Introducción a Matlab® para Resolver Problemas de Ingeniería Aplicando Algoritmos Genéticos

Norberto Hernández Romero,
Joselito Medina Marín,
Juan Carlos Seck Tuoh Mora

Octubre 2012

Directorio:

Contenido

Capítulo 1

Introducción

1.1 Objetivo General

Este material bibliográfico fue elaborado con la finalidad de mostrar la aplicación de Matlab® en el desarrollo de metodologías no convencionales en el campo de la optimización de sistemas discretos.

A pesar de la existencia de herramientas matemáticas en el campo de la optimización, su aplicación en entornos reales es una tarea que requiere un amplio conocimiento formal en el área y experiencia en su aplicación. Además, si se requiere automatizar el proceso de optimización pueden utilizarse software de aplicación general que pudieran no llegar a cubrir las necesidades del problema. Por otro lado, si se desea desarrollar una aplicación a la medida para el problema de optimización, se necesita contar con conocimientos en lenguajes de programación de alto nivel.

En cambio, con la aparición de heurísticas alternas a los métodos clásicos de optimización, es posible encontrar soluciones aceptables a problemas encontrados en la realidad, sin necesidad de desarrollar modelos matemáticos demasiado complejos. Una de estas alternativas de optimización son los Algoritmos Genéticos (AG), cuya implementación no requiere de conocimientos matemáticos profundos y pueden encontrar soluciones aceptables del problema.

1.2 Relevancia de Matlab®

Los algoritmos genéticos pueden ser programados fácilmente en cualquier lenguaje de programación. Sin embargo, uno de los que más se está utilizando actualmente en las áreas de la ingeniería es Matlab®.

Matlab® es un software de cálculo numérico que ofrece un ambiente para realizar operaciones aritméticas y vectoriales en línea de comandos. Además, ofrece un entorno de programación de scripts que facilitan la implementación de códigos para la solución de problemas de gran envergadura.

Matlab® ha sido utilizado en la implementación de algoritmos de optimización, y en particular en la implementación de algoritmos genéticos, debido a su facilidad de programación y su poder de cómputo en el manejo de vectores y matrices. Además, ofrece herramientas para graficar funciones o series de datos en 2 y 3 dimensiones, lo que permite analizar los resultados de la optimización de forma visual. Este software de cálculo numérico cuenta con un conjunto de funciones predefinidas para realizar operaciones matriciales, para que el usuario no invierta tiempo en la programación de tales operaciones.

Además, en este software se incluyen módulos adicionales llamados "toolbox", los cuales contienen entornos para el desarrollo de aplicaciones más específicas. Entre estos "toolbox" se encuentran Simulink – utilizado en la simulación de sistemas dinámicos en diferentes dominios; Symbolic Math – para el desarrollo de operaciones con matemática simbólica; Optimization – ofrece algoritmos de optimización estándar; Statistics – proporciona algoritmos y herramientas para organizar, analizar y modelar conjuntos de datos; entre muchos otros, que facilitan el desarrollo de aplicaciones robustas.

1.3 Relevancia de los Algoritmos Genéticos

Los AG tienen un gran impacto como medios de optimización de procesos industriales, biológicos, químicos, modelos matemáticos, etc., debido que trabajan con funciones lineales, no lineales, discretas, no suaves, probabilísticas y no necesitan de la existencia de derivada de la función objetivo [15], así las razones por las cuales los AG pueden tener éxito en este tipo de planteamiento son las siguientes:

1. Los AG trabajan con un conjunto de parámetros codificados, no con los parámetros de forma directa.

2. Los AG realizan una búsqueda con un conjunto de individuos en todo el espacio factible no solo con un punto.

3. Los AG usan la información de la función objetivo, no con sus derivadas.

4. Los AG usan reglas de transición probabilística, no reglas deterministas.

5. Los AG son procedimientos robustos de optimización que pueden ser implementados en arquitecturas de computo paralelo.

1.4 Optimización en Ingeniería

La optimización puede plantearse como ajustar las entradas o características de un proceso industrial, modelo matemático o experimento para lograr que la salida alcance un punto mínimo o máximo dependiendo de los intereses. Si la salida es ganancia el objetivo es maximizar y si son pérdidas la salida es minimizar.

Si el problema a optimizar se puede plantear mediante un modelo matemático descrito por $f : X \subseteq R^n \rightarrow R$, donde X es el espacio de búsqueda. Así el problema de optimización puede definirse como:

$$\min f(X)_{X \subseteq R^n} \text{ sujeto a } C_k \leq 0 \tag{1.1}$$

Donde $X \subseteq R^n$ es un conjunto en el espacio Euclidiano de dimensión n, $f(X)$ es un conjunto de números reales, k es el número de restricciones y C_k proporciona la forma de la restricción y la restricción es una función que limita el espacio de búsqueda, a esto se le llama problema de optimización restringido. Un punto X_i se le llama punto factible si satisface todas las restricciones y al conjunto de todos los puntos factibles se le llama conjunto factible de búsqueda. En el caso donde las restricciones no se definen entonces es un problema de optimización no restringido, por lo tanto, el dominio de la función es el espacio factible de búsqueda. Los diferentes campos de la optimización puedes clasificarse por las propiedades de función objetivo, la forma de las restricciones, el dominio, naturaleza de la función objetivo, etc. La siguiente tabla muestra una clasificación respecto de las diferentes técnicas matemáticas desarrolladas para la optimización de funciones objetivo [22].

Hay un gran número de métodos para resolver eficazmente los problemas de optimización definidos en la tabla 1 como se muestran en las siguientes referencias [12] [5] [2] [23] [7] [4] [6] [3]. Sin embargo, bajo todos los supuestos que se realizan en la modelación del problema es frecuente para ciertas aplicaciones de la ingeniería que el modelo no represente la realidad y en otras, la función objetivo no puede ser definida por una ecuación algebraica o diferencial, debido a que la función objetivo puede contener regiones en donde la función no es suave o no tiene derivada o la salida de la función objetivo es una colección de datos que se obtienen de instrumentos de medición y por lo tanto, ante la ausencia de una función objetivo no es posible aplicar los técnicas definidas en la Tabla 1 y como consecuencia esta es una de las razones para acudir a las técnicas computacionales evolutivas.

Tabla 1.1: Herramientas analíticas para la optimización de sistemas

Criterio de clasificación	Tipo de problema de optimización	Características
Por la forma de la función objetivo y/o sus restricciones	Lineal	Función objetivo y restricciones lineales.
	No lineal	Función objetivo y restricciones no lineales, con existencia de sus derivadas.
	Convexa	Función objetivo y conjunto de búsqueda representado por un conjunto convexo.
	Estocástica	Función objetivo con ruido.
Por el espacio de búsqueda	Continuo	Variables discretas en la función objetivo y restricciones.
	Discreto	Variables reales en la función objetivo y restricciones.
	Híbrido	Variables reales y discretas en la función objetivo y restricciones.
Por la naturaleza del problema	Dinámica	Función objetivo variante en el tiempo.
	Multiobjetivo	Conjunto de funciones objetivo

1.5 Relevancia del Problema de Empacado en Contenedores

Con el fin de ejemplificar la aplicación de AG implementados en Matlab® a problemas de ingeniería, se tomará el problema de empacado en contenedores. El problema consiste en empacar un conjunto de objetos en la menor cantidad de empaques posibles, donde todos estos tienen una capacidad máxima [8].

Este problema resulta de fácil formulación, sin embargo sus características combinatorias hacen que su solución no sea sencilla, por lo que se requieren de técnicas evolutivas y heurísticas para encontrar una solución satisfactoria para instancias del problema que tomen en cuenta decenas de objetos [11] [8].

Aplicaciones de este problema se pueden encontrar en la asignación de tareas y máquinas en sistemas de manufactura [20] [19], asignación de tareas en microprocesadores [10] [18] [25] y la minimización de espacios y balance en el empaque de mercancías en contenedores [26] [21] [16].

1.6 Organización y Enfoque del Libro

Este libro está enfocado a exponer los conceptos esenciales y aplicaciones de los algoritmos genéticos sin tener que dominar a profundidad un lenguaje de programación.

Es por esto que se ha elegido Matlab® como sistema computacional para implementar los ejemplos del libro ya que permita una implementación rápida y fácil de aprender, lo que nos permite concentrarnos en los detalles y el uso de los algoritmos genéticos así como de su aplicación al problema de empacado en contenedores, más que en las técnicas de programación.

