Métodos Numéricos con Octave

Por:
Luis Guillermo Bustamante
Camilo Duque Velásquez
Camilo Zuluaga
Asesor Docente
Francisco José Correa Zabala

Introducción

La primera motivación que se nos ocurrió al comenzar este proyecto fue: MatLab no puede ser el único, con único queríamos decir que MatLab era muy bueno para resolver problemas matemáticos y alguno de nosotros ya había trabajado con el en un semestre anterior y las experiencias fueron muy buenas, definitivamente MatLab es un lenguaje diseñado para el curso de Análisis Numérico; sin embargo volviendo a la pregunta inicial buscando en Internet le preguntamos a Google (El que todo lo sabe) que otro lenguaje existía para resolver este tipo de problemas matemáticos y nos encontramos con Octave entre otros, pero particularmente Octave, vimos que la sintaxis era muy parecida a MatLab, lo que le daba alguna ventaja sobre los otros, por otro lado vimos que podría funcionar bien en Windows, que también le dio otro punto a favor y finalmente y la que nos terminó de cautivar fue la cantidad de documentación que podíamos encontrar para Octave y obviamente gran parte de la de MatLab nos podría servir también; Ha y se nos olvidaba: como todas las cosas buenas de la vida, era Gratis!.
Bueno ya teníamos el lenguaje ideal para el curso, ahora lo que faltaba era un proyecto y fue la con ayuda de Pacho, el profesor, que nos animamos a escribir un manual sobre Octave y finalmente se convirtió en lo que es hoy por hoy. No algo demasiado sofisticado ni complicado ni algo demasiado simple, terminó siendo un trabajo realizado por estudiantes para estudiantes. Posiblemente le faltaron algunas cositas que se podrán pulir con el tiempo y con el trabajo de futuros estudiantes. Es por esto que este trabajo no puede quedar en vano, sería una excelente idea un libro donde no existiera un autor ni un coautor, sino mas bien una comunidad de estudiantes y profesores que pongan su granito de arena con un único objetivo en mente: El Conocimiento.

Objetivos

1. Instruir tanto al alumno como al docente en esta herramienta Open Source para el desarrollo de métodos numéricos y presentando a Octave como una herramienta viable cuando se está en proyectos de investigación con bajo presupuesto.

2. Mediante este manual de Octave se pretende dar una cercanía al alumno del curso de Análisis Numérico con un breve resumen y su aplicación en Octave para cada uno de los métodos vistos durante el curso. De tal manera desarrollar en el alumno habilidades de programación estructurada para la resolución de problemas matemáticos haciendo uso del lenguaje Octave .

3. Brindar una herramienta de apoyo que facilite el aprendizaje de un lenguaje de programación como lo es Octave para la realización de futuros trabajos que se puedan presentar tanto dentro del curso como en proyectos de investigación.

4. Motivar al estudiante al futuro aporte de este manual, presentando bien sea nuevos métodos numéricos para la resolución de problemas matemáticos o bien, para ampliar su contenido y que en un futuro se pueda convertir en un libro construido por estudiantes para estudiantes.

5. Presentar el manual de Octave como una guía de referencia,consulta y de fácil entendimiento tanto para estudiantes como para profesores y profesionales en áreas afines.

Parte I

Introducción a Octave

Capítulo 1

Manual de Instalación

1.1. Instalación en Linux

Para comenzar la instalación de octave en linux, se debe proceder primero a descargar los archivos. Actualmente existen 3 métodos para descargar el Octave:

Instalación de Octave utilizando apt

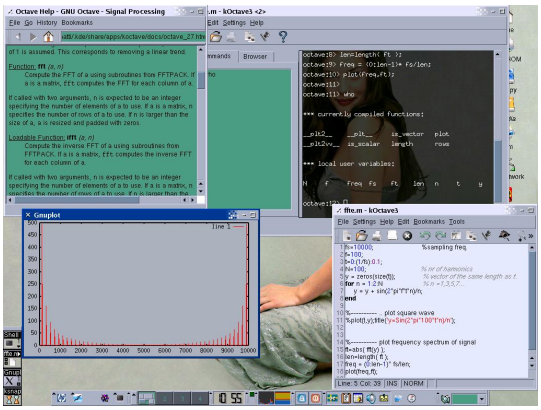
apt es el manejador de actualizaciones y descargas por excelencia para el sistema operativo Linux Debian, la aplicación puede ser descargada de la dirección http://packages.debian.org/stable/base/apt.html , el proceso de instalación de apt, no será explicado en este manual debido a que no es de nuestra relevancia; por ende daremos por hecho que el lector ya ha descargado esta herramienta y la tiene actualmente instalada y funcionando. Ahora, para proceder con la instalación de Octave simplemente ejecutamos como root el siguiente comando en una ventana de consola y asegurandonos que estamos conectados a Internet:
$apt-get install octave
Este comando busca en los repositorios y posteriormente compara los repositorios con los paquetes que tenemos instalados actualmente en el computador, para el caso de debian los paquetes tienen extención .deb . Ahora bien, luego de haber ejecutado este comando podemos proceder a continuación con la ejecución de octave desde una ventana de consola invocando el comando:
$octave
Es importante saber que apt funciona de la misma manera para muchas otras distribuciones de linux como Fedora Core, Red Hat y las distribuciones que están basadas en Debian.

Instalación de Octave utilizando yum

yum al igual que apt, es un manejador de actualizaciones para la distribución de linux basada en Red Hat: Yellow Dog, de aquí viene el nombre de esta herramienta *Yellowdog Update Manager* por sus siglas en inglés. Esta herramienta al igual que el apt puede ser descargada de la dirección http://www.fedora.us/wiki/FedoraHOWTO aquí también encontrará un pequeño FAQ sobre apt y yum y algunos de sus comandos básicos.
Ahora que hemos descargado e instalado yum procedemos a descargar Octave utilizando yum ejecutando el siguiente comando como usuario root:
$yum install octave
Al igual que apt, el funcionamiento de yum es relativamente parecido para el usuario final. Finalmente para ejecutar Octave ejecutamos el comando:
$octave

Instalación manual de Octave

Para algunos puristas de linux aún prefieren realizar instalaciones manuales de aplicaciones que descargar el paquete de instalación. Entiéndase por instalación manual la compilación de las fuentes. Para realizar este procedimiento nos dirigimos a la página web de octave http://www.octave.org/download.html y descargamos la versión que deseemos y con la extención tar.gz
Luego de haber descargado el archivo de extención .tar.gz procedemos a descomprimirlo y copiarlo en una carpeta. Nuevamente abrimos una sesión de shell o ventana de consola y nos movemos hasta el directorio donde guardamos el archivo descargado utilizando el comando *$cd*. Y ejecutamos el comando *$tar -zxvf octave.xx.tar.gz* donde xx es el número de la versión que descargó, luego de haber descomprimido el archivo nos movemos a la carpeta que fue creada por el descompresor y procedemos a ejecutar los comandos
$./configure
$make
y como usuario root ejecutamos: $make install
Si todo resulto bien y no hubo dependencias sin resolver podemos ejecutar octave de la misma forma como se ejecuto cuando instalamos octave con yum y apt:
$octave



Instalación de un entorno gráfico para Octave

En esta sección nos dedicaremos exclusivamente a la instalación de una aplicación llamada KOctave; KOctave es una aplicación que facilita el trabajo con Octave y corre sobre el manejador de escritorio KDE. Para comenzar con la instalación comenzamos con descargar el programa desde la página web de KOctave http://bubben.homelinux.net/ matti/koctave , luego de haber descargado el paquete procedemos con la descompresión e instalación de KOctave con los siguientes comandos:
$tar -zxvf koctave.xx.tar.gz
$cd koctave
$./configure
$make
y somo super usuario ejecutamos el siguiente comando
$make install

ahora luego de que hemos instalado KOctave nos dirigimos a la carpeta koctave3 y ejecutamos el siguiente comando.
$./koctave3
Y obtenemos una ventana como esta donde estamos listos para comenzar con Octave!

1.2. Instalación en Windows

Como Octave es una herramienta desarrollada inicialmente para Linux, entonces debemos usar ciertas ayudas adicionales para trabajar este programa en Windows de modo que podamos trabajar de forma gráfica, pero si no

se requiere de ayudas gráficas lo mejor es trabajar con el octave+forge.

1.2.1. Instalación de octave+forge

Este es el modo básico de octave para Windows con el cual podremos realizar todos los cálculos que necesitemos.

1. Vaya a la página http://sourceforge.net/projects/matlinks
2. Seleccione el octave versión para Windows.
3. Cuando haya terminado la descarga haga doble-click en el archivo ejecutable.
4. Siga las instrucciones que le indica el instalador

1.2.2. Instalación de Cygwin

1. Vaya a la página http://www.cygwin.com
2. Haga clic en el enlace Install or update now!
3. Guarde el archivo ejecutable en una carpeta donde lo pueda ejecutar sin problemas.
4. Cuando el archivo haya terminado de bajar ejecútelo, haciendo doble clic.
5. Seleccione la opción "Install from Internet".
6. Seleccione la carpeta en donde quiere que se instale las herramientas del Cygwin.
7. Tendrá la opción de instalar para todos los usuarios o lo puede instalar solo para el usurario de ese momento.
8. Seleccione la opción de instalar en modo Unix.
9. Seleccione la carpeta en donde desea que el Cygwin descargue los componentes necesarios para su instalación(Este directorio será temporal).
10. Seleccione su conexión a Internet, sino tiene Firewalls instalados en su equipo, escoja la opción "Direct Connection".
11. Escoja el sitio desde donde desea bajar los componentes del Cygwin.

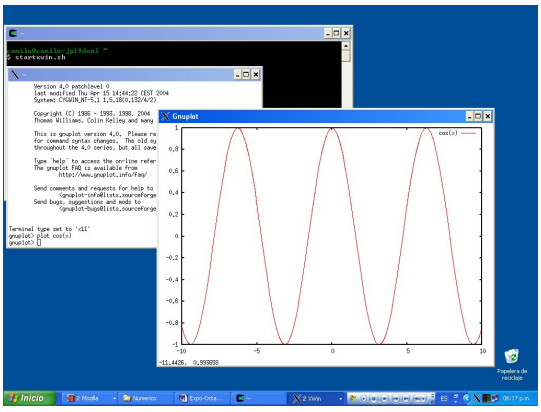
12. Seleccione los paquetes que sean de su interés, el Cygwin trae una gran variedad de utilidades.

13. Una vez hayan bajado los archivos de instalación, de clic en finalizar.

NOTA: Recuerde que durante todo el proceso de instalación usted debe estar conectado a Internet.

1.2.3. Corriendo Octave

Después de instalado en Cygwin necesitamos dar unos comandos para hacer que corra el octave y así probar todo su potencial gráfico.
Pruebe correr el Octave, dando el comando Octave en la shell del Cygwin. Cuando se encuentre en la consola ejecute el siguiente comando: startxwin.sh, luego ingrese el siguiente comando en la nueva consola: gnuplot, ya aquí estará preparado para graficar las funciones que desee.
En la Figura 1 se puede apreciar un ejemplo de cómo debería ejecutarse la aplicación, realizando la gráfica para la función F(x) = cos (x).



