extensivo de operaciones con matrices de tamaño variable.

Es cierto que el problema puede resolverse utilizando lenguaje C, en su versión para Windows desarrollado también por Borland, C++ Builder, con capacidad para crear componentes; pero también lo es que con Delphi la programación es más clara y sencilla para efectos académicos.

Lenguaje C permite un manejo de matrices excelente debido a su calidad de operaciones con apuntadores, no así con las versiones de Pascal, incluido Delphi. Se tuvo que encontrar una alternativa para realizar tales manejos de memoria para las operaciones con matrices, como utilizar arreglos estáticos de pequeño tamaño, estructuras de datos como listas ligadas, y la programación de los componentes se hacía cada vez más complicada; hasta el descubrimiento del tipo de dato *Variant*.

Variant es un nuevo tipo de datos para Pascal, incluido en la versión de Delphi 3, este tipo de dato fue creado explícitamente para el desarrollo de componentes Active X y tiene como propiedad poder adquirir cualquier tipo de datos, incluso el de matrices de tamaño variable.

Con esto, se replantearon las operaciones de matrices a este tipo de dato, implicó volver a hacer algunas rutinas y crear nuevas para optimizar su uso. El resultado de estos cambios fue un desarrollo más sencillo y claro de los componentes.

El hecho de que existieron "retos", en nuestros intentos iniciales de organizar los componentes, nos alerta ante la necesidad de poner más esmero para cerciorarnos de que nuestros algoritmos sean correctos; de verificar si nuestro trabajo no ha sido desarrollado ya por alguien más y revisar los últimos avances en los lenguajes que utilicemos. Un buen programa inicia con un buen algoritmo y una buena recopilación de información.

Al desarrollar los componentes, nos encontramos con los beneficios de contar ya con la algunos de los procedimientos, desarrollados para diversas aplicaciones. Reutilizar este material sólo consistió en parametrizarlos y, a decir verdad, los cambios a los programas originales fueron mínimos y se refieren a la creación de componentes por su sintaxis especial que requieren.

Al desarrollar los ejercicios, nos encontramos con los beneficios de una programación más rápida, y que en un momento dado permite a programadores, con poca experiencia en el desarrollo de problemas matemáticos, realicen sistemas más complejos.

El desarrollo de componentes es un excelente medio para familiarizarse con el lenguaje en que se desarrollan, pero no limita su uso a otros lenguajes, pueden compartirse.

Bibliografía

AHO, Alfred V. "*Compiladores. Principios, técnicas y herramientas*". Addison - Wesley Iberoamericana. Delawere, E.U.A. 1990.

ACTON, F.S., "*Numerical Methods That Work*"; Mathematical Association of America, EUA Washington, 1990 edición corregida.

BERS, Lipman, Karal Frank. "*Cálculo*", Ed. Interamericana, 2ª edición, México 1985

BURDEN, Richard, Faires Douglas. "*Análisis Numérico*", Gpo. Editorial Iberoamericana, México 1985

DAHLQUIST, G., and Bjorck, A., "*Numerical Methods*". Prentice-Hall, EUA 1974.

FORSYTHE, G.E., Malcolm, M.A., and Moler, C.B., "*Computer Methods for Mathematical Computations*" Prentice-Hall, EUA 1977.

GHEZZI, Carlo. "*Conceptos de lenguajes de programación*". Diaz de Santos; Madrid; 1986

KATZAN, Harry Jr. "*Operating Systems*". Van Norand Reinhold Company, EUA 1973.

KRUSE, Robert L. "*Estructura de Datos y diseño de programas*", Ed. Prentice Hall, México 1988

MONRO, Donald. "*Curso dinámico de Pascal*". Mc Graw-Hill. México 1985.

NORTON, Peter, & Harley Hahn. "*Guide to Unix*". Bantam Computer Books, EUA 1991.

NORTON, Peter, Peter Aitken, Richard Witon "*The Peter Norton PC Programmer's Bible*". Microsoft Press, EUA 1993.

OSIER, Dan, Steve Grobman, Steve Batson. "*Teach Yourself Delphi 3 in 14 days*". Sams Publishing, EUA 1997.

PC Media. "*Optimización de los sistemas operativos (2)*". Editorial Ness, Méx. Año II No. 8.
RIDDERS, C.J.F. "*Advances in Engineering Software*", vol 4, no 2, pp 75-76
PRESS, William, Teukolsky Saul, et al. "Numerical Recipes in C : The Art of Scientific Computing". Cambridge University Press 2ª. edición 1993.

STOER, J, Bulirsch,R. "*Introduction to Numerical Analysis*". Springer-Verlag, EUA Nueva York, 1980.

WIRTH, Niclaus, G. "*Algorithms & Data Structures*". Prentice Hall, EUA 1986.