El libro está organizado de la siguiente manera, el Capítulo 1 presenta los conceptos básicos acerca del ambiente de trabajo de Matlab® así como las instrucciones más importantes que se utilizarán para la implementación de algoritmos genéticos. Esta parte contiene una gran variedad de ejemplos que permitirán al lector realizar programas no triviales de manera sencilla. El Capítulo 2 describe los los conceptos básicos de los algoritmos genéticos binarios. Se explican ejemplos para la optimización de funciones y su implementación utilizando las instrucciones vistas en el capítulo anterior. Por último, el Capítulo 3 expone la aplicación de algoritmos genéticos al problema de empacado en contenedores. Se describe paso a paso la implementación tanto de la generación aleatoria de soluciones como de la implementación discreta de los operadores genéticos para la solución del problema descrito anteriormente. Se muestran ejemplos de instancias del problema y su

solución para ejemplificar y analizar el funcionamiento del algoritmo genético. Estos ejemplo fueron tomados de recursos en línea que son comúnmente utilizados para validar algoritmos para el problema de empacado en contenedores. Por último, se presentan comentarios finales indicando otras direcciones en las cuales se pueden aplicar los algoritmos genéticos para problemas discretos.

Capítulo 2

Fundamentos de Programación en Matlab®

Cleve Moler crea MatLab® (MATrix LABoratory) en 1984 que originalmente fue diseñado para simplificar las rutinas numéricas aplicadas al álgebra lineal. Actualmente Matlab® tiene un gran desarrollo como software de cálculo numérico, de tal forma que abarca un amplio rango de aplicaciones en las diferentes áreas de ingeniería tales como aeroespacial, genética, mecánica, eléctrica, electrónica, financiera, informática y aplicaciones de la computación evolutiva.

Así, el propósito de este capítulo es dar los fundamentos más relevantes de la programación en Matlab® que le permitan al lector una visión de los alcances de la programación con Matlab® en la solución de problemas en la ingeniería. Inicialmente se abordan los diferentes operadores en Matlab® (aritméticos, lógicos y relacionales), posteriormente los tipos de datos, funciones matemáticas, escritura de archivos script, comandos de secuencia, manejo de vectores y matrices.

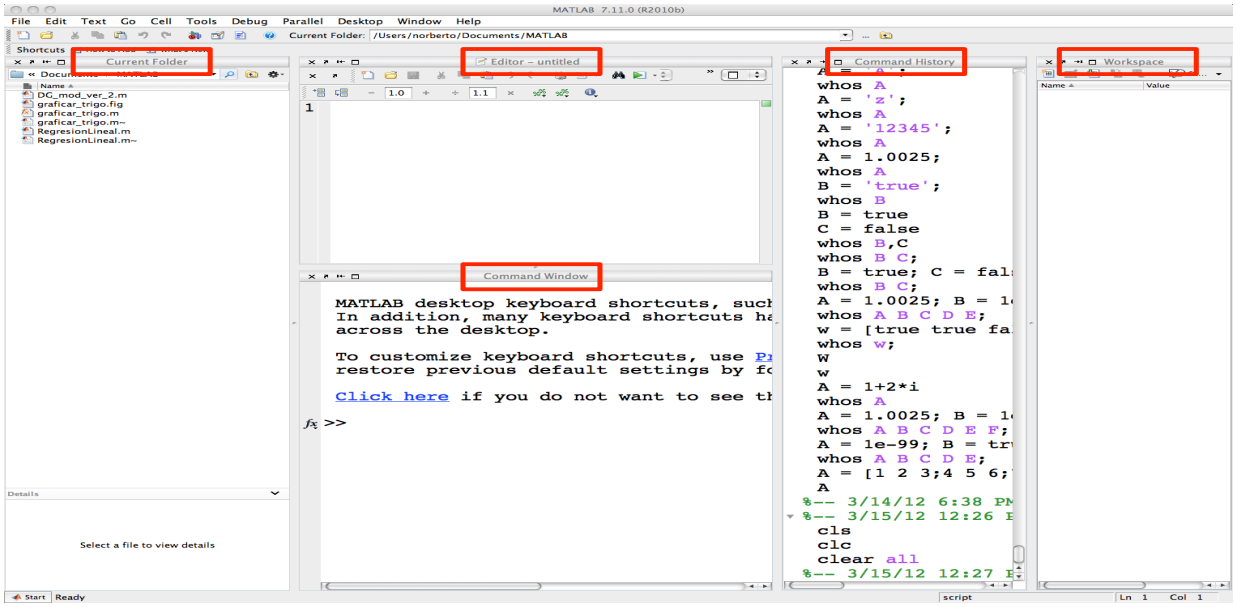

Figura 2.1: Ventana principal de Matlab®

2.1 Introducción a Matlab®

La forma de interactuar con Matlab® es a través de un conjunto de ventanas que se encuentran colocadas en una ventana principal, en la Fig. 2.1 se puede observar la ventana principal de Matlab®, la cual contiene las ventanas **Command Window**, **Workspace**, **Command History**, **Current Directory** y **Editor**. Existen versiones de MatLab® para los sistemas operativos Windows® y MAC®, sin embargo, los comandos de Matlab® vistos en el contenido de este libro son genéricos y se pueden utilizar en cualquier sistema operativo.

La ventana **Command Window** es la interfaz directa de Matlab® con el usuario, esta ventana nos permite usar a Matlab® como una simple calculadora en donde podemos introducir directamente datos y comandos, también es una ventana que se usa para visualizar datos de salida. Cuando se muestra el símbolo >> nos indica que Matlab® está preparado para recibir instrucciones.

El listado 2.1 proporciona un ejemplo sencillo de cómo se puede usar la ventana de **Command Window** como una simple calculadora, básicamente asignamos a la variable A el valor de uno, a la variable B el valor de dos y realizamos operaciones aritméticas de suma y división.

```
>> A = 1
A =
     1
>> B = 2
B =
     2
>> C = (A+B)/B
C =
    1.5000
>>
```

Listado 2.1: Cálculos sencillos en la ventana **Command Window**

Existe un conjunto de instrucciones que interactúan directamente con **Command Window**, la Tabla 2.1 muestra un resumen de algunas de las instrucciones principales.

Tabla 2.1: Comandos directos en la ventana de Command Window

Comando	Descripción
computer	Muestra el tipo de plataforma donde está ejecutando Matlab®
clc	Limpia la ventana Command Window
dir ó ls	Lista los archivos del directorio
cd	Cambiar de directorio
diary	Salva en un archivo de texto la sesión de Matlab®
clear	Limpia variables y funciones que se encuentran en el entorno de Matlab®
whos	Lista las variables que se encuentran declaradas en el entorno Matlab® y sus propiedades
help	Muestra ayuda sobre los comandos de Matlab®
quit ó exit	Cierra la aplicación de Matlab®

Para algunas instrucciones de la Tabla 2.1 es necesario especificar alguna ruta de archivos, nombre de una variable, nombre de un archivo, etc. Por lo tanto, en la ventana de **Command Window** podemos solicitar ayuda de cierto comando con la instrucción *help*, por ejemplo el listado 2.2 muestra las diferentes opciones que tiene el comando *clc*. También es importante mencionar que, cuando nos encontramos en

la ventana **Command Window**, podemos recuperar comandos introducidos anteriormente, con las teclas de flecha arriba ↑ y flecha abajo ↓ podemos ir recorriendo los comandos que fueron introducidos en el pasado.

```
>> help clc
 CLC     Clear command window.
    CLC clears the command window and homes the cursor.

    See also home.

    Reference page in Help browser
       doc clc

>>
```

Listado 2.2: Cálculos sencillos en la ventana **Command Window**

La ventana **Workspace** es el espacio de trabajo donde podemos observar las variables que se encuentran declaradas en ese instante en Matlab® por el usuario, así como el valor que contiene. En el listado 2.3 muestra un ejemplo desde la ventana de comandos de Matlab® en donde se introducen diferentes tipos de variables, A una variable real del tipo doble, B una variable del tipo compleja, C una variable del tipo arreglo y Z una variable del tipo real.

```
>> A = 1.25e99;
>> B = 1+i

B =

   1.0000 + 1.0000i

>> A = 1.25e99;
>> B = 1+i;
>> C = ('HOLA');
>> Z = 1.2345;
>>
```

Listado 2.3: Entrada de datos en **Command Window**

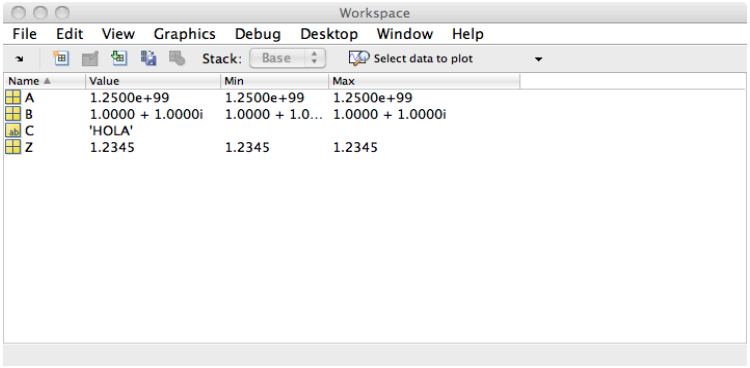

Figura 2.2: Propiedades de las diferentes variables que se encuentran declaradas en Matlab®.