Capítulo 2

Aspectos Generales

2.1. Comenzando con Octave

Octave es un lenguaje de alto nivel poderoso y flexible diseñado para computaciones numéricas. Viene provisto de un manual de mas de 250 paginas de fácil lectura y un sistema de ayuda en linea basado en el sistema GNU Info. Aunque la documentación no lo explicite Octave esta destinado a ser un clon de Matlab, o Matlab compatible al menos. Octave es probablemente el lenguaje de alto nivel libre mas parecido a Matlab. El paquete octave-forge ha logrado incluir aun mas compatibilidad con Matlab.[Cas]

2.2. Aritmética

El manejo de la operaciones básicas en Octave se hace de una manera muy simple, en los ejemplos siguientes se puede observar como se realizan estas operaciones.
Ejemplo 2.1.1
Si queremos realizar la suma de dos números cualquiera.
5 + 7
Simplemente realizamos lo siguiente

octave:1>5 + 7
ans = 12
Se trabaja igual que con la suma
octave:3>2 / 3
ans = 0.66667
Como se puede observar en los ejemplos anteriores, los resultados son prece-

didos por la palabra ans. Esta es la forma en que Octave devuelve todos sus resultados.
Las operaciones de resta (-) y multiplicación (*), se trabajan de igual forma que la suma y la división.

Como todo lenguaje de alto nivel Octave permite la asignación de variables, pero esta se realiza de una forma mucho mas sencilla que en muchos otros lenguajes, debido a que no hay que especificarle el tipo de la variable que se le va a introducir ya que él mismo lo puede reconocer.
Ejemplo 2.1.2
A continuación se mostrará una serie de ejemplos de cómo introducir variables en Octave y como él las recibe:
octave:6>x = 3 + 4
x = 7

octave:7>x
x = 7

octave:8>x;

octave:9>y = 1 / 2;

octave:10>y
y = 0.50000

octave:11>z = 4 + 3i
z = 4 + 3i
Como se vio en el ejemplo anterior con los números, Octave permite trabajar los Strings de forma similar, sin tener que cambiar o redefinir los tipos de las variables.
Ejemplo 2.1.3

```
octave:1> d = 5
d = 5

octave:2> d
d = 5

octave:3> d = "casa"
d= casa
```

```
octave:4> d
d = casa
```

Con este ejemplo, se puede ver que la variable d fue asignada primero con un número y luego se cambió por un String. Aquí se puede ver la facilidad del manejo de las variables con Octave.

Esta es la función que maneja Octave para controlar de forma ordenada la salida de datos. Esta función permite que se ingresen Strings y variables numéricas.[Amu]
Ejemplo 2.1.4

```
octave:1> disp("El valor de Pi es:"), disp(pi)
El valor de Pi es 3.1416
octave:2>
También permite que se le ingresen operaciones aritméticas:

octave:1> disp(8 + 10)
18
```

2.3. Manipulación de matrices y vectores

Matrices Como ya se comentó, el tipo básico de datos es la matriz bidimensional de números complejos en punto flotante. Las matrices son rectangulares, por lo que todas las filas tienen el mismo número de columnas y todas las columnas el mismo número de filas.

La matrices pueden tener cualquier tamaño y pueden ser ampliadas o reducidas dinámicamente, es decir, no es necesario declarar el tamaño previamente, como ocurre en otros lenguajes de programación, ya que el entorno se encarga de conseguir la memoria necesaria.
Las matrices pueden generarse, a groso modo:

- mediante definición explícita
- como resultado de operaciones
- como resultado devuelto por una función

Definición explícita
Se realiza indicando los elemento entre corchetes ([ ]). La coma (,) o el espacio se utiliza para separar los elementos de una fila. El punto y coma (;) o el salto de línea se utiliza para separar una fila de la siguiente. Siempre es recomendable la utilización de la coma y punto y coma en vez del espacio y el salto de línea, ya que estos últimos pueden llevar a errores difíciles de localizar.

para representar la matriz $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ escribiremos:

A=[1, 2, 3; 4, 5, 6; 7, 8, 9]

Las elementos pueden ser constantes, expresiones u otras matrices, siempre y cuando de lugar a una matriz de dimensiones correctas.

para construir la matriz $B = \begin{pmatrix} 1 & 2 & 3 & 11 \\ 4 & 5 & 6 & 12 \\ 7 & 8 & 9 & 13 \end{pmatrix}$ podemos escribir:

B=[A, [11; 6*2; 10+3]]

ya que con esto añadimos a la matriz A una columna con los elementos 11, 12 y 13 para generar B.

La matriz vacía se define con dos corchetes sin elementos: []

Rangos
Genera vector fila de números equiespaciados. Tiene las siguientes formas:
exp1:exp3:exp2
exp1:exp2
En este segundo caso se asume exp3=1

El vector generado estará formado por: [exp1, exp1+exp3, exp1+2*exp3 ....], el elemento final será aquel de la serie que sea: menor o igual que exp2 si exp3 es positiva; mayor o igual si exp3 es negativa.
el rango 1:2:8 es equivalente a [1, 3, 5, 7]
el rango 10:-3:-5 es equivalente a [10, 7, 4, 1, -2, -5]
el rango 1:-1:5 dará lugar a la matriz vacía []

Funciones que generan matrices Existe una serie de funciones que se utilizan muy a menudo para generar matrices de ciertas características como son las siguientes:

- **zeros(n)**:crea una matriz cuadrada de dimensión n (con n filas y n columnas) con todos sus elementos a 0.
- **zeros(n,m) zeros([n,m])**: crea una matriz de n filas y m columnas con todos sus elementos a 0.
- **ones** Genera una matriz con todos sus elementos a 1. Tiene las mismas formas de invocación que zeros: ones(n), ones(n,m), ones([n,m]).
- **eye** devuelve una matriz con sus elementos a 0 salvo en la diagonal principal, o primera diagonal, cuyos elementos están a 1. En el caso de que sea cuadrada representa la identidad para la dimensión correspondiente. Tiene las mismas formas de invocación que zeros.
- **rand** devuelve una matriz de elementos aleatorios con distribución normal (gausiana de media 0 y varianza 1). Tiene la mismas formas de invocación que zeros.
- **linspace(inicial,final,n)**:devuelve un vector de elementos equiespaciados a partir del valor inicial y hasta el final. Habrá tantos elementos como indique n, o 100 si este parámetro no se especifica. A diferencia de los rangos, en que se conoce la separación de los elementos pero no su número, en este caso se conoce el número de elementos pero no su separación.
- **logspace(inicial,final,n)**:similar a linspace pero con una separación logarítmica entre los elementos del vector. En este caso inicial y final representan los exponentes de 10 para el valor inicial y final del vector. En el caso de que final valga π el valor final será π y no 10^{π}.

Indexación Cuando tenemos una variable matriz, representada por su identificador, es posible especificar submatrices de esta para operar con ellas. Esto se realiza mediante las expresiones de indexación que consisten en:

- NombreVariable(indice-filas, indices-columnas) para el caso de matrices
- NombreVariables(indice-elemento) para el caso de vectores

Los posibles tipos de índices son:

Escalar
Entero que selecciona sólo la fila/columna indicada. Las filas/columnas se numeran comenzando por el número 1.

La expresión A(3,1) representa al elemento de la tercera fila, primera columna. En la matriz A anterior su valor es 7.

selecciona todas las filas/columnas

A(2,:) representa toda la segunda fila

A(:,1) representa toda la primera columna

A(:,:) representa a todas las filas y columnas, es decir, a toda la matriz

vector se selecciona cada una de las filas/columnas indicadas por los elementos del vector y en el orden indicado por este.

A(2,[3 1]) representa la fila 1 columnas 3 y 1. En nuestro ejemplo devolverá [6,4]

A([2 1 2],1) selecciona el primer elemento de la fila 2 ^{a} la 1 ^{a} y nuevamente la 2 ^{a}. En nuestro ejemplo devolverá [4; 1; 4]

A(3:-1:1,:) devuelve la matriz con el orden de las filas invertido. En nuestro ejemplo [7,8,9;4,5,6;1,2,3]

lógico
cuando el índice es un vector de ceros y unos de la misma longitud que número de filas/columnas se seleccionan únicamente aquellas que tienen el elemento correspondiente a 1. No suele utilizarse directamente sino a través de las operaciones de relación y lógicas. En caso de conflicto con el índice vector (por que todos los elementos son 1), por defecto, se supone el vector.

A([1 0 1],3) se selecciona el tercer elemento de las filas 1 ^{a} y 3 ^{a}, ya que el elemento 1 ^{a} y 3 ^{a} del vector están a 1.

A(:,[1 1 1]) como entra en conflicto con el índice vector selecciona la 1 [a] columna 3 veces.

2.4. Polinomios

En octave los polinomios se representan mediante un vector con los coeficientes en orden descendente. Dado el vector p=[c1,c2, ... cN], éste representa al polinomio $p(x) = c_1x^{N-1} + c_2x^{N-2} + ... + c_{N-1}x + c_N$

Las funciones que realizan operaciones en polinomios son:

- **polyval(p,X)**: evalúa el polinomio para todos los elementos de X.
- **polyvalm(p,A)**: para A matriz cuadrada, evalúa el polinomio en sentido matricial.
- **roots(p)**: obtiene las raíces del polinomio.
- **poly(A)**: siendo A matriz cuadrada, devuelve el polinomio característico.
- **poly(r)**: siendo r un vector, devuelve el polinomio cuyas raíces son los elementos de r.
- **conv(p,s)**: devuelve el producto de dos polinomios.
- **[c,r]=deconv(x,y)**: devuelve el cociente y el resto de la división del polinomio x entre el y.
- **polyderiv(p)**: devuelve el polinomio derivada.
- **polyfit(x,y,n)**: devuelve el polinomio de orden n que mejor se ajusta, en mínimos cuadrados, a los puntos formados por (x,y).
- **[r,p,k,e]=residue(num,den)**: calcula la descomposición en fracciones simples del cociente del polinomio num entre el den. r son los residuos (numeradores), p son los polos (denominadores), k es el cociente (si grado del numerador mayor que el del denominador), y e son los exponentes para cada denominador. Es decir:
$\frac{num(s)}{den(s)} = k(s) + \frac{r(1)}{(s-p(1))^{e(1)}} + \frac{r(2)}{(s-p(2))^{e(2)}} + ...$

2.5. Graficando con 2 y 3 variables

Función : plot (args)

Esta función produce gráficas en dos dimensiones. Muchas combinaciones de argumentos son posibles. La forma más simple es:

plot (y)

Donde el argumento es tomado de una serie de coordenadas y y las coordenadas x son tomadas como los indices de los elementos, comenzando con 1.

Si más de un argumento es dado, entonces son interpretados así:

plot (x, y, fmt ...)

donde y fmt son opcionales, y cualquier numero de argumentos dados son tomados como:
- Si un solo argumento es dado, es tomado como una serie de coordinadas y y las coordinadas x son tomadas como los indices de los elementos, empezando con 1.
- Si el primer argumento es un vector y el segundo una matriz, el vector es dibujado versus las columnas (o filas) de la matriz.
- Si el primer argumento es una matriz y el segundo un vector, las columnas (o filas) de la matriz son dibujadas versus el vector.
- Si ambos argumentos son vectores, los elementos de y son dibujados versus los elementos de x.
- Si ambos argumentos son matrices, las columnas de y son dibujadas versus las columnas de x. En este caso ambas matrices deben tener el mismo numero de filas y columnas.
- Si ambos argumentos son escalares, un solo punto es dibujado.

2.6. Ciclos y Sentencias

La forma más general de la sentencia if permite la combinación de múltiples decisiones en un solo estamento. Se ve como esta:

```
if (condición)
  then-body
```

```
elseif (condición)
  elseif-body
else
  else-body
endif
```

Forma general de la sentencia switch:

```
switch expression
  case label
    command_list
  case label
    command_list
  ...

  otherwise
    command_list
endswitch
```

La sentencia while es el caso más simple de un ciclo en Octave. Este se ejecutará repetidamente hasta que la condición sea verdadera.

```
while (condición)
  body
endwhile
```

El do-until en Octave se ve de esta forma, este es como un while solo que asegura almenos una ejecución del cuerpo del estamento.

```
do
  body
until (condición)
```

La sentencia for hace más conveniente la cuenta de iteraciones en un loop. La forma general es así:

```
for var = expression
  body
endfor
```

2.7. Programando con Octave

2.7.1. Ficheros de comandos

Es posible ejecutar comandos que están en un fichero. Estos ficheros son de texto normal y deben de tener la extensión '.m', por lo que suele llamárselas también ficheros-m.

En estos ficheros es posible poner comentarios que comienzan por el carácter # y continúan hasta el final de la línea. Las líneas de comentario que aparecen juntas al principio del fichero o tras la definición de una función representan la ayuda, y son presentadas si ejecutamos: help nombre

2.7.2. Ficheros de función

Son ficheros que contienen únicamente la definición de una función. Se deben llamar con el mismo nombre que la función. Cuando se invoca a una función, Octave la busca en el espacio de trabajo actual. Si no existe allí, busca si existe un fichero con ese nombre y la extensión '.m' en el directorio actual y el los directorios de una lista (LOADPATH). Si existe lo carga en memoria y ejecuta la función. En las siguientes invocaciones Octave sólo comprueba si el fichero ha sido modificado, si es así lo carga nuevamente, en caso contrario utiliza la copia existente en memoria.

2.7.3. Ficheros de script

Estos ficheros contiene comandos normales que se ejecutan como si fueran tecleados directamente en el entorno, es decir, trabajan sobre el espacio de trabajo global, por lo que pueden acceder a las variables existentes, modificarlas o crear nuevas.
Para invocarlo basta poner el nombre del fichero, sin la extensión '.m'.

Es posible definir funciones dentro de un fichero script, pero el primer comando ejecutable no puede ser una definición de función ya que, en caso contrario, el fichero se considerará de función.

Parte II

Métodos Numéricos

Capítulo 3

Raíces de Ecuaciones

3.1. Métodos Cerrados

3.1.1. Método de Bisección

El método de bisección, conocido también como de corte binario, de partición de intervalos o de Bolzano, es un tipo de busqueda incremental en el que el intervalo se divide siempre a la mitad. Si la función cambia de signo sobre el intervalo, se evalúa el valor de la función en el punto medio. La posición de la raíz se determina situándola en el punto medio del subintervalo, dentro del cual ocurre un cambio de signo. El proceso se repite hasta obtener una mejor aproximación. Para el método de bisección tenemos el siguiente código realizado en Octave y compatible con MatLab.

Pseudocódigo

```
Entradas: a, b, Nmax, tol

fa = f(a)

if fa = 0
  "a es raíz aproximada"
  stop
end if

fb = f(b)

if fb = 0
  "b es raiz aproximada"
```

```
  stop
end if

if signo(fa) = signo (fb)
  stop
end if

error = b-a
c = (a+b)/2

fc = f(c)

k = 1

while k <= Nmax and Error > tol y fc !=
  aux = c
  if signo(fc) != signo (fa)
    b = c
    fb = fc
  else
    a = c
    fa = fc
  end if

  c = (a + b) / 2
  fc = f(c)
  Error = |c-aux|
  k = k + 1
end while

if fc = 0
  "raiz aproximada x = c"
else if error <= tol
    "raiz aproximada x=c con error = error"
else
  "superado #maximo de iteraciones"
end

stop
```

Código

```
function [c,yc,err,P] = bisect(a,b,delta)
f = input("ingrese f(x): ", "s");
f = inline(f);
err = 0;
P = [a b err];
ya = feval(f,a);
yb = feval(f,b);
if ya*yb > 0, break, end
max1 = 1 + round((log(b-a)-log(delta))/log(2));
for k=1:max1,
  c  = (a+b)/2;
  yc = feval(f,c);
  if  yc == 0,
    a = c;
    b = c;
  elseif  yb*yc > 0,
    b = c;
    yb = yc;
  else
    a = c;
    ya = yc;
  end
  err = abs(b-a)/2;
  P = [P;a b err];
  if b-a < delta, break, end
end
c  = (a+b)/2;
yc = feval(f,c);
err = abs(b-a)/2;

disp(c);
disp(yc);
disp(err);
disp('  X0       X1        ERROR');
disp(P);
```

```
end
```

3.1.2. Regla Falsa

Otro método que comúnmente se emplea es el de Regula Falsi. Este método también tiene otras denominaciones, como son: Regla falsa, Posición falsa o Interpolación Lineal. Su nombre original que esta en Latín, denota su antigüedad. La idea del método es bastante similar al del método de Bisección. Requiere un intervalo que cumpla los mismos supuestos que el método de Bisección. En lugar de obtener el punto medio en cada iteración, el método busca reemplazar la función original por otra a la cual sea más simple localizar su raíz. Dado que comenzamos con solo un intervalo, es decir, sólo tenemos 2 puntos, buscamos la curva más simple que pase por estos 2 puntos. Lógicamente usamos una línea recta. Entonces en vez de obtener puntos medios en este método se halla las raíces de las rectas que pasen por los puntos que determinen nuestros intervalos.

Pseudocódigo

```
Entradas: a, b, Nmax, tol

fa = f(a)

if fa = 0
  "a es raiz aproximada"
  stop
end if

fb = f(b)

if fb = 0
  "b es raiz aproximada"
  stop
end if

if signo(fa) = signo (fb)
  stop
end if

error = b-a
```

```
c = (a+b)/2

fc = f(c)

k = 1

while k <= Nmax and Error > tol y fc !=
  aux = c
  if signo(fc) != signo (fa)
    b = c
    fb = fc
  else
    a = c
    fa = fc
  end if

  c = a - fa*(b-a)/(fb-fa)
  fc = f(c)
  Error = |c-aux|
  k = k + 1
end while

if fc = 0
  "raiz aproximada x = c"
else if error <= tol
    "raiz aproximada x=c con error = error"
else
  "superado #maximo de iteraciones"
end

stop
```

Código

```
function [c,yc,err,P] = regula(a,b,delta,max1)

f = input("ingrese f(x): ", "s");
f = inline(f);
epsilon = 1.0842e-19;
```

```
P = [a b 0 0 0];
ya = feval(f,a);
yb = feval(f,b);
if ya*yb > 0, break, end
for k=1:max1,
  dx = yb*(b - a)/(yb - ya);
  c  = b - dx;
  ac = c - a;
  yc = feval(f,c);
  if  yc == 0,
    break;
  elseif  yb*yc > 0,
    b = c;
    yb = yc;
  else
    a = c;
    ya = yc;
  end

  P = [P;a b c yc err];
  dx = min(abs(dx),ac);
  err = abs(dx);
  if abs(dx) < delta,   break, end
  if abs(yc) < epsilon, break, end
end
err = abs(dx);

disp('    X0      X1       Raiz    FunctValor  Error')
disp(P)

end
```

3.1.3. Búsqueda por incrementos

Además de verificar una respuesta individual, se debe determinar si se han localizado todas las raíces posibles. Como se mencionó anteriormente, por lo general una gráfica de la función ayudará a realizar dicha tarea. Otra opción es incorporar una búsqueda incremental al inicio del programa. Esto consiste en empezar en un extremo del intervalo de interés y realizar evaluaciones de

la función con pequeños incrementos a lo largo del intervalo. Si la función cambia de signo, se supone que la raíz esta dentro del subintervalo.

Pseudocódigo

```
Entradas: x0, d, nmax

y0 = f(x0)
if y0 != 0
  k = 1
  x1 = x0 + d
  y1 = f(x1)
  while y0*y1>0 and  k<=nmax
    y0 = y1
    x0 = x1
    x1 = x0 + d
    y1 = f(x1)
    k = k+1
  end while
end if

if y1 = 0
  print "Es raiz en x1"
else if y1*y0 < 0
  print "Hay una raiz entre x0 y x1"
else
  print "No hay raiz"
end if

stop
```

3.2. Métodos Abiertos

3.2.1. Punto Fijo

El método de punto fijo se aplica para ecuaciones de la forma $x = g(x)$. Se parte de un punto inicial x_0 y se aplica la fórmula $x_{n+1} = g(x_n)$ para $n \geq 0$ en caso de que exista $lim_{n\to\infty} x_n = p$, si g es continua en este valor p se tiene que: $p = lim_{n\to\infty} x_n = lim_{n\to\infty} g(x_{n-1}) = g(lim_{n\to\infty} x_{n-1}) = g(p)$ donde p sería la solución buscada.

Pseudocódigo

```
Entradas: p0, tol, N0

i = 1

while i <= N0
  p = f(p)
  if |p - p0| < tol
    print p
    stop
  end if

  i = i + 1
  p0 = p
end while

print "el metodo fracaso despues de N0 iteraciones"
stop
```

Código

```
function [p0,err,P] = fixpt(p0,tol,max1)

g = input("ingrese f(x): ", "s");
g = inline(g);
P(1) = p0;

P2(1)= p0;
```

```
P2(2)= 0;
P2(3) =0;

err = 1;
relerr = 1;
p1 = p0;
for k=1:max1,
  p1 = feval(g,p0);
  err = abs(p1-p0);
  relerr = err/(abs(p1)+eps);
  if (err<tol) | (relerr<tol), break; end
  p0 = p1;
  P(k+1) = p1;
  P2=[P2;p0 err relerr];
end
disp('   IniVal   Error   RelError  ');
disp(P2);
end
```

3.2.2. Método de Newton-Raphson

Una de las formas mas utilizadas para encontrar raices es la formula de Newton-Raphson. Si el valor inicial para la raíz es x_i, entonces se puede trazar una tangente desde el punto $[x_i, f(x_i)]$ de la curva. Por lo común el punto donde esta tangente cruza al eje x representa una aproximación mejorada de la raíz.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Pseudocódigo

```
Entradas: p0, tol, N0

i = 1

while i <= N0
  p = p0 - f(p)/f'(p0)

  if |p - p0| < tol
    print p
    stop
```

```
  end if

  i = i+1
  p0 = p
end while

print "el metodo fracaso despues de N0 iteraciones"
stop
```

Código

```
function [p0,y0,err,P] = newton(p0,delta,max1)

f = input("ingrese f(x): ", "s");

epsilon = 1.0842e-19;


f = inline(f);
df = input("Ingrese f'(x)", "s");
df = inline(df);
P(1) = p0;
P(2) = 0;
P(3) = 0;
P(4) = 0;
P(5) = 0;
y0 = feval(f,p0);
for k=1:max1,
  df0 = feval(df,p0);
  if df0 == 0,
    dp = 0;
  else
    dp = y0/df0;
  end
  p1 = p0 - dp;
  y1 = feval(f,p1);
  err = abs(dp);
  relerr = err/(abs(p1)+eps);
  p0 = p1;
  y0 = y1;
  P = [P;p1 y0 df0 err relerr];
```

```
  if (err<delta)|(relerr<delta)|(abs(y1)<epsilon), break, end

end
disp('    Xi    F(Xi)    F'(Xi)    ErrorAbs   ErrorRel   ');
disp(P);

disp('Error Absoluto: '),disp(err),
disp('  Error Relativo: '), disp(relerr);
end
```

3.2.3. Método de la Secante

El método de la secante es un método que permite evaluar las raices de funciones cuya derivada es difícil de calcular. En dichos casos, la derivada se puede aproximar mediante una diferencia finita dividida hacia atrás. Y se obtiene de este modo la siguiente fórmula iterativa.