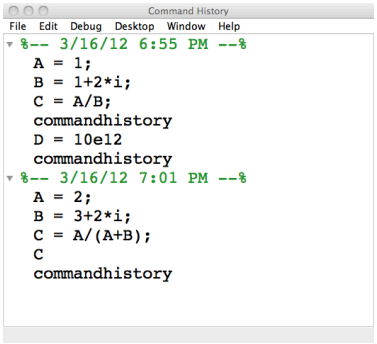

Figura 2.3: Ventana historial de comandos

Posteriormente en la ventana **Workspace** Fig. 2.2 muestra las distintas variables que se encuentran declaradas en Matlab. Primer columna nombre de la variable, segunda columna valor de la variable y tercera y cuarta columna valor mínimo y máximo cuando se trata de un vector numérico. Podemos eliminar determinadas variables del espacio de trabajo con el comando *clear* seguido del nombre de la variable y mediante el comando *clear all* podemos eliminar todas las variables que se encuentran declaradas.

La ventana **Command History** abre el historial de comandos usados en la ventana de **Command Window**, si la ventana se encuentra cerrada esta podrá activarse tecleando >> *commandhistory* en la ventana de comandos.

La Fig. 2.3 muestra un ejemplo de esta ventana, donde se muestra la apertura de sesiones en Matlab® (Fecha y hora) y su correspondiente historial de comandos. Mediante el cursor del mouse se puede simplemente dar doble click a los comandos en la ventana de historial y se volverá a ejecutar el comando seleccionado. Mediante el comando *diary* podemos guardar en un archivo de texto el el historial de comandos. El listado 2.4 muestra un ejemplo del uso del comando *diary*. En este listado se muestra que inicialmente tenemos que activar la opción diary tecleando *diary on*, posteriormente vienen los diferentes comandos introducidos en Matlab® y cuando deseamos guardar nuestro historial en un archivo ejecutamos $diary('prueba1.txt')$ y finalmente apagamos el comando del historial mediante la opción $diary\ off$.

```
>> diary on
>> C = 4*atan(1);
>> b = sqrt(-1) +10;
>> D = C+b;
>> D
D =
   13.1416 + 1.0000i
>> C
C =
    3.1416
>> diary('prueba1.txt');
>> diary off
>>
```

Listado 2.4: Uso del comando *diary*

El símbolo de Matlab® >> se ejecuta en una cierta ruta que está especificada en la ventana **Current Folder**, esta ventana es un explorador de archivos que muestra los detalles sobre la extensión de los archivos y carpetas en donde se está ejecutando actualmente Matlab®. También puede utilizarse esta ventana como una herramienta para realizar operaciones en los archivos y carpetas, tales como mover, comprimir, cambiar

de nombre, crear y eliminar archivos. Apuntando con el mouse a el archivo que se desea mover y con la tecla *ctrl + click* en ambiente MACOS, o botón derecho del mouse en ambiente Windows, aparece el menú de herramientas para manipular los archivos.

Finalmente, en lo que refiere a las ventanas, estas pueden desactivarse o activarse en el menú de la ventana principal, en la opción desktop, para que se tenga un espacio de trabajo organizado y tener activadas solamente las ventanas que se estén utilizando.

2.2 Operadores Aritméticos

Matlab® es un lenguaje que tiene 5 operadores aritméticos para la suma, resta, producto y división, la Tabla 2.2 muestra estos operadores. Para la división podemos usar dos operadores dependiendo del sentido de ejecución. Aunque no es un operador aritmético, incluimos en esta tabla el operador potencia. El listado 2.5 muestra algunas operaciones aritméticas

Tabla 2.2: Operadores aritméticos

Operador	Descripción
$+$	Suma
$-$	Resta
$*$	Producto
$/$	División a la derecha
$\backslash$	División a la izquierda
$\hat{}$	Potencia

El listado 2.5 muestra un ejemplo de la aplicación de los operadores aritméticos, para el caso del operador divisoón se observan los dos casos de ejecución en ambos sentidos.

```
>> 4+3+1-7-2
ans =
     -1
>> 5/2
ans =
    2.5000
>> 2\5
ans =
    2.5000
>> 3^2
ans =
     9
>> 3*4*5
ans =
    60
```

Listado 2.5: Operaciones aritméticas en Matlab®

2.3 Operadores Relacionales y Lógicos

Los operadores relacionales y lógicos se usan para controlar el flujo de ejecución en los programas, para detectar cuando ocurre cierto suceso o para ejecutar n veces un bloque de código, etc. En Matlab® podemos referir seis tipos de operadores relacionales y tres tipos de operadores lógicos que se muestran en la Tabla 2.3

y 2.4. En la segunda tabla, E_1 y E_2 son expresiones que pueden ser VERDADERAS O FALSAS y podemos unir o intersectar las expresiones mediante los operadores & (and) y | (or).

Tabla 2.3: Operadores relacionales

Operador	Comando	Descripción
$<$	lt	Menor que
$<=$	le	Menor o igual que
$>$	gt	Mayor que
$>=$	ge	Mayor o igual que
$==$	eq	Igual que
$=\sim$, $\sim=$	ne	Diferente de

Tabla 2.4: Operadores Lógicos

Operador	Comando	Descripción
&	and	(E_1) y (E_2)
|	or	(E_1) o (E_2)
$\sim$	not	Negación (E_1)

Ejemplo Si E_1 y E_2 son expresiones que pueden tomar valores de $FALSO = 0$ y $VERDADERO = 1$ y les aplicamos los operadores *and* y *or*, el resultado es el siguiente:

Para el operador *and*

Tabla 2.5: Evaluación de dos expresiones lógicas mediante el operador *and*

E_1	E_2	Operación	Resultado
0	0	$E_1 \& E_2$	0
0	1	$E_1 \& E_2$	0
1	0	$E_1 \& E_2$	0
1	1	$E_1 \& E_2$	1

Para el operador *or*

Tabla 2.6: Evaluación de dos expresiones lógicas mediante el operador *or*

E_1	E_2	Operación	Resultado
0	0	$E_1 \mid E_2$	0
0	1	$E_1 \mid E_2$	1
1	0	$E_1 \mid E_2$	1
1	1	$E_1 \mid E_2$	1

2.4 Constantes, Tipos de Datos y Variables

En la Tabla 2.7 se enlistan algunas de las constantes matemáticas más importantes que se encuentran definidas en el entorno de Matlab®. La constante *pi* retorna el valor más cercano, hasta donde se lo permite la resolución de Matlab® del valor de π, las expresiones $4 * atan(1)$ e $imag(log(-1))$ proporcionan el mismo valor. Las constantes i o j se pueden usar de manera indiferente para representar la unidad imaginaria $\sqrt{-1}$.

La constante Inf se usa para representar un valor infinito y es el resultado de operaciones numéricas tales como 1.0/0.0. La constante NaN ó *nan* es resultado de operaciones numéricas que no están permitidas matemáticamente como son 0.0/0.0 o Inf/Inf, cuando la constante NaN se multiplica por cero ó se divide por infinito el resultado sigue siendo NaN. La constante *eps* se determina por 2^{-52}. La constante *realmax* es el número de punto flotante más grande que se puede representar en la computadora en la que se está ejecutando Matlab® y la constante *realmin* es el número positivo de punto flotante más pequeño que se puede representar en Matlab®.

Tabla 2.7: Constantes matemáticas en Matlab®

Sintáxis	Valor	Descripción
pi	3.1415926535897....	Valor de π
i ó j	$\sqrt{-1}$	Unidad imaginaria
Inf ó inf	∞	Infinito
NaN	$0/0$ ó ∞/∞	Valor no definido de forma numérica (**Not a Number**)
eps	2.2204e-16	Diferencia entre 1 y el número en punto flotante inmediantamente superior
realmax	1.7977e+308	El número más grande en punto flotante que puede representar Matlab®
realmin	2.2251e-308	El número más pequeño en punto flotante que puede representar Matlab®

Los principales tipos de datos que maneja Matlab® y los necesarios para los siguientes tópicos que cubre este libro son datos tipo flotante, lógicos y cadena de caracteres o arreglos. El listado 2.6 muestra un ejemplo de estos tipos de datos.

```
>> A = pi;
>> B = [1 2 3 4];
>> C = 'HOLA';
>> D = true;
>> E = false;
>> A
A =
    3.1416
>> B
B =
     1     2     3     4
>> C
C =
HOLA
>> D
D =
     1
>> E
E =
     0
>>
```

Listado 2.6: Principales tipos de datos utilizados en MatLab®

Los formatos de visualización para los números por defecto son el entero, el punto y cuatro cifras significativas a la derecha del punto decimal. Para cuando el número a representar sale de este rango entonces Matlab® usa la notación científica. Sin embargo, existe el comando *format* para establecer formatos predefinidos para visualizar los valores numéricos. La Tabla 2.8 es ejemplo de algunas de las opciones que ofrece el comando y se muestra el valor de π en dichos formatos.

Tabla 2.8: Formatos de visualización de valores numéricos

Comando	Descripción	Ejemplo
format long	16 digitos	3.141592653589793
format short e	5 digitos	3.1416e+00
format short	5 digitos	3.1416
format hex	hexadecimal	400921fb54442d18
format rat	racional aproximado	355/133

Para terminar esta sección vamos a comentar las reglas para designar variables en Matlab®, es importante conocer estas reglas para evitar que Matlab® envíe errores de sintáxis cuando escribimos nuestros programas. A continuación se muestran siete reglas que deben ser consideradas para construir el nombre de nuestras variables.

1. En la designación de variables Matlab® es sensible a las mayúsculas y minúsculas.
2. Todas las variables deberán iniciar con una letra mayúscula o minúscula.
3. La longitud máxima de una variable deberá ser de hasta 63 caracteres.
4. Letras y números están permitidas para designar variables.
5. El signo guión bajo está permitido para designar variables.
6. Los caracteres tales como (. $ # &), etc. no están permitidos en la designación de variables.
7. Si se usan operaciones con números complejos evitar usar i o j como nombre de variable.

2.5 Funciones Matemáticas

Matlab® contiene una gran cantidad de funciones matemáticas, aqui enunciaremos las principales como son las trigonométricas, exponenciales, entre algunas otras.

2.5.1 Funciones Trigonométricas

La Tabla 2.9 muestra una relación de las principales funciones trigonómetricas que contiene Matlab®. En la tabla se muestran los intervalos para el argumento de la función trigonométrica, este intervalo está definido para que la función regrese un valor real, sin embargo, la función puede recibir valores fuera de este rango y entonces la función regresa un número en el espacio de los complejos.

2.5.2 Funciones Exponenciales y Logarítmicas

La Tabla 2.10 muestra una relación de las funciones exponenciales, logarítmicas y de potencia. La tabla también muestra los intervalos válidos que reciben los argumentos de las funciones.

Tabla 2.9: Funciones trigonómetricas

Sintáxis	Valor(Unidad)	Descripción
$y = sin(x)$	$-\infty < x < \infty$ (radianes)	Seno del argumento x.
$y = cos(x)$	$-\infty < x < \infty$ (radianes)	Coseno del argumento x.
$y = tan(x)$	$-\pi/2 < x < \pi/2$ (radianes)	Tangente del argumento x.
$x = asin(y)$	$-1.0 < y < 1.0$	Seno inverso del argumento y.
$x = acos(y)$	$-1.0 < y < 1.0$	Coseno inverso del argumento y.
$x = atan(y)$	$-\infty < y < \infty$	Tangente inversa del argumento y.
$y = sind(x)$	$-\infty < x < \infty$ (grados)	Seno del argumento x.
$y = cosd(x)$	$-\infty < x < \infty$ (grados)	Coseno del argumento x.
$y = tand(x)$	$-90^0 < x < 90^0$ (grados)	Tangente del argumento x.

Tabla 2.10: Funciones Exponenciales y Logarítmicas

Sintáxis	Valor	Descripción
$y = exp(x)$	$-\infty < x < \infty$	Exponencial del argumento x.
$y = log(x)$	$0 < x < \infty$	Logarítmo natural del argumento x.
$y = log2(x)$	$0 < x < \infty$	Logarítmo de base 2 del argumento x.
$y = log10(x)$	$0 < x < \infty$	Logarítmo de base 10 del argumento x.
$y = pow2(x)$	$-\infty < x < \infty$	Potencia de base 2 del argumento x.

2.5.3 Funciones Matemáticas Especiales

Matlab® contiene una cantidad considerable de funciones especiales, en la Tabla 2.11 se enumeran solo las funciones que se hacen necesarias para poder abordar los siguientes capítulos del libro. Dentro de estas funciones se encuentran la raíz cuadrada de un número, redondeo, extración de parte real y compleja, así como la generación de un número aleatorio.

Tabla 2.11: Diversas funciones matemáticas

Sintáxis	Valor	Descripción
$y = sqrt(x)$	$0 < x < \infty$	Raíz de x.
$y = ceil(x)$	$-\infty < x < \infty$	Redondea x hacia infinito.
$y = fix(x)$	$-\infty < x < \infty$	Redondea x a el entero más cercano a cero.
$y = floor(x)$	$-\infty < x < \infty$	Redondea x hacia infinito.
$y = real(x)$	$a + bi$	Parte real de un número complejo.
$y = imag(x)$	$a + bi$	Parte imaginaria de un número complejo.
$y = rem(x, z)$	$-\infty < x < \infty$	Remanente de la división x/z.
	$-\infty < z < \infty$	
$y = round(x)$	$-\infty < x < \infty$	Redondea al entero más próximo.
$y = rand()$	$0 < y < 1$	Genera un número aleatorio uniformemente distribuido.
y = randi(n)	$1 < n < \infty$	Genera un número aleatorio en el intervalo de 1 hasta n.

2.6 Vectores y Matrices

Matlab® es por naturaleza un ambiente soportado en vectores y matrices. Por ejemplo, si se declara una variable A como se muestra en el listado 2.7, A es un escalar que se le asigna el valor de 1.0025 y mediante

el comando *whos* solicitamos a Matlab® las características de la variable. Matlab® guarda la variable como una matriz de 1×1, le reserva un espacio en memoria de 8 bytes y es del tipo doble.

Listado 2.7: Propiedades de la variable A

```
1 >> A = 1.0025;
2 >> whos A
3   Name        Size              Bytes  Class     Attributes
4
5   A           1x1                   8  double
6
7 >>
```

El siguiente ejemplo en Matlab® es para diferentes tipos de datos que son almacenados en memoria como matrices de 1×1. Cuando se introduce una cadena de cuatro caracteres en la variable D el espacio reservado en memoria es para una matriz de tamaño 1×4, para el caso, es una variable del tipo caracter. Con estos ejemplos se puede verificar que cuando declaramos una variable del tipo escalar, Matlab® le da un trato del tipo matricial. Los siguientes puntos de esta sección abordan la introducción a vectores y matrices.

Listado 2.8: Iteración de punto fijo

```
1 >> A = 1e-99; B = true; C = false; D = 'hola'; E = 2+2*i;
2 >> whos A B C D E;
3   Name        Size              Bytes  Class     Attributes
4
5   A           1x1                   8  double
6   B           1x1                   1  logical
7   C           1x1                   1  logical
8   D           1x4                   8  char
9   E           1x1                  16  double    complex
10
11 >>
```

2.6.1 Operaciones con Vectores

Un vector es un arreglo de diferentes elementos agrupados por una fila o columna, puede ser de números reales, complejos o lógicos. El listado 2.9 proporciona la forma de introducir en Matlab® un vector. Para declarar un vector asignamos el nombre de una variable seguida del signo igual y un corchete de apertura, después los elementos que va a contener el arreglo, si es un arreglo fila los elementos se separan por un espacio o por una coma, si el arreglo es del tipo columna se separan por punto y coma, finalmente ponemos el corchete de cierre. Así el arreglo se encuentra declarado en el ambiente de trabajo de Matlab®.