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1}-x_i)}{f(x_{i-1})-f(x_i)}$$

Pseudocódigo

```
Entradas: xo, x1,ea, er, iteraciones

if f(xo) == 0
    print "xo es raiz"

else
    c=x1-(f(x1)*(x1-xo))/((f(x1))-f(xo))
    error1 = ea + 1
    error2 =  er + 1
    iter = 1;


    er1=error1

    while f(c)!=0 and error2 > er and iter < iteraciones and error1>ea
        xo = x1
        x1 = c
        c=x1-(f(x1)*(x1-xo))/((f(x1))-f(xo))
```

```
        error1 = abs(c - xo)
        error2 = abs(c - xo)/c
        A1=error1/er1
        A2=error1/(er1)^2
        er1=error1
        iter = iter + 1

    end while


    print "Iteraciones  Raiz   error relativo   error absoluto  A1 A2"

end if

if f(c) !=0
    print "Raiz exacta: c"
else if error1 > ea
    print "Raiz con error admitido: c"

else if error2>er
    print "Raiz con error relativo admitido: c"

else
   print "Pocas iteraciones, raiz parcial c"
end if

stop
```

Código

```
function [p1,y1,err,P] = secant(p0,p1,delta,max1)

f = input("ingrese f(x): ", "s");
f = inline(f);
epsilon = 1.0842e-19;
P(1) = p0;
P(2) = p1;
P(3) = 0;
P(4) = 0;
P(5) = 0;
```

```
P(6) = 0;
y0 = feval(f,p0);
y1 = feval(f,p1);
for k=1:max1,
  df = (y1-y0)/(p1-p0);
  if df == 0,
    dp = 0;
  else
    dp = y1/df;
  end
  p2 = p1 - dp;
  y2 = feval(f,p2);
  err = abs(dp);
  relerr = err/(abs(p2)+eps);
  p0 = p1;
  y0 = y1;
  p1 = p2;
  y1 = y2;
  P = [P;p0 p2 p1 y1 err relerr];
  if (err<delta)|(relerr<delta)|(abs(y2)<epsilon), break, end
end
disp('   P0       P2       Xi     F(Xi)    ErrAbs   ErrorRelativo');
disp(P);
disp('Error Absoluto:');
disp(err);
disp('Error Relativo:');
disp(relerr);
end
```

3.3. Raíces de Polinomios

3.3.1. Método de Mûller

El método de Mûller es similar al método de la secante, pero a diferencia de éste; el método de Müller hace uso de una parábola para aproximar a la raíz. El método consiste en obtener los coeficientes de la parábola que pasan por los tres puntos, Dichos coeficientes se sustituyen en la fórmula cuadrática para obtener el valor donde la parábola intersecta al eje x; es decir, la raíz estimada. La aproximación se facilita al escribir la ecuación de la parábola en una forma conveniente.

Pseudocódigo

```
Entradas: x0, x1, x2, tol, N

h1 = x1 - x0
h2 = x2 - x1
d1 = (f(x1)-f(x0))/h1
d2 = (f(x2)-f(x1))/h2
d = (d2 - d1)/(h2 + h1)
i = 3

while i <= N
  b = d2 + h2*d
  DD = (b^2 - 4*f(x2)*d)^(1/2)

  if |b - DD| < |b + DD|
    E = b + DD
  else
    E = b - DD
  end if

  h = -2*f(x2)/E
  p = x2 + h

  if |h| < tol
    print p
    stop
  end if

  x0 = x1
  x1 = x2
  x2 = p
  h1 = x1 - x0
  h2 = x2 - x1
  d1 = (f(x1) - f(x0))/h1
  d2 = (f(x2) - f(x1))/h2
  d = (d2 - d1)/(h2 + h1)
  i = i + 1

end while
```

```
print "el metodo fallo despues de N iteraciones"
stop
```

Código

```
function [p2,y2,err,P] = muller(p0,p1,p2,delta,max1)

f = input("ingrese f(x): ", "s");
f = inline(f);
epsilon = 1.0842e-19;
P(1) = p0;
P(2) = p1;
P(3) = p2;
P(4) = 0;
P(5) = 0;
y0  = feval(f,p0);
y1  = feval(f,p1);
y2  = feval(f,p2);
for k=1:max1,
  h0 = p0 - p2;
  h1 = p1 - p2;
  c  = y2;
  e0 = y0 - c;
  e1 = y1 - c;
  det1 = h0*h1*(h0-h1);
  a  = (e0*h1 - h0*e1)/det1;
  b  = (h0^2*e1 - h1^2*e0)/det1;
  if  b^2 > 4*a*c,
    disc = sqrt(b^2 - 4*a*c);
  else
    disc = 0;
  end
  if b < 0, disc = - disc; end
  z = - 2*c/(b + disc);
  p3 = p2 + z;
  if  abs(p3-p1) < abs(p3-p0),
    u = p1;
    p1 = p0;
    p0 = u;
    v = y1;
```

```
    y1 = y0;
    y0 = v;
  end
  if  abs(p3-p2) < abs(p3-p1),
    u = p2;
    p2 = p1;
    p1 = u;
    v = y2;
    y2 = y1;
    y1 = v;
  end
  p2 = p3;
  y2 = feval(f,p2);

  err = abs(z);
  relerr = err/(abs(p3)+eps);
  P = [P;0 0 p2 y2 err relerr];
  if (err<delta)|(relerr<delta)|(abs(y1)<epsilon), break, end
end
disp(P);
end
```

Capítulo 4

Solución Numérica de Sistemas de Ecuaciones

4.1. Eliminación de Gauss

La técnica que se describe en esta sección se conoce como la eliminación de Gauss, ya que implica una combinación de ecuaciones para eliminar incógnitas. Aunque éste es uno de los métodos más antiguos para resolver ecuaciones lineales simultáneas, continúa siendo uno de los algoritmos de la mayor importancia, y la base para resolver ecuaciones lineales en muchos paquetes de software populares.

4.1.1. Eliminación de Gauss Simple

Esta sección presenta las técnicas sistemáticas para la eliminación hacia adelante y la sustitución hacia atrás que la eliminación Gaussiana comprende, se requiere de algunas modificaciones para obtener un algoritmo confiable. En particular, el programa debe evitar la división entre cero. A continuación encontramos el algoritmo en Octave para crear la matriz triangular superior.
Pseudo Código

```
// Eliminación hacia adelante
Haga (K = 1, n -1)
Haga(I = K + 1, n)
Factor = a (i, k)/ a(k,k)
Haga (j = K + 1 hasta n)
a(i,j) = a (i, j) - Factor * a(k, j)
fin
```

```
b (i) = b (i) - Factor * b(k)
fin
fin

//Sustitución hacia atras

X (n) = b (n) / a (n,n)
Haga (I = n - 1, 1, - 1)
Sum = 0
Haga(j = I + 1, n)
Sum = Sum + a(i, j) * X (j)
Fin
X(i) = (b (i) - Sum) / a(i,j)
Fin
```

Código

```
function [M,L] = gauss(A)
[m,n] = size(A);
L = eye(m)
M = A
for i = 1:m-1
   if M(i,i) == 0 error('Pivote es 0'); return; end
   for k = i+1:m
       L(k,i) = M(k,i)/M(i,i)
   for j = i+1:n
       M(k,j) = M(k,j) - L(k,i)* M(i,j);
   end
   for j = 1:i
       M(k,j) = 0;
   end
   M
    end
end
end
```

Luego de que se tiene la matríz triangular superior procedemos a realizar la sustitución hacia atrás. El algorítmo lo encontramos a continuación

```
function  X = backsub(A,B)
# --------
```

```
#  Llamada
#    X = backsub(A,B)
#  Parametros
#    A   Matriz de coeficientes triangular
#  superior obtenida de gauss(A)
#    B   Vector lado derecho de la ecuacion
#  Devuelve
#    X   Vector de Solucion
#
# -------

n = length(B);
det1 = A(n,n);
X = zeros(n,1);
X(n) = B(n)/A(n,n);
for r = n-1:-1:1,
  det1 = det1*A(r,r);
  if det1 == 0, break, end
  X(r) = (B(r) - A(r,r+1:n)*X(r+1:n))/A(r,r);
end
end
```

4.1.2. Eliminación de Gauss Jordan

El método de Gauss Jordan es una variación de la eliminación de Gauss. La principal diferencia consiste en que cuando una incógnita se elimina en el método de Gauss-Jordan, esta se elimina de todas las otras ecuaciones, no solo de las subsecuentes. Además todos los renglones se normalizan al dividirlos entre su elemento pivote. De esta forma, el paso de eliminación genera una matriz identidad en vez de una triangular. En consecuencia no es necesario usar la sustitución hacia atrás para obtener la solución.

```
function  X = gaussj(A,B)

#  Forma de llamar la funcion
#    X = gauss(A,B)
#  Entradas
#    A   Matriz de Coeficientes
#    B   Vector del lado derecho
#  Devuelve
```

```
#   X   Vector Solucion
#
# -----------------

[n n] = size(A);
A = [A';B']';
X = zeros(n,1);
for p = 1:n,
  for k = [1:p-1,p+1:n],
    if A(p,p)==0, break, end
    mult = A(k,p)/A(p,p);
    A(k,:) = A(k,:) - mult*A(p,:);
  end
end
X = A(:,n+1)./diag(A);
end
```

4.2. Descomposición LU e Inversión de Matrices

4.2.1. Descomposición LU

Esta forma de calcular la solución de los sistemas de ecuaciones por medio de matrices, es llamativa con respecto a la eliminación de Gauss debido a que sus operaciones se realizan sobre los coeficientes de la matriz A, lo cual disminuye el tiempo de ejecución al encontrar la solución al sistema de ecuaciones.

Lo que hace el método es convertir la matriz A = LU, donde L es una matriz triangular inferior y U es una matriz triangular superior, que al ser multiplicadas dan como resultado la matriz A (Hay que tener en cuenta que no todas las matrices pueden ser factorizadas).
Si la matriz A ha sido factorizada entonces el sistema de ecuaciones puede solucionarse en un proceso de dos pasos.

1. y = Ux

2. Ly = b para y.

Los pasos anteriores implican que el número de operaciones a realizar se ve reducido con respecto al número de operaciones usados al aplicar la

eliminación de Gauss.

Pseudocódigo

```
Entradas: n, Aij

for k = 1,2,...,m do
  Especificar un valor para Lkk o Ukk

  Lkk*Ukk = Akk - Sumatoria desde s=1 hasta k-1 de Lks*Usk

  for j = k+1, k+2, ... , n do
    Ukj = ( Akj - Sumatoria desde s=1 hasta k-1 de Lks*Usk) / Lkk
  end for

  for i = k+1,k+2,...,n do
    Ljk = ( Aik - Sumatoria desde s=1 hasta k-1 de Lis*Usk) / Ukk
  end for
end

print Lij, Uij

stop
```

Código

```
function [L, U] = LU_F(A)

%  Factorizacion LU de una matriz A

% usando eliminacion de Gauss sin intercambio de filas.

%  Input:

%  A n-by-n matrix

%  Output:

%  L triangular inferior con 1's en la diagonal
```

```
%  U triangular superior

[n, m] = size(A);

L = eye(n);    % inicializa las matrices

U = A;

for j = 1:n

   for i = j+1:n

      L(i,j) = U(i,j) / U(j,j);

      U(i,:) = U(i,:) - L(i,j)*U(j,:);

   end

end
```

4.2.2. Versión de la eliminación de Gauss usando la descomposición LU

Aunque a primera vista podría parecer que la eliminación de Gauss no está relacionada con la eliminación LU, aquella puede usarse para descomponer [A] en [L] y [U], lo cual se observa fácilmente para [U], que es el resultado directo de la eliminación hacia adelante. Recuerde que en el paso correspondiente a esta eliminación se pretende reducir la matriz de coeficientes [A] en un matriz triangular superior. Para el ejemplo ver el código presentado para la Factorización LU, donde la matriz U, nos brinda la matriz triangular superior necesaria para realizar la sustitución hacia atrás.

4.2.3. Descomposición de Crout

Observamos que en la descomposición LU con la eliminación de Gauss, la matriz [L] tiene números 1 en la diagonal. Formalmente, a esto se le denomina descomposición o factorización de Doolittle. Un método alternativo usa una matriz [U] con números 1 en la diagonal. Esto se conoce como la descomposición de Crout. Aunque hay algunas diferencias entre estos mé-

todos, su funcionamiento es comparable. El método de descomposición de Crout genera [U] y [L] barriendo las columnas y los renglones de la matriz.
Pseudo Código

```
Haga ( j = 2, n)
a(1, j) = a (1, j) / a (1, 1)
fin

Haga (j = 2, n -1)
Haga (i = j, n)
Sum = 0
Haga(K = 1, j -1)
Sum = Sum + a (i, k) * a (k, j)
Fin
a(i, j) = a (i, j) - Sum
fin

Haga ( k = j + 1, n)
Sum = 0
Haga(i = 1, j -1)
Sum = Sum + a (i, j) * a (i, k)
Fin
a(i, k)  = (a (i, k ) - Sum) / a (j, j)
fin
fin
Sum = 0
Haga (k = 1, n -1)
Sum = Sum + a(n, k) * a (k,n)
Fin
a(n, n) = a(n, n) - Sum
```

4.3. Matrices especiales y el método de Gauss-Seidel

4.3.1. Descomposición de Cholesky

Uno de los métodos mas populares es la descomposición de Cholesky. Este algorítmo se basa en el hecho de que una matríz simétrica se descompone así:

$$[A] = [L][L]^T$$

Es decir, los factores triangulares resultantes son la transpuesta uno de otro. Los términos de la ecuación anterior se desarrollan al multiplicar e igualar entre sí ambos lados. El resultado se expreza en forma simple mediante relaciones de recurrencia. Para el renglón K-ésimo,

$$l_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} l_{ij} l_{kj}}{l_{ii}} \, para \, i = 1, 2, ..., k-1$$

y

$$l_{kk} = \sqrt{a_{kk} - \sum_{j=1}^{k-1} l_{kj}^2}$$

Para realizar la descomposición de Cholesky tanto MatLab como Octave incluyen la función chol(A), donde A es la matriz de coeficientes.

4.3.2. Gauss-Seidel

Los métodos iterativos rara vez se usan para resolver sistemas lineales de pequeña dimensión, ya que el tiempo necesario para conseguir una exactitud satisfactoria rebasa el que requieren los métodos directos. Sin embargo, en el caso de sistemas grandes con un alto porcentaje de elementos cero, son eficientes tanto en almacenamiento de computadora como en el tiempo de cómputo. Este tipo de sistemas se presentan constantemente en los análisis de tiempo de cómputo. Este tipo de sistemas se presentan constantemente en los análisis de circuitos y en la solución numérica de los problemas con valor en la frontera y de ecuaciones diferenciales parciales.

Un método iterativo con el cual se resuelve el sistema lineal Ax=b comienza con una aproximación inicial x(0) a la solución x y genera una sucesión de vectores x(k) desde k=0 ->inf que converge a x. Los métodos iterativos traen consigo un proceso que convierte el sistema Ax=b en otro equivalente de la forma x = Tx+c para alguna matriz fija T y un vector c.

Luego de seleccionar el vector inicial x(0) la sucesión de los vectores de la solución aproximada se genera calculando x(k) = Tx(k-1) + c, para cada k = 1,2,3... Este resultado debería recordarnos la iteración de punto fijo.

Pseudocódigo

```
Entradas: n, Aij, bi, Xi, M
```

```
for k = 1,2,...,n do
  for i = 1,2,...,n do
    ui = ( bi - Sumatoria de j hasta n con j!=i de Aij*Xj)
  end for

  for i = 1,2,...,n do
    xi = Ui
  end for
end for

print k, xj

stop
```

Código

```
function [P,dP,Z] = gseid(A,B,P,delta,max1)

%GSEID   Iteracion para resolver sistemas lineales Gauss-Seidel
% Llamado de la funcion
%   [X,dX] = gseid(A,B,P,delta,max1)
%   [X,dX,Z] = gseid(A,B,P,delta,max1)
% Entradas
%   A       Matriz de coeficientes
%   B       vector de soluciones
%   P       vector de inicio
%   delta   tolerancia
%   max1    maximo numero de iteraciones
% Devuelve
%   X       solution vector
%   dX      error estimate vector
%   Z       History matrix of the iterations
%


Z = P';
n = length(B);
Pold = P;
for k=1:max1,
```

```
  for r = 1:n,
    Sum1 = B(r) - A(r,[1:r-1,r+1:n])*P([1:r-1,r+1:n]);
    P(r) = Sum1/A(r,r);
  end
  dP = abs(Pold-P);
  err = norm(dP);
  relerr = err/(norm(P)+eps);
  Pold = P;
  Z = [Z;P'];
  if (err<delta)|(relerr<delta), break, end
end
end
```

4.3.3. Metodo de Jacobi

En la iteración de Jacobi, se escoge una matriz Q que es diagonal y cuyos elementos diagonales son los mismos que los de la matriz A. La matriz Q toma la forma:

$$Q = \begin{pmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{pmatrix}$$

y la ecuación general se puede escribir como
Qx(k) = (Q-A)x(k-1) + b (1)

Si denominamos R a la matriz A-Q:

$$R = \begin{pmatrix} 0 & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & 0 & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & 0 & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & 0 \end{pmatrix}$$

la ecuación (1) se puede reescribir como:
Qx(k) = -Rx(k-1) + b

El producto de la matriz Q por el vector columna x(k) será un vector columna. De modo análogo, el producto de la matriz R por el vector columna x(k-1) será también un vector columna. La expresión anterior, que es una ecuación vectorial, se puede expresar por necuaciones escalares (una para cada componente del vector). De este modo, podemos escribir, para

un elemento i cualquiera y teniendo en cuenta que se trata de un producto matriz-vector:

$$\sum_{j=1}^{n} q_{ij} x_j^{(k)} = -\sum_{j=1}^{n} r_{ij} x_j^{(k-1)} + b_i$$

Si tenemos en cuenta que en la matriz Q todos los elementos fuera de la diagonal son cero, en el primer miembro el único término no nulo del sumatorio es el que contiene el elemento diagonal qii, que es precisamente aii. Más aún, los elementos de la diagonal de Rson cero, por lo que podemos eliminar el término i=j en el sumatorio del segundo miembro. De acuerdo con lo dicho, la expresión anterior se puede reescribir como:

$$a_{ii} x_i^{(k)} = -\sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k-1)} + b_i$$

de donde despejando xi(k) obtenemos:

$$x_i^{(k)} = \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k-1)} \right) / a_{ii}$$

que es la expresión que nos proporciona las nuevas componentes del vector x(k) en función de vector anterior x(k-1) en la iteración de Jacobi.

El método de Jacobi se basa en escribir el sistema de ecuaciones en la forma:

$$\left\{ \; x_1 \;\; = \;\; \left(b_1 - a_{2.....,2n} x_2 - \cdots - \right) / a_{nn} \right.$$

(2)

Partimos de una aproximación inicial para las soluciones al sistema de ecuaciones y sustituimos estos valores en la ecuación (2). De esta forma, se genera una nueva aproximación a la solución del sistema, que en determinadas condiciones, es mejor que la aproximación inicial. Esta nueva aproximación se puede sustituir de nuevo en la parte derecha de la ecuación (2) y así sucesivamente hasta obtener la convergencia.[Vil]

Pseudocódigo

```
Entradas n, Aij, Bi, Xi, M

for k = 1,2,...,M do
```

```
  for i = 1,2,...,n do
    ui = (bi - Sumatoria de j hasta n con j!=i de Aij*Xj) / Aii
  end for

  for i = 1,2,...,n do
    Xi = Ui
  end for
end for

print k, Xj

stop
```

Código

```
function [P,dP,Z] = jacobi(A,B,P,delta,max1)

%JACOBI   Iteracion de Jacobi
% Llamada
%   [X,dX] = jacobi(A,B,P,delta,max1)
%   [X,dX,Z] = jacobi(A,B,P,delta,max1)
% Entradas
%   A       Matriz de coeficientes
%   B       Vector de variables dependientes
%   P       Vector de inicio
%   delta   tolerancia
%   max1    Maximo numero de iteraciones
% Return
%   X       vector solucion
%   dX      error estimado
%   Z       Matriz de iteraciones
%

Z = P';
n = length(B);
Pnew = P;
for k=1:max1,
  for r = 1:n,
    Sum1 = B(r) - A(r,[1:r-1,r+1:n])*P([1:r-1,r+1:n]);
    Pnew(r) = Sum1/A(r,r);
```

```
  end
  dP = abs(Pnew-P);
  err = norm(dP);
  relerr = err/(norm(Pnew)+eps);
  P = Pnew;
  Z = [Z;P'];
  if (err<delta)|(relerr<delta), break, end
end
end
```

4.3.4. Mejoramiento de la convergencia usando relajación

La relajación representa una ligera modificación al método de Gauss-Seidel y esta permite mejorar la convergencia. Después de que se calcula cada nuevo valor de x, ese valor se modifica mediante un promedio ponderado de los resultados de la iteración actual y la anterior:

$$x_i^{nuevo} = \lambda x_i^{nuevo} + (1-\lambda)x_i^{anterior}$$

donde λ es un factor que tiene un valor entre 0 y 2. Si $\lambda = 1, (1-\lambda) = 0$ y el resultado no se modifica. Para valores λ de 1 a 2, se le da una ponderación extra al valor actual. La elección de un valor adecuado de λ es especificado por el problema y se determina de forma empírica. Para la solución de un solo sistema de ecuaciones, con frecuencia es innecesaria. El algorítmo de SOR relajación:

```
function [x, error, iter, flag]  = sor(A, x, b, w, max_it, tol)

%
% sor.m resuelve el sistema linear Ax=b Sor Relajacion, por el metodo de Gauss Seidel
%cuando omega=1
%
% Entradas   A        Matriz
%         x        Vector inicial
%         b        Vector de solucion
%         w        Escalar real de relajacion
%         max_it   Numero maximo de iteraciones
%         tol      Tolerancia
%
% Salida:  x        Vector de solucion
%         error    Error normal
```

```
%         iter     Numero de iteraciones realizadas
%         flag     INT: 0 = Solucion encontrada con tolerancia dada
%                         1 = No converge con el numero maximo de iteraciones

  flag = 0;
  iter = 0;

  bnrm2 = norm( b );
  if  ( bnrm2 == 0.0 ), bnrm2 = 1.0; end

  r = b - A*x;
  error = norm( r ) / bnrm2;
  if ( error < tol ) return, end

     b = w * b
     M =  w * tril( A, -1 ) + diag(diag( A ))
     N = -w * triu( A,  1 ) + ( 1.0 - w ) * diag(diag( A ))

  for iter = 1:max_it

     x_1 = x;
     x   = M \ ( N*x + b );
     error = norm( x - x_1 ) / norm( x );     % calcula el error
     if ( error <= tol ), break, end          % Verifica la convergencia

  end
  b = b / w;

  if ( error > tol ) flag = 1; end;           % no converge

% END sor.m
end
```