Listado 2.9: A y B son un vector fila y C es un vector columna

```
1 >> A = [1  2  3];
2 >> B = [1 ,2 ,3];
3 >> C = [1;2;3];
4 >> A
5 A =
6       1       2       3
7 >> B
8 B =
9       1       2       3
10 >> C
```

```
11 C =
12        1
13        2
14        3
15 >>
```

Cuando existe la necesidad de introducir vectores que tienen un incremento o decremento constante entre elementos se pueden introducir como se muestra en el listado 2.10. A la variable x le asignamos un arreglo fila de tal forma, que el primer elemento tiene un valor de -1, el último elemento tiene un valor de 1 y el incremento entre cada elemento es de 0.1, de esta forma podemos introducir vectores que contengan una gran cantidad de elementos. Es importante observar, que no es necesario colocar los corchetes, sin embargo, se podrían colocar sin ningún problema y el resultado es el mismo.

Listado 2.10: Declaraciones de vectores con incrementos constantes entre sus elementos

```
1 >> x = -1: 0.1:1;
2 >> x
3 x =
4    Columns 1 through 7
5     -1.0000   -0.9000   -0.8000   -0.7000   -0.6000   -0.5000   -0.4000
6    Columns 8 through 14
7     -0.3000   -0.2000   -0.1000         0    0.1000    0.2000    0.3000
8    Columns 15 through 21
9      0.4000    0.5000    0.6000    0.7000    0.8000    0.9000    1.0000
10 >>
```

Si intentamos el mismo arreglo anterior pero ahora con un decremento, intercambiamos los límites superior e inferior, el listado 2.11 muestra el arreglo declarado con decremento. También se puede observar que el arreglo fue declarado entre corchetes y asignado a la variable x. En este ejemplo, el elemento en la columna 1 tiene un valor de -1 y el elemento de la columna 21 tiene un valor de -1. Para el arreglo asignado a la variable y muestra que si no se declara el incremento, Matlab® por default asigna un incremento o decremento unitario.

Listado 2.11: x vector con decremento, y vector con incremento por default

```
1 >> x = [1:-0.1:-1];
2 >> y = 0:5;
3 >> x
4 x =
5    Columns 1 through 7
6      1.0000    0.9000    0.8000    0.7000    0.6000    0.5000    0.4000
7    Columns 8 through 14
8      0.3000    0.2000    0.1000         0   -0.1000   -0.2000   -0.3000
9    Columns 15 through 21
10    -0.4000   -0.5000   -0.6000   -0.7000   -0.8000   -0.9000   -1.0000
11 >> y
12 y =
13      0     1     2     3     4     5
14 >>
```

Existe otra forma de introducir un arreglo en Matlab® especificando los argumentos, límites superior e inferior y la cantidad de elementos que se desean en el arreglo, esto se puede realizar con la función *linspace*,

el formato de esta función es el siguiente:

$variable = linspace(lim\_inferior, lim\_superior, num\_elementos)$.

El siguiente ejemplo muestra un arreglo 5 elementos entre los valores de $-\pi$ y π.

Listado 2.12: Generación de vectores con el comando *linspace*

```
1 >> x = linspace(-pi,pi,5);
2 >> x
3 x =
4     -3.1416    -1.5708         0    1.5708    3.1416
5 >>
```

Cada elemento de un arreglo contiene un índice y este indica la posición en la que se encuentra dentro del arreglo. Los índices en Matlab® son a partir de uno y se colocan entre paréntesis, el formato de acceso a un elemento de un vector es **variable(índice)** y para el acceso a un segmento del vector es **variable(índice1:índice2)**. El listado 2.13 es un ejemplo de acceso a elementos de un vector. La variable x es un arreglo de 11 elementos, el primer elemento del arreglo es -10, por lo tanto, podemos accesar a dicho elemento mediante el índice uno, así en la variable y se asigna el primero elemento. En la variable z colocamos una porción del arreglo x desde el elemento 2 hasta el elemento 5 y en la variable w se pone la última porción del arreglo x del elemento 6 hasta el elemento 11.

Listado 2.13: Acceso a elementos de un arreglo mediante el índice

```
1  >> x = -10:2:10;
2  >> y = x(1);
3  >> z = x(2:5);
4  >> w = x(6:11);
5  >> x
6  x =
7      -10    -8    -6    -4    -2     0     2     4     6     8    10
8  >> y
9  y =
10    -10
11 >> z
12 z =
13     -8    -6    -4    -2
14 >> w
15 w =
16      0     2     4     6     8    10
17 >>
```

Para sumar dos vectores, deberán ser del mismo tipo de dato, fila o columna y ambos deberán contener el mismo número de elementos. El listado 2.14 muestra las operaciones de adición y sustracción entre vectores. En este ejemplo, a y b son dos vectores del tipo fila con 4 elementos cada uno, c es un vector también tipo fila pero con cinco elementos y d es un vector de 5 elementos del tipo columna. Así, la adición y sustracción en las variables x y y se pueden llevar a cabo de forma correcta. Sin embargo la adición del vector a y c no se puede realizar porqué los vectores son de distinto tamaño. También en la adición entre las variables c y d Matlab® enviá el error debido a que son de la misma longitud los vectores, pero un vector es fila y el otro columna.

Listado 2.14: Operaciones aritméticas con arreglos

```
>> a = [1 2 3 4];
>> b = [4 3 2 1];
>> c = [5 6 7 8 9];
>> d = [1; 2; 3; 4; 5];
>> x = a + b;
>> y = a - b;
>> z = a + c;
Error using  +
Matrix dimensions must agree.
>> w = c + d;
Error using  +
Matrix dimensions must agree.
>> x
x =
     5     5     5     5
>> y
y =
    -3    -1     1     3
>>
```

Otra operación con vectores en Matlab® es que podemos construir arreglos a partir de otros arreglos, a esta operación se le llama **concatenación**. El si-guiente listado muestra la declaración de los arreglos x e y, que posteriormente son concatenados en un solo arreglo z. En el listado 2.15 se muestra que la concatenación se logra colocando entre corchetes los arreglos que se desean unir y las variables deben separase por un espacio. Es requisito en la concatenación que ambos arreglos sean del tipo fila o columna.

Listado 2.15: Concatenación de arreglos del tipo fila

```
>> x = 0:3;
>> y = -2:2;
>> z = [x y];
>> w = [x(1:3) y(2:4)];
>> z
z =
     0     1     2     3    -2    -1     0     1     2
>> w
w =
     0     1     2    -1     0     1
>>
```

El listado 2.16 es un ejemplo de concatenación de vectores del tipo columna en un arreglo tipo fila o columna. En las variables x e y se declaran vectores del tipo columna y son concatenados en la variable z en un vector también del tipo columna, para concatenar vectores columna se ponen las variables entre corchetes y se separan con un punto y coma. Si x e y se desean concatenar en un vector tipo fila, entonces cada vector debera ser convertido a un vector tipo fila mediante la transpuesta de un vector. En Matlab® la transpuesta de un vector se realiza con el operador ', de esta forma, la variable w es una concatenación fila y se obtiene con las transpuestas de los vectores x e y.

Listado 2.16: Concatenación de arreglos del tipo columna

```
>> x = [1;2;3];
>> y = [4;5;6];
>> z = [x;y];
>> w = [x' y'];
>> z
 z =
      1
      2
      3
      4
      5
      6
>> w
w =
      1      2      3      4      5      6
>>
```

El listado 2.17 muestra la forma de realizar la concatenación de dos vectores, el cual uno es es fila y el otro columna. Para el caso si deseamos construir un vector fila, entonces el vector columna le aplicamos la transpuesta como se muestra en la asignación del vector a la variable z. Para el otro caso en la construcción del vector columna, al vector fila se le aplica la transpuesta y la concatenación se realiza con el punto y coma, la variable w es la concatenación del vector x e y en un vector columna.

Listado 2.17: Concatenación de arreglos del tipo fila y columna

```
>> x = [1  2  3];
>> y = [4;5;6];
>> z = [x  y'];
>> w = [x';y];
>> z
 z =
      1      2      3      4      5      6
>> w
w =
      1
      2
      3
      4
      5
      6
>>
```

El listado 2.18 es un ejemplo para llevar a cabo las operaciones de adición y sustracción, la condición que debe cuidarse en este tipo de operaciones es que los vectores a sumarse o sustraerse deben ser de la misma longitud, contener el mismo tipo de dato y ambos deben ser fila o columna. Esta operación aritmética se realiza elemento a elemento, esto es, para el caso de la suma $z(1) = x(1) + y(1)$, $z(2) = x(2) + y(2)$, y así sucesivamente hasta el último elemento.