4.3.5. Funciones especiales de Octave para manejo de Matrices

cond(A) Numero de condición de una matríz
norm(A) Norma vectorial o matricial
rcond(A) Estimador de condición recíproca
det(A) Determinante

trace(A) Suma de los elementos de la diagonal
orth(A) Ortogonalización
rref(A) Forma escalonada reducida por renglones
chol(A) Factorización de cholesky
lu(A) Factores para eliminación de gauss
inv(A) Matríz Inversa
qr(A) Descomposición ortogonal-triangular
qrdelete(A) Suprimer una columna de la factorización QR
qrinsert(A) Inserta una columna de la factorización QR
pinv(A) Pseudo inversa

Capítulo 5

Interpolación

Cuando desee estimar los valores intermedios entre datos definidos por dos puntos, el método mas común es la interpolación polinomial. Recordemos que la fórmula general para un polinomio de n-ésimo grado es:

$$f(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$$

Dados n+1 puntos, hay uno y solo un polinomio de grado n que pasa a través de todos los puntos.
La interpolación polinomial consiste el polinomio único de n-ésimo grado que se ajuste a n+1 puntos.

5.1. Interpolación polinomial de Newton de Diferencias Divididas

5.1.1. Interpolación lineal

Es la forma mas simple de interpolación y consiste en unir dos puntos con una línea recta. A continuación presentamos la fórmula de interpolación lineal. La notación $f_1(x)$ denota que este es un polinomio de interpolación de primer grado:

$$f_1x = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

5.1.2. Interpolación cuadrática

La interpolación cuadrática se usa cuando se tienen tres puntos como datos, estos pueden ajustarse a un polinomio de segundo grado

$$f_2(x) = b_0 + b_1(x - x_0) + b_2(x - x_1)$$

donde los valores de $b_0, b_1 y b_2$ son los siguientes:

$$b_0 = f(x_0)$$

$$b_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

$$b_2 = \frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0}$$

Luego de que podemos observar una recurrencia en el método de Newton de Diferencias Divididas nos es mas fácil entender y analizar el código a continuación. Para ejecutar este código debemos ingresar dos vectores como parámetros de la función. Como ejemplo resolverémos el ejemplo 18.2 del libro de Chapra:

```
octave:1> Xi=[1,4,6]\\
octave:2> fxi = [0,1.386294,1.175917]
octave:3> divdiff(Xi,fxi)

ans =

   0.00000   0.46210  -0.05187


function nf = divdiff ( xi, fi )
%
%     Argumentos:
%              xi      Vector que contiene los valores de los xi
%              fi      Vector que contiene los valores de los f(xi)
%%
%     Salida:
%              nf      Devuelve un vector que contiene la respuesta utilizando
%       Diferencias dividias
```

```
%
%     NOTA:
%              Recuerde que la magnitud del vector xi debe ser igual
%        a la manitud de el vector de los f(xi)
%

n = length ( xi );
m = length ( fi );

if ( n ~= m )
   disp ( 'El número de los xi debe ser igual al numero de los f(xi)' )
   return
end

nf = fi;
for j = 2:n
    for i = n:-1:j
    nf(i) = ( nf(i) - nf(i-1) ) / ( xi(i) - xi(i-j+1) );
end
end
```

5.2. Polinomios de interpolación de Lagrange

El polinomio de interpolación de Lagrange es simplemente una reformulación del polinomio de Newton que evita el cálculo de las diferencias divididas, y se representa de manera concisa como:

$$f_n(x) = \sum_{i=0}^{n} L_i(x) f(x_i)$$

donde

$$L_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

Donde obtenemos que

$$f_1(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1)$$

Y para un polinomio de segundo grado obtenemos que:

$$f_2(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}f(x_0)+\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}f(x_1)+\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}f(x_2)$$

Al igual que el método de diferencias divididas de Newton, la forma de ejecutar el código de Lagrange es similar, donde la función recibe dos parámetros como entrada: el vector de los xi y el vector de los fxi.Veamos el siguiente ejemplo:

Ejemplo

Con un polinomio de interpolación de Lagrange de primero y segundo grado evalúe Ln2 basándose en los siguientes datos: $x_0 = 1$ $f(x_0) = 0$
$x_1 = 4$ $f(x_0) = 1{,}386294$
$x_2 = 6$ $f(x_0) = 1{,}791760$
Para poner a correr nuestro programa simplemente ejecutamos

```
octave:1> Xi=[1,4,6]\\
octave:2> fxi = [0,1.386294,1.175917]
octave:3> lagrange(Xi,fxi)

ans =

   0.00000   0.46210  -0.05187
```

Y obtenemos la misma respuesta que obtuvimos con el método de diferencias dividas de Newton.

Pseudocódigo

```
Entradas: x, X[], y[], Lit

r=0
num=1
den=1
 for i=0 while i<Lit do //para el total de polinomios
  for j=0 while j<Lit //para cada polinomio
      if i != j
        num = num*(x - X[j])
        den = den*(X[i] - X[j])
      end if
  end for
```

```
  num=y[i]
  r = r + num/den
  num=den=1
 end for
 print "El resultado es r"
```

Código

```
function lp = lagrange ( xi, fi )

%
%     Argumentos:
%              xi      Vector que contiene los valores de los xi
%              fi      Vector que contiene los valores de los f(xi)
%%
%     Salida:
%              nf      Devuelve un vector que contiene la respuesta utilizando
%       el metodo de lagrange
%
%     NOTA:
%              Recuerde que la magnitud del vector xi debe ser igual
%        a la manitud de el vector de los f(xi)
%

n = length ( xi ); %Evaluamos la magnitud de los vectores ingresados
m = length ( fi );

if ( n ~= m )
   disp ( 'El numero de los xi debe ser igual al numero de los f(xi)' )
   return
end

temp = [0];
for i = 1:n
    temp = temp + lagrange-poly ( xi, i ) * fi(i);
end

if ( nargout == 0 )
   disp(temp)
else
```

```
   lp = temp;
end
```

Si observamos bien en el código anterior podemos notar que se hace una llamada a una función lagrange-poly (xi, i) esta función es la que se encarga de construir el polinomio y se realizó en una función aparte para dar mejor claridad al código. Con esto podemos notar que Octave permite hacer llamadas a funciones externas bien sea que las hayamos construido o funciones predeterminadas de Octave. Por ende para poder ejecutar el algoritmo de Lagrange primero debemos cargar en octave la función lagrange-poly y luego cargar la función lagrange. **Código de la función que genera el polinomio de Lagrange:**

```
function lp = lagrange-poly ( xi, i )

%
%     Argumentos:
%              xi      vector que contiene los puntos de interpolación
%              i       indice asociado al punto de interpolación.
%


n = length ( xi );
temp = [1];
denom = 1;

if ( ( i < 1 ) | ( i > n ) )
   disp ( 'lagrange_poly error: index i out of bounds' )
   return
elseif ( i == 1 )
   for j = 2:n
       temp = conv ( temp, [1 -xi(j)] );
   denom = denom * ( xi(1) - xi(j) );
   end
elseif ( i == n )
   for j = 1:n-1
       temp = conv ( temp, [1 -xi(j)] );
   denom = denom * ( xi(n) - xi(j) );
```

```
   end
else
   for j = 1:i-1
       temp = conv ( temp, [1 -xi(j)] );
   denom = denom * ( xi(i) - xi(j) );
   end
   for j = i+1:n
       temp = conv ( temp, [1 -xi(j)] );
   denom = denom * ( xi(i) - xi(j) );
   end
end

if ( nargout == 0 )
   disp(temp/denom)
else
   lp = temp/denom;
end
```

5.3. Interpolación de Neville

La eliminación de Neville es un método para resolver sistemas de ecuaciones lineales que aparece de forma natural cuando la estrategia de interpolación de Neville es usada. Este proceso es una alternativa a la eliminación Gaussiana y se ha demostrado que tiene un mayor rendimiento cuando se trabaja con matrices totalmente positivas, matrices regulares de signo u otros tipos de matrices relacionadas.

Pseudo Código

```
Entradas: F, (x0,y0), (x1,y1),...,(xn,yn)

for i = 0,1,2,...,n
 Q[i,0] = yi
end for

for i = 1,2,...n
 for j = 1,2,...,i

   Q[i,j](x) = ((x-x[i-j])Q[i,j-1](x)-(x-x[i])Q[i-1,j-1](x))/ x[i]-x[j-i]
```

```
 end for
end for

print Q[i,j]
```

Capítulo 6

Diferenciación e Integración Numérica

6.1. Integración de Newton-Cotes

Las fórmulas de Newton-Cotes son las formulas de integración numérica mas comunes. Se basan en la estrategia de reemplazar una función complicada o datos tabulados por un polinomio de aproximación que es fácil de integrar:

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

donde

$$f_n(x) = a_0 + a_1 x + ... + a_{n-1}x^{n-1} + a_n x^n$$

donde n es el grado del polinomio.
Existen formas cerradas y abiertas de las fórmulas de Newton-Cotes. Las formas cerradas son aquellas donde se conocen los datos al inicio y al final de los limites de integración. Las formas abiertas tienen limites de integración que se extienden mas allá del intervalo de los datos. Por lo general, las formas abiertas de Newton-Cotes no se usan para integración definida. Sin embargo, se utilizan para evaluar integrales impropias y para obtener la solución de ecuaciones diferenciales ordinarias.

6.1.1. La regla del trapecio

La regla del trapecio es la primera de las fórmulas cerradas de integración de Newton-Cotes. Corresponde al caso donde el polinomio de la siguiente

ecuación es de primer grado:

$$I = \int_a^b f(x)dx \simeq \int_a^b f_1(x)dx$$

Recordemos que una linea recta se puede representar como:

$$f_1(x) = f(a) + \frac{f(b) - f(a)}{b - a} + (x - a)$$

Y el área bajo esta línea recta es una aproximación de la integral de f(x) entre los límites a y b:

$$I = \int_a^b [f(a) + \frac{f(b) - f(a)}{b - a} + (x - a)]dx$$

El resultado de esta integral es lo que denominamos la regla del trapecio:

$$I = (b - a)\frac{f(a) + f(b)}{2}$$

Pero cuando empleamos la integral bajo un segmento de línea recta para aproximar la integral bajo una curva, obviamente se tiene un error que puede ser importante. Una estimación del error de truncamiento total para una sola aplicación de la regla del trapecio es:

$$E_t = -\frac{1}{12}f''(\xi)(b - a)^3$$

El código en Octave para la regla del trapecio es:

```
function y = trap ( f, a, b, n )

%     inputs:
%             f       string que contiene el polinomio que desea evaluar
%             a       limite inferior de integracion
%             b       Limite superior de integracion
%             n       numero de subintervalos en los que los intervalos
% de integración van a ser subdivididos
%
%
%     NOTA:
%           Recuerde que el modo de ingresar un polinomio o una funcion es:
%  f = inline('0.2+25*x-200*x^(2)+675*x^3-900*x^4+400*x^5')
```

```
%

h = (b-a)/n;
x = linspace ( a, b, n+1 );
for i = 1:n+1
    fx(i) = feval ( f, x(i) );
end
w = [ 0 ones(1,n) ] + [ ones(1,n) 0 ];

if ( nargout == 1 )
   y = (h/2) * sum ( w .* fx );
else
   disp ( (h/2) * sum ( w .* fx ) );
end
```

Ejemplo:
Con la siguiente ecuación integre numericamente desde a=0 hasta b=0.8 y siendo el valor exacto de la integral 1.640533

$$f(x) = 0{,}2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

Solucionando con Octave

```
octave:4> f = inline('0.2+25*x-200*x^(2)+675*x^3-900*x^4+400*x^5')
f =

f(x) = 0.2+25*x-200*x^(2)+675*x^3-900*x^4+400*x^5

octave:8> trap(f,0,0.8,1)
0.17280
octave:9> trap(f,0,0.8,2)
1.0688
octave:10> trap(f,0,0.8,3)
1.3696
octave:11> trap(f,0,0.8,4)
1.4848
octave:12> trap(f,0,0.8,5)
1.5399
octave:13> trap(f,0,0.8,7)
```

```
1.5887
```

Conforme a la respuesta vemos que cada vez que aumentamos el número de subintervalos, la solución se aproxima cada vez mas a la solución obtenida de forma analítica, reduciendo así el error de truncamiento. Dejamos como ejercicio para el lector que además de presentar la solución de la integración numérica, despliegue la información del error de truncamiento, bien sea ingresando la solución obtenida analíticamente como un argumento y sin esta información.

6.1.2. Regla del trapecio de aplicación múltiple

Pudimos observar en la sección anterior que el número de subintervalos se puede ampliar para disminuir el error de truncamiento que se presenta con la regla del trapecio, y la fórmula para representar la regla del trapecio para n subintervalos es[Hof70]:

$$I = (b-a)\frac{f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)}{2n}$$

y donde el error aproximado es:

$$E_a = \frac{(b-a)^3}{12n^2} f''$$

6.1.3. Reglas de Simpson

Además de aplicar la regla trapezoidal con segmentos cada vez más finos, otra manera de obtener una estimación más exacta de una integral, es la de usar polinomios de orden superior para conectar los puntos. Por ejemplo, si hay un punto medio extra entre f(a) y f(b), entonces los tres puntos se pueden conectar con un polinomio de tercer orden.

A las fórmulas resultantes de calcular la integral bajo estos polinomios se les llaman Reglas de Simpson.

6.1.4. Regla de Simpson 1/3

La Regla de Simpson de 1/3 proporciona una aproximación más precisa, ya que consiste en conectar grupos sucesivos de tres puntos sobre la curva mediante parábolas de segundo grado, y sumar las áreas bajo las parábolas

para obtener el área aproximada bajo la curva. La regla de simpson 1/3 se representa con la siguiente ecuación:

$$I = \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)] - \underbrace{\frac{1}{90}f^{(4)}(\xi)h^5}$$

donde la parte encerrada por el corchete corresponde al error de truncamiento.

Pseudo Código

```
Entradas: f, h, x0, xf

ans = 0
ans =  ans + f(x0) + f(xf)
n = 2 * h

for xi = x0+h, adding n, to xf-h do
    ans = ans + 4 * f(xi)
end for

for xi2 = x0+2*h, adding n, to xf-2*h do
    ans = ans + 2 * f(xi2)
end for

ans = (ans * h)/3

print "La integral solucion es ans"

stop
```

Código

```
function y = simp ( f, a, b, n )

%
%     Argumentos:
%             f       string que contiene la función
%             a       Límite inferior de integración
%             b       Lìmite superior de integración
%             n       Numero de subintervalos deseados
%
```

```
%

if ( rem(n,2) ~= 0 )
   disp ( 'n debe ser un numero par' )
   return
end

h = (b-a)/n;
x = linspace ( a, b, n+1 );
for i = 1:n+1
    fx(i) = feval ( f, x(i) );
end
w = [ 1 zeros(1,n-1) 1 ];
w(2:2:n) = 4*ones(1,n/2);
w(3:2:n-1) = 2*ones(1,n/2-1);

if ( nargout == 1 )
   y = (h/3) * sum ( w .* fx );
else
   disp ( (h/3) * sum ( w .* fx ) );
end
```

Ejemplo: Con la ecuación $f(x) = 0{,}2+25x-200x^2+675x^3-900x^3+400x^5$ desde a =0 hasta b=0.8
Solución

```
octave:2> f = inline('0.2+25*x-200*x^(2)+675*x^3-900*x^4+400*x^5')
f =

f(x) = 0.2+25*x-200*x^(2)+675*x^3-900*x^4+400*x^5

octave:7> simp(f,0,0.8,2)
n = 2
n = 2
n = 2
1.3675
octave:8> simp(f,0,0.8,4)
n = 4
n = 4
```

```
n = 4
1.6235
octave:9> simp(f,0,0.8,6)
n = 6
n = 6
n = 6
1.6372
```

Observamos que cada vez que aumentamos el número de subintervalos la respuesta calculada numericamente se aproxima cada vez mas a la respuesta real(1.640533).

6.1.5. Regla de Simpson 3/8

De manera similar a la regla del trapecio y Simpson 1/3, es posible ajustar un polinomio de Lagrange de tercer grado a cuatro puntos e integrarlo:

$$I = \int_a^b f(x)dx \simeq \int_a^b f_3(x)dx$$

para obtener:

$$I \simeq \frac{3h}{8}[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)]$$

donde $h = (b-a)/3$. Esta regla se llama la regla de simpson 3/8 porque esta multiplicada por 3/8. La regla de Simpson 3/8 se expresa tambien mediante la ecuación:

$$I \simeq (b-a)\frac{f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)}{8}$$

Y su error se puede exprezar como:

$$E_t = -\frac{(b-a)^5}{6480}f^{(4)}(\xi)$$

6.1.6. Notas acerca de las formas de integración de Newton-Cotes

En términos generales, las fórmulas de Newton-Cotes no son adecuadas para utilizarse en intervalos de integración grande. Para estos casos se requieren fórmulas de grado superior, y los valores de sus coeficientes son

difíciles de obtener. Además, las fórmulas de Newton-Cotes se basaron en los polinomios interpolantes que emplean nodos con espacios iguales, procedimiento que resulta inexacto en intervalos grandes a causa de la naturaleza oscilatoria de los polinomios de grado superior. Para poder resolver este problema se utiliza la integración numérica compuesta, en la cual se aplican las fórmulas de Newton-Cotes de bajo orden. Estos son los métodos de mayor uso. Ejemplos de estos métodos son: La regla compuesta de Simpson y la regla compuesta del Trapecio. En la integración de Romberg se usa la regla compuesta del Trapecio para obtener aproximaciones preliminares y luego el proceso de extrapolación de Richardson para mejorar las aproximaciones. Las fórmulas de Newton-Cotes se derivaron integrando los polinomios interpolantes. En todas las fórmulas de Newton-Cotes se emplean valores de la función en puntos equidistantes (igual distancia entre un punto y otro). Esta práctica es adecuada cuando las fórmulas se combinan para formar las reglas compuestas sin embargo, esta restricción puede afectar considerablemente la exactitud de la aproximación.[Vil]

6.1.7. Regla Compuesta de Simpson

La regla de simpson se representa mediante la siguiente ecuación:

$$\tau_{simp}^{h}(f)_{a}^{b} = \frac{h}{3}[f(a) + 4\sum_{j=1}^{n} f(x_{2j-1}) + 2\sum_{j=1}^{n-1} f(x_{2j}) + f(b)]$$

[Mat92]

```
function s=simprl(f,a,b,M)

%Argumentos   - String de la función
%         - a Limite inferior de integración
%   - b Límite superior de integración
%         - M Número de subintervalos

h=(b-a)/(2*M);
s1=0;
s2=0;

for k=1:M
   x=a+h*(2*k-1);
   s1=s1+feval(f,x);
end
```

```
for k=1:(M-1)
   x=a+h*2*k;
   s2=s2+feval(f,x);
end

s=h*(feval(f,a)+feval(f,b)+4*s1+2*s2)/3;
```

6.2. Integración de Romberg

La integración de Romberg es una técnica diseñada para obtener integrales numéricas de funciones de manera eficiente. Es muy parecida a las técnicas de integración de Newton-Cotes, en el sentido en que se basan en aplicaciones sucesivas de la regla del trapecio. Sin embargo a través de las manipulaciones matemáticas, se alcanzan mejores resultados con menos trabajo. A continuación tenemos la formulación matemática de Romberg, que posteriormente nos permitirá comprender mejor el algoritmo:

$$I_{j,k} \simeq \frac{4^{k-1} I_{j+1,k-1} - I_{j,k-1}}{4^{k-1} - 1}$$

Código:

```
function [R,quad,err,h]=romber(f,a,b,n,tol)

%Argumentos  - f es la funcion a integrar
%       - a y b limites inferior y superior de integración
%       - n es el máximo numero de filas en la tabla
%       - tol Tolerancia
%
%
%Salida - R Tabla de Romberg
%       - quad valor de la cuadratura
%       - err error estimado
%       - h delta usado


M=1;
h=b-a;
err=1;
```

```
J=0;
R=zeros(4,4);
R(1,1)=h*(feval(f,a)+feval(f,b))/2;

while((err>tol)&(J<n))|(J<4)
   J=J+1;
   h=h/2;
   s=0;
   for p=1:M
      x=a+h*(2*p-1);
      s=s+feval(f,x);
   end
   R(J+1,1)=R(J,1)/2+h*s;
   M=2*M;
   for K=1:J
      R(J+1,K+1)=R(J+1,K)+(R(J+1,K)-R(J,K))/(4^K-1);
   end
   err=abs(R(J,J)-R(J+1,K+1));
end

quad=R(J+1,J+1);
```

Ejemplo:
Use el algoritmo de Romberg para calcular el valor de la integral
$\int_0^{0,8} [0{,}2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5]dx$
Tenga en cuenta que el valor de la integral es de 1.640533 Solución:

```
octave:15> f = inline('0.2+25*x-200*x^(2)+675*x^3-900*x^4+400*x^5')
f =

f(x) = 0.2+25*x-200*x^(2)+675*x^3-900*x^4+400*x^5

octave:16> romber(f,0,0.8,1,0.005)
n = 1
ans =

  0.17280  0.00000  0.00000  0.00000  0.00000
```

```
  1.06880  1.36747  0.00000  0.00000  0.00000
  1.48480  1.62347  1.64053  0.00000  0.00000
  1.60080  1.63947  1.64053  1.64053  0.00000
  1.63055  1.64047  1.64053  1.64053  1.64053

octave:17> romber(f,0,0.8,2,0.005)
n = 2
ans =

  0.17280  0.00000  0.00000  0.00000  0.00000
  1.06880  1.36747  0.00000  0.00000  0.00000
  1.48480  1.62347  1.64053  0.00000  0.00000
  1.60080  1.63947  1.64053  1.64053  0.00000
  1.63055  1.64047  1.64053  1.64053  1.64053

octave:18> romber(f,0,0.8,3,0.005)
n = 3
ans =

  0.17280  0.00000  0.00000  0.00000  0.00000
  1.06880  1.36747  0.00000  0.00000  0.00000
  1.48480  1.62347  1.64053  0.00000  0.00000
  1.60080  1.63947  1.64053  1.64053  0.00000
  1.63055  1.64047  1.64053  1.64053  1.64053
```

Podemos obervar que en la última fila de la tabla generada por el algorítmo de Romberg se encuentra una solución igual al resultado obtenido de manera implícita y el error es casi 0. La solución anterior se dio para 1,2 y 3 subintervalos.