Listado 2.18: Operaciones artiméticas de adición y sustracción con vectores

```
>> x = [1 2 3 4 5];
>> y = [5 4 3 2 1];
>> z = x+y;
>> w = y-x;
>> z
 z =
      6     6     6     6     6
>> w
 w =
      4     2     0    -2    -4
>>
```

El listado 2.19 es un ejemplo para realizar el producto o división de un vector con un escalar, las variables definidas son: y es el escalar y x es un vector fila, la operación muestra que los elementos del vector fila en la variable z son escalados por un factor de 0.5 y los elementos del vector fila en la variable w son escalados por un factor de 2 en el producto y división respectivamente.

Listado 2.19: Operaciones producto y división de un vector con un escalar

```
>> x = [1 2 3 4 5 6];
>> y = 0.5;
>> z = x*y;
>> w = x/y;
>> z
 z =
    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
>> w
 w =
     2     4     6     8    10    12
>>
```

El listado 2.20 es un ejemplo para mostrar el producto o división de vectores elemento a elemento. Para solicitarle a Matlab® este tipo de operación es usando un punto antes del operador aritmético producto o división $(.*)$ o $(./)$, de esta forma, Matlab® ejecuta un producto o división entre los elementos, para el ejemplo, $x(1)$ con $y(1)$, $x(2)$ con $y(2)$, hasta $x(n)$ con $y(n)$ donde n será el último elemento del arreglo. También para ejecutar esta operación es importante que ambos arreglos sean fila o columna, de la misma longitud y tipo de dato.

Listado 2.20: Operaciones producto y división entre vectores elemento a elemento

```
>> x = [1 2 3 4 5];
>> y = [0.1 0.2 0.3 0.4 0.5];
>> z = x.*y;
>> w = x./y;
>> z
 z =
    0.1000    0.4000    0.9000    1.6000    2.5000
>> w
 w =
    10    10    10    10    10
```

```
11 >>
```

Las funciones matemáticas tales como trigonométricas, exponenciales, logarítmicas, raíz y otras funciones especiales, pueden aplicarse sobre vectores, el listado 2.21 es un ejemplo de la declaración de un vector en la variable x y posteriormente se asigna a la variable y la función sinusoidal del vector x y así para la variable z y w contienen la exponencial y raíz del vector x, respectivamente.

Listado 2.21: Funciones senoidal, exponencial y raíz en vectores

```
1  >> x = linspace(0,pi,5);
2  >> y = sin(x);
3  >> z = exp(x);
4  >> w = sqrt(x);
5  >> x
6  x =
7            0    0.7854    1.5708    2.3562    3.1416
8  >> y
9  y =
10           0    0.7071    1.0000    0.7071    0.0000
11 >> z
12 z =
13    1.0000    2.1933    4.8105   10.5507   23.1407
14 >> w
15 w =
16           0    0.8862    1.2533    1.5350    1.7725
17 >>
```

La función $rand()$ genera un número aleatorio entre 0 y 1 cuando se usa sin argumentos y mediante los argumentos con el formato $rand(fila, columna)$ podemos generar vectores de números aleatorios del tipo fila o columna. En el listado 2.22 a la variable y se le asigna un vector fila de cinco elementos aleatorios y en la variable z se asigna una columna de cinco elementos de valores aleatorios.

Listado 2.22: Vectores fila y columna con valores aleatorios

```
1  >> y = rand(1,5);
2  >> z = rand(5,1);
3  >> y
4  y =
5     0.7482    0.4505    0.0838    0.2290    0.9133
6  >> z
7  z =
8     0.1524
9     0.8258
10    0.5383
11    0.9961
12    0.0782
13 >>
```

2.6.2 Operaciones con Matrices

Las matrices se introducen en Matlab® fila por fila, separando las filas por un punto y coma o un retorno de línea (Enter/Intro), El listado 2.23 proporciona el ejemplo de declarar las matrices A y B de tamaño 3×3 usando ambos formatos. En la línea 1 se declara la variable A y se le asigna una matriz de 3×3 separando las filas por punto y coma. En la línea 2 se declara la variable B, abrimos corchete, ponemos la primer fila de la matriz y se da (Enter), en la tercer línea se introduce la segunda fila de la matriz y se da Enter, en la cuarta línea se pone la tercer fila de la matriz y se cierra corchete. Posteriormente podemos acceder ambas matrices.

Listado 2.23: Declaración de matrices en Matlab®

```
1  >> A = [1  2  3; 4 5  6; 7 8  9];
2  >> B = [9  8  7
3  6  5  4
4  3  2  1];
5  >> A
6  A =
7          1       2       3
8          4       5       6
9          7       8       9
10 >> B
11 B =
12         9       8       7
13         6       5       4
14         3       2       1
```

Para realizar el acceso parcial a los elementos de una matriz es a través de los índices de la matriz, con el formato $variable(i, j)$, donde i es el número de fila y j es el número de columna. Ejemplo, en el listado 2.24 se declara una matriz de 3×3 en la variable A y el acceso al elemento $(3, 2)$, fila 3, columna 2, se realiza como $A(3, 2)$. También podemos tomar un vector fila o columna completo usando el siguiente formato $variable(:, j)$ o $variable(i, :)$, donde los dos puntos (:) indica que se acceda a toda la fila o columna, para el ejemplo asignamos a la variable B la columna central y a la variable C la última fila de la matriz A. De la misma manera, si deseamos recuperar una submatriz de A, podemos indicarlo como $A(i_1 : i_2, j_1 : j_2)$; por ejemplo para obtener solamente los dos primeros renglones con las dos últimas columnas de A se especifica como $A(1 : 2, 2 : 3)$.

La concatenación de matrices se hace de forma similar a la concatenación de vectores. El listado 2.25 es un ejemplo que muestra la definición de dos matrices A y B de dimensión 2×3 y su concatenación en forma de fila en la matriz $C(2 \times 6)$ y en forma de columna en la matriz $D(4 \times 3)$.

Listado 2.24: Acceso a elementos de una matriz

```
1  >> A = [1  2  3; 4 5  6; 7 8  9];
2  >> A
3  A =
4          1       2       3
5          4       5       6
6          7       8       9
7  >> A(3,2)
8  ans =
9          8
10 >> A(1,2)*A(2,3)
11 ans =
```

```
12      12
13  >> B = A(:,2);
14  >> C = A(3,:);
15  >> B
16  B =
17        2
18        5
19        8
20  >> C
21  C =
22        7       8       9
23  >>
```

Listado 2.25: Concatenación de matrices

```
 1  >> A = [1  2  3;7  8  9];
 2  >> B = [4  5  6; 10  11  12];
 3  >> C = [A B];
 4  >> D = [A; B];
 5  >> A
 6  A =
 7        1       2       3
 8        7       8       9
 9  >> B
10  B =
11        4       5       6
12       10      11      12
13  >> C
14  C =
15        1       2       3       4       5       6
16        7       8       9      10      11      12
17  >> D
18  D =
19        1       2       3
20        7       8       9
21        4       5       6
22       10      11      12
23  >>
```

Existen una gran cantidad de operaciones matriciales tales como adición, sustracción, producto, inversa, determinante, transpuesta, traza, etc., que Matlab® tiene desarrolladas. En el listado 2.26 se muestra un ejemplo de adición, sustracción y producto entre matrices. En el ejemplo se hace uso de la función $ones(fila, columna)$ de Matlab® que genera una matriz de 3×3 con sus elementos en uno.

Listado 2.26: Operaciones Aritméticas Matriciales

```
1  >> A = [1  2  3; 4  5  6; 7  8  9];
2  >> B = ones(3,3);
3  >> C = A+B;
4  >> D = B-A;
5  >> E = A*B;
```

```
>> A
A =
     1     2     3
     4     5     6
     7     8     9
>> B
B =
     1     1     1
     1     1     1
     1     1     1
>> C
C =
     2     3     4
     5     6     7
     8     9    10
>> D
D =
     0    -1    -2
    -3    -4    -5
    -6    -7    -8
>> E
E =
     6     6     6
    15    15    15
    24    24    24
>>
```

El operador división no existe matricialmente, pero en su lugar existe el operador de matriz inversa. El listado 2.27 es un ejemplo del comando $inv(x)$ que determina la inversa de una matriz cuadrada x. Se usa la función $randi()$ para generar una matriz de 4×4 con valores aleatorios enteros en el rango entre 1 y 10, posteriormente asignamos en B la inversa de A y por lo tanto en la matriz C obtenemos la matriz identidad.