6.3. Diferenciación Numérica

Diferencias Divididas hacia adelante Primera Derivada

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h}$$

$$f'(x_i) = \frac{-f(x_{i+2}) + 4f(x_{i+1}) - 3f(x_i)}{2h}$$

Segunda Derivada

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2}$$

$$f''(x_i) = \frac{-f(x_{i+3}) + 4f(x_{i+2}) - 5f(x_{i+1}) + 2f(x_i)}{h^2}$$

Diferencias Divididas hacia atrás Primera Derivada

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h}$$

$$f'(x_i) = \frac{3f(x_i) - 4f(x_{i-1}) + f(x_{i-2}}{2h}$$

Segunda Derivada

$$f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{h^2}$$

$$f''(x_i) = \frac{2f(x_i) - 5f(x_{i-1}) + 4f(x_{i-2}) - f(x_{1-3})}{h^2}$$

Diferencias Divididas finitas centradas Primera Derivada

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h}$$

$$f'(x_i) = \frac{-f(x_{i+2}) + 8f(x_{i+1}) - 8f(x_{i-1}) + f(x_{i-2})}{12h}$$

Segunda Derivada

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2}$$

$$f''(x_i) = \frac{-f(x_{i+2}) + 16f(x_{i+1}) - 30f(x_i) + 16f(x_{i-1}) - f(x_{i-2})}{12h^2}$$

Capítulo 7

Solución Numérica de Ecuaciones Diferenciales

7.0.1. El método de Heun

Un método para mejorar la estimación de la pendiente emplea la determinación de dos derivadas en el intervalo (una en el punto inicial y otra en el final). Las dos derivadas se promedian después con la finalidad de obtener una mejor estimación de la pendiente en todo el intervalo. Este procedimieto es conocido como el método de Heun.
En el método de Euler, la pendiente al inicio de un intervalo
$y'_i = f(x_i, y_i)$
Se utiliza para extrapolar linealmente a y_{i+1}:
$y^0_{i+1} = y_i + (x_i, y_i)h$
En el método de Heun y^0_{i+1} es una predicción intermedia y no la respuesta final. Por este motivo se le conoce como ecuación predictora o predictor. Y tenemos la ecuación correctora o simplemente corrector:[Cha04]

$$y_{i+1} = y_i + \frac{f(x_i, y_i) + f(x_{i+1}, y^0_{i+1})}{2}h$$

Código:

```
function H=heun(f,a,b,ya,M)

%Argumentos  - f string de la funcion
%        - a y b son los puntos finales de izquierda y derecha
%        - ya condicion inicial y(a)
%        - M Tamaño de paso
%
%
```

```
%Salidas  - H=[T' Y'] T: vector de abcisas Y: vector de ordinales

h=(b-a)/M;
T=zeros(1,M+1);
Y=zeros(1,M+1);
T=a:h:b;
Y(1)=ya;
for j=1:M
   k1=feval(f,T(j),Y(j));
   k2=feval(f,T(j+1),Y(j)+h*k1);
   Y(j+1)=Y(j)+(h/2)*(k1+k2);
end

H=[T' Y'];
```

7.0.2. Método de Runge-Kutta

Los métodos de Runge-Kutta(RK) logran la exactitud del procedimiento de la serie de Taylor sin necesitar el cálculo de derivadas de orden superior. Existen muchas variantes, pero todas tienen la forma generalizada de ecuacion:

$$y_{i+1} = y_i + \phi(x_i, y_i, h)h$$

donde $\phi(x_i, y_i, h)$ se conoce como la función incremento, la cual puede interpretarse como una pendiente representativa en el intervalo. La función incremento se escribe en forma general como

$$\phi = a_1k_1 + a_2k_2 + ... + a_nk_n$$

donde las a son constantes y las k son:
$k_1 = f(x_i, y_i)$

$k_2 = f(x_i + p_1h, y_i + q_{11}k_1 + q_{22}k_2h)$

$k_3 = f(x_i + p_{n-1}h, y_i + q_{21}k_1h + q_{22}k_2h)$

.
.
.

$k_n = f(x_i, +p_{n-1}h, y_i + q_{n-1}k_1h + q_{n-1,2}k_2h + ... + q_{n-1,n-1}k_{n-1}h)$

donde las p y las q son constantes. Observe que las k son relaciones de recurrencia. Es decir k_1 aparece en la ecuación k_2, la cual aparece en la ecuación k_3, etc. Como cada k es una evaluación funcional, esta recurrencia vuelve eficientes a los métodos Runge-Kutta para cálculos en computadora.[Cha04] Luego de haber observado esta relación de recurrencia tal vez nos sea mas fácil entender el siguiente algoritmo y aterrizar los conceptos mediante el siguiente código:

Pseudo Código

```
Runge kutta orden 4

y'= f(t,y), a<=t<=b, y(a)=as

Entradas: a,b,N,as

h = (b - a)/N
t = a
w = as
print t,w

for i = 1,2,...,N do
  k1 = h*f(t,w)
  k2 = h*f(t+h/2, w+k1/2)
  k3 = h*f(t+h/2, w+k2/2)
  k4 = h*f(t+h, w+k3)

  w = w + (k1 + 2k2 + 2k3 + k4)/6
  t = a + i*h
  print t,w
end for

stop

para orden 2

k1 = f(t ,w)
k2 = f(t + 3/4 h, t +3/4 hk1)
w = w +(1/3 k1 + 2/3 k2) h
```

```
para orden 3

k1 = f(t , w)
k2 = f(t + 1/2 h, w + 1/2 hk1)
k3 = f(t + h, w - hk1 +2hk2)
w = w + 1/6(k1 + 4 k2 + k3) h
```

Código Fuente

```
function [wi, ti] = opt_rk2 ( RHS, t0, x0, tf, N )

%  Método Runge-Kutta de segundo orden
%
%     Argumentos:
%              RHS     String que contiene la funcion
%              t0      Valor inicial para variable independiente
%              x0      valor inicial para las variables dependientes
%
%              tf      valor final de variable independiente
%              N       Numero de pasos necesarios para RK


neqn = length ( x0 );
ti = linspace ( t0, tf, N+1 );
wi = [ zeros( neqn, N+1 ) ];
wi(1:neqn, 1) = x0';

h = ( tf - t0 ) / N;

for i = 1:N
    f1 = feval ( RHS, t0, x0 );
xtilde = x0 + (2*h/3) * f1;
    x0 = x0 + (h/4) * ( f1 + 3*feval ( RHS, t0+2*h/3, xtilde ) );
t0 = t0 + h;

    wi(1:neqn,i+1) = x0';
end;
```

Conclusiones

1. Octave resulta ser una alternativa económicamente viable al momento de resolver un problema mediante métodos numéricos y como se pudo observar a lo largo de este manual tiene un tipo de codificación muy similar a los lenguajes tradicionales como C++ y Java, además resulta perfecto para aquellos que ya han trabajado con lenguajes como MatLab anteriormente debido a que su sintaxis es muy similar.

2. Gracias a este manual se pudo observar que Octave ofrece a los desarrolladores un gran número de funciones predefinidas que ayudan y facilitan el trabajo en el momento de desarrollar un software para resolver problemas matemáticos mediante el uso del computador. Demostró ser una gran herramienta con un muy buen performance y simplificó mucho el trabajo del desarrollador.

3. Se puede decir que Octave es la Herramienta Ideal como software educativo, debido a que durante el desarrollo del curso solo se encontró una función que era soportada por MatLab y no soportada por Octave: diff(). Sin embargo esta función promete estar funcionando en una futura versión. En todos los casos encontramos una alta compatibilidad entre MatLab y Octave y durante el desarrollo del curso no se presentó ningún inconveniente, algo que le da otro punto a favor a Octave: Se puede realizar un curso como lo es Analisis Numérico o Métodos Numéricos utilizando Octave, esto fue demostrado en este trabajo.

4. Lo mas seguro es que MatLab aún sea mas robusto que Octave y soporte muchas mas funciones y sea mas fuerte en la parte gráfica, sin embargo para los objetivos del curso, Octave paso la prueba.

5. Se encontraron también algunas debilidades en Octave para MS Windows con algunos equipos.

6. Octave esta diseñado para trabajar en ambiente Linux y sus herramientas mas poderosas han sido diseñadas bajo Linux.

7. Octave esta en perfectas condiciones para convertirse en el lenguaje por defecto del curso de Analisis Numérico, debido a que esta diseñado bajo fundamentos matriciales y soporta una amplia gama de funciones que facilitan mucho la labor del desarrollador al momento de realizar el programa.

Trabajo Futuro

- Realizar un Benchmarking mas a fondo con gráficas y estadísticas entre las herramientas Octave, MatLab y Wolfram Mathematica que demuestre la versatilidad y potencia que puedan tener estas herramientas para el desarrollo de proyectos de investigación dentro de la Universidad. Y realizar a la vez un análisis costo-beneficio con cada una de estas herramientas teniendo como base el Çaso Colombiano".
- Crecer cada vez mas este libro y hacerlo de una manera comunitaria donde futuros estudiantes y docentes realicen su aporte, para poderlo publicar en algún futuro no muy lejano.
- Realizar nuevos aportes de nuevos métodos o problemas que hayan sido resueltos por los estudiantes que utilicen Octave en semestres posteriores y añadirlos como casos de estudio dentro del manual.
- Dar a conocer a los estudiantes las herramientas actuales Open Source como Octave y LaTeX para facilitar sus trabajos de investigación y tesis.
- Realizar un Desarrollo Web que utilice todas las funciones aquí programadas(Se sugiere utilizar PHP,Apache y Linux).
- Publicar este libro.

Bibliografía

[Amu] Henri Amuasi. Octave tutorial. "http://www.aims.ac.za/resources/tutorials/octave/".

[Cas] Alberto F. Hamilton Castro. Introducción al octave. "http://www.cyc.ull.es/asignaturas/octave/ApuntesOctave/ApuntesOctave.html".

[Cha04] Steven C. Chapra. *Métodos Numéricos para ingenieros*. McGraw Hill, Mexico, D.F., 2004.

[Hof70] Joe D. Hoffman. *Numerical Methods for Engineers and Scientists*. Marcel Decker, New York, 1970.

[Mat92] John H. Mathews. *NUMERICAL METHODS: MATLAB Programs*. Prentice Hall, Englewood, NJ, 1992.

[Vil] Wladimiro Díaz Villanueva. Apuntes del curso 1997 a 1998. "http://www.uv.es/ diaz/".

Índice general