Listado 2.27: Inversa de una matrix cuadrada de 4×4

```
>> A = randi([1 10],4,4);
>> B = inv(A);
>> C = A*B;
>> A
A =
     7     8     5     6
     2     1     5     6
     4    10     5     9
     7     8     4     8
>> B
B =
    0.0688   -0.0153   -0.2141    0.2008
    0.0937   -0.1319    0.1530   -0.1434
    0.3461    0.1453    0.0344   -0.4073
   -0.3270    0.0727    0.0172    0.2964
>> C
C =
    1.0000    0.0000   -0.0000         0
   -0.0000    1.0000   -0.0000    0.0000
   -0.0000    0.0000    1.0000    0.0000
```

```
21      -0.0000            0     -0.0000      1.0000
22  >>
```

Existen otras funciones para trabajar con matrices, en la Tabla 2.12 se enumeran algunas de ellas. Se deja al alumno que verifique cada una de ellas y realice algunos ejercicios.

Tabla 2.12: Funciones Matriciales

Sintáxis	Descripción
$diag(v)$	Matriz diagonal con el vector v como diagonal principal.
$ones(n)$	Matriz de $n \times n$ con todos los valores iguales a uno.
$zeros(n)$	Matriz de $n \times n$ con todos los valores iguales a cero.
$eye(n)$	Matriz identidad de $n \times n$.
$rand(n)$	Matriz de $n \times n$ con elementos de valor aleatorio entre 0 y 1 con una distribución uniforme
$randn(n)$	Matriz de $n \times n$ con elementos que tienen una distribución normal, (media 0 y varianza 1)
$det(A)$	Determinante de una matriz A cuadrada
$rank(A)$	Número de filas o columnas linealmente independientes
$size(A)$	Número de filas o columnas que tiene la matriz A

2.7 Archivos Script

Un script es una secuencia de comandos que se pueden ejecutar frecuentemente y que se pueden guardar en un archivo con extensión .m, los archivos script en Matlab® puede contener una función o un listado de comandos. Este tipo de archivos se puede crear, editar y guardar con cualquier editor de textos, sin embargo, Matlab® tiene su propio editor de textos adecuado para la ejecución y depuración de archivos script. Cuando se escribe el nombre del archivo en el prompt de Matlab®, su contenido se ejecuta. Para que un archivo .m sea una función, tiene que empezar por la palabra *function* seguida de las variables de salida entre corchetes, el nombre de la función y las variables de entrada. En las siguientes secciones se hara uso de archivos del tipo script.

El siguiente es un ejemplo de un archivo script tipo función para solucionar un sistema de ecuaciones lineales, si el sistema de ecuaciones está definido por la forma $Ax = B$, donde $A_{n\times n}$, $x_{1\times n}$ y $B_{1\times n}$, siendo n el número de incógnitas. Para implantar la solución en una función los argumentos de la función son, la matriz A y B, y la solución es un vector C, que se puede obtener como $C = inv(A) * B$. El siguiente archivo script es un ejemplo de cómo llevar a cabo la solución del sistema de ecuaciones lineales.

```
1  function [ C] = sol_ecuaciones_lineales( A,B )
2  % A es una matriz de n x n
3  % B es un vector de n x 1
4  % C es el vector solucion de n x 1
5  C = inv(A)*B;
6  end
7
```

Es importante mencionar que para archivos script tipo función es necesario que el nombre del archivo y el nombre de la función sea el mismo.

2.7.1 Graficación 2D

Matlab® tiene diferentes herramientas para realizar la graficación en dos dimensiones, en este trabajo se reporta el uso de la función *plot*() que para nuestros objetivos es suficiente. La función *plot*() entre algunos de sus formatos son:

$$\begin{array}{l} plot(x); \\ plot(x,y); \\ plot(x,y1,'tipo\_y\_color\_trazo',x,y2,'tipo\_y\_color',...); \end{array} \tag{2.1}$$

Se puede cambiar el tipo de trazo y color de acuerdo a la Tabla 2.13 donde se usa una letra para cambiar el color del trazo y un símbolo para ajustar el tipo de línea.

Tabla 2.13: Opciones para configurar el color y tipo de trazo en la función *plot*()

Letra	Color	Símbolo	Estilo de línea
y	amarillo	.	punto
m	magenta	*o*	círculo
c	cian	*x*	marca cruz
r	rojo	+	más
g	verde	*	estrella
b	azul	−	línea sólida
w	blanco	:	línea punteada
k	negro	−.	línea punto-guión
		−−	línea discontinua

Existen otras funciones complementarias a la función *plot*(), para agregar un título a la gráfica, texto a la abscisa, texto a la ordenada, ajustar los ejes y/o poner una rejilla auxiliar a la gráfica. La Tabla 2.14 muestra una descripción de las funciones complementarias a la función *plot*().

Tabla 2.14: Funciones complementarias para la función *plot*()

Función	Descripción
$axis([xmin\ xmax\ ymin\ ymax])$;	Pone los valores de los ejes de acuerdo al vector fila
$xlabel('Etiqueta')$;	Coloca una etiqueta a la abscisa de la gráfica
$ylabel('Etiqueta')$;	Determina una etiqueta a la ordenada de la gráfica
$title('Etiqueta')$;	Asigna un título en la parte superior de la gráfica
grid on;	Traza una rejilla auxiliar en la gráfica

El siguiente archivo script muestra la forma de realizar la graficación de 2 funciones trigonométricas, las funciones están definidas por $y_1 = Asin(x)$ y $y_2 = Acos(x)$, para el rango de $x = [-10, 10]$ y la amplitud de las funciones senoidales es de 10. La Figura 2.4 muestra la salida de la función correspondiente a la gráfica de las 2 funciones trigonométricas, la función senoidal con la marca x y trazo rojo, y la función cosenoidal con la marca o y trazo azul.

```
1 % Uso de la función plot en Matlab
2 % para la graficación de funciones
3 % en dos dimensiones.
4
5 x = -10:0.1:10;
6 A = 10;
```

```
y1 = A*sin(x);
y2 = A*cos(x);
plot(x,y1,'rx',x,y2,'bo');
grid on;
title('Gráficas de la funcion seno y coseno');
xlabel('x en radianes');
ylabel('Amplitud de y1 y y2');
```

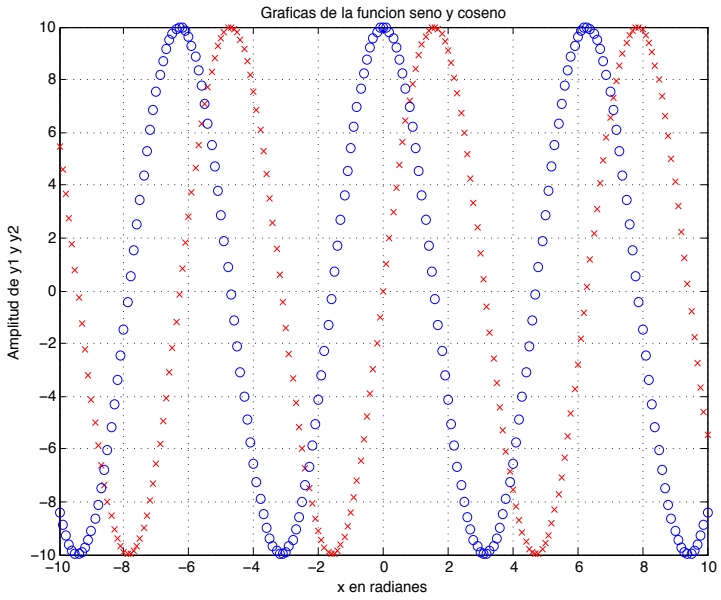

Figura 2.4: Uso de la función plot para la graficación de funciones en dos dimensiones

El siguiente archivo script realiza la graficación de las funciones anteriores con la variante de usar un trazo sólido y también muestra el uso del comando *axis*.

```
% Uso de la funcion plot en Matlab
% para la graficacion de funciones
% en dos dimensiones.

x = -10:0.1:10;
A = 5;
y1 = A*sin(x);
y2 = A*cos(x);
xmin = -12;
xmax = 12;
ymin = -6;
ymax = 6;
plot(x,y1,'r-',x,y2,'b-');
axis([xmin xmax ymin ymax]);
grid on;
title('Funciones seno y coseno');
xlabel('x en radianes');
ylabel('Amplitud de y1 y y2');
```

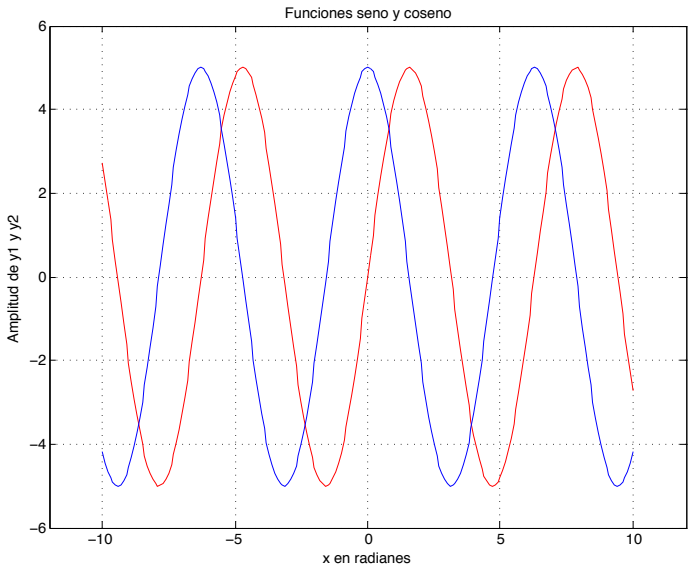

Figura 2.5: Aplicación de funciones complementarias a la función *plot*()

2.7.2 Graficación 3D

La graficación en 3 dimensiones (3D) se puede realizar básicamente mediante el uso de los comandos *meshgrid*() y $surf()$ de Matlab. La función $[X,\ Y] = meshgrid(x, y)$ recibe como argumentos dos vectores tipo fila o columna, $x_(1 \times n)$ e $y_(1 \times m)$. La función regresa dos matrices $X_(m \times n)$ e $Y_(m \times n)$.

$$X = \begin{bmatrix} x_{(1\times n)_1} \\ x_{(1\times n)_2} \\ \vdots \\ x_{(1\times n)_m} \end{bmatrix} \tag{2.2}$$

$$Y = \begin{bmatrix} y'_{(1\times m)_1} & y'_{(1\times m)_2} & \cdots & y'_{(1\times m)_n} \end{bmatrix} \tag{2.3}$$

La matriz X se construye realizando replicas del vector fila $x(1 \times n)$, m veces, teniendo como resultado una matriz $X(m \times n)$, la Ec. 2.2 muestra esta operación. La matriz Y se construye realizando replicas del vector columna y', n veces como se muestra en la Ec. 2.3.

La función $surf(X, Y, Z)$ construye una superficie coloreada en 3D. Las etiquetas de los ejes están determinadas por los rangos de X, Y y Z. Ejemplo:

Graficar la siguiente función Ec. 2.4 en el intervalo de $x = [-2, 2]$ e $y = [-3, 3]$.

$$f(x, y) = -3x - 2y - 1 \tag{2.4}$$

El siguiente programa muestra la declaración de las variables x e y con sus respectivos intervalos, el uso de *meshgrid*() para la construcción de las matrices X e Y y el cálculo de los valores para la variables dependiente $f(X, Y) = Z$. Finalmente el trazo de la superficie con el comando $surf(X, Y, Z)$.

```
1 %Programa que realiza la graficación de una
2 %función en 3D.
3
4 %Se define el intervalo de las
5 %variables independientes.
```

```
6  x = -2:1:2;
7  y = -3:1:3;
8
9  %Se realiza la replica de los vectores.
10 [X,Y] = meshgrid(x,y);
11
12 %Se obtienen los valores de la variable
13 %dependiente.
14 Z = -1 - 3*X -2*Y;
15
16 %Se traza la grafica.
17 surf(X,Y,Z)
```

La Fig. 2.6 muestra el resultado de la graficación de la Ec. 2.4, donde el intervalo de $[-3, 3]$ es el eje y, el intervalo de $[-2, 2]$ es el eje x y la variable de pendiente $f(x, y)$ se encuentra dentro del intervalo $[-15, 15]$.

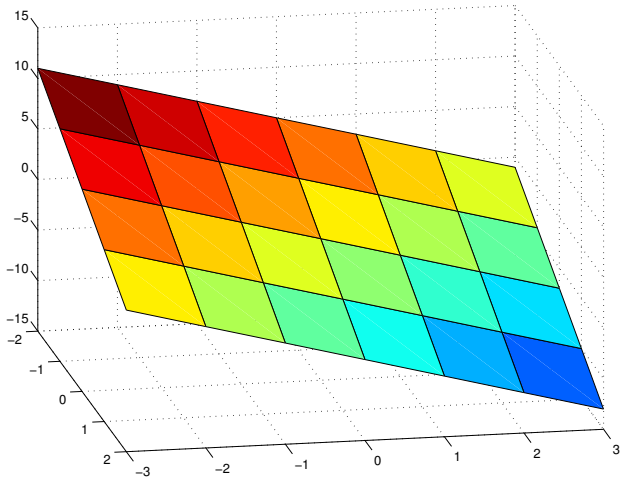

Figura 2.6: Graficación de un plano en 3D usando $surf()$

2.8 Control de Flujo

2.8.1 Estructura $if - elseif - else$

La estructura $if - elseif - else$ ejecuta bloques de código que están sujetos a cumplir una o más expresiones de condición lógica, la estructura es la siguiente:

Listado 2.28: Estructura if

```
1 if (expresión 1)
2
3          bloque de código 1
4
5 elseif (expresión 2)
6
7          bloque de código 2
8
```

```
else
        bloque de código 3
end
```

Ejemplo Realizar un programa en Matlab®, para determinar el tipo y las raíces de un polinomio de segundo grado de la forma $ax^2 + bx + c = 0$, usando una secuencia del tipo $if - elseif - else$.

```
%Programa para determinar las raíces de un polinomio de
%segundo grado (ax^2 + bx +c = 0 ) usando la formula general.
%Datos de entrada (a, b, c)
%Datos de salida  (x1,x2)
clear all;
clc;
%Leemos desde el teclado los coeficientes del polinomio
a = input('Introducir el valor de a ');
b = input('Introducir el valor de b ');
c = input('Introducir el valor de c ');
% Determinamos el tipo de raíz
aux = sqrt(b^2-4*a*c);
if aux > 0
   disp('Las raíces son reales diferentes ');
   x1 = (-b-aux)/(2*a)
   x2 = (-b+aux)/(2*a)
elseif aux == 0
    disp('Las raíces son reales múltiples ');
    x1 = -b/(2*a)
    x2 = -b/(2*a)
else
    disp('Las raíces son complejas conjugadas ');
    x1 = (-b-aux)/(2*a)
    x2 = (-b+aux)/(2*a)
end


```

2.8.2 Estructura *for*

La instrucción *for* incluye una expresión que especifica el valor inicial de un índice, otra expresión que determina cuándo se continúa o no el ciclo, y una tercera expresión que permite que el índice se modifique al final de cada pasada. La forma general de la instrucción for es:

Listado 2.29: Estructura for

```
for expresión 1: expresión 2: expresión 3

        bloque de código

end
```

donde expresión 1 se utiliza para iniciar algún parámetro, que controla la repetición del ciclo, expresión 2 se utiliza para modificar el valor del parámetro inicialmente asignado por expresión 1 y expresión 3 representa una condición que debe ser satisfecha para que se ejecute el ciclo, también llamada condición de paro.

El siguiente programa es un ejemplo de lectura de una matriz de $n \times m$ que se asigna a la variable A, La variable N índica el máximo numero de filas que contiene la matriz, así como M es el valor máximo de

columnas en la matriz, los índices i_1 e i_2 se usan para indicar la posición de la fila o columna respectivamente que se lee en cada pasada de los ciclos for.

```
1  %Programa que lee una matriz desde el teclado
2  % Datos de entrada  n = número de filas,
3  % m = número de columnas y
4  % n x m elementos a(n,m).
5  clear all;
6  clc;
7  %Introducir el número filas y columnas de la matriz
8  N = input('Numero de filas ');
9  M = input('Numero de columnas ');
10  %Introducir el elemento correspondiente a la fila y columna
11 for i1 = 1: N
12     for  i2 = 1:M
13         disp(['A(' int2str(i1) ',' int2str(i2) ')= ']);
14         A(i1,i2) = input('');
15     end
16 end
17  %Mostrar el contenido de la matriz
18 disp('La matriz A = ');
19 disp(A);
```

El siguiente programa es un ejemplo para realizar el producto matricial de matrices cuadradas que son conformes de tamaño $m \times m$. En la línea 8 introducimos el valor m que representa el tamaño de filas y columnas de las matrices. En las líneas 10 y 11 se generan los elementos de las matrices de forma aleatoria, en las líneas 13 y 15 se imprimen las matrices al workspace de Matlab® y en las líneas 19 hasta 25 se encuentran tres for anidados para realizar el producto matricial. Finalmente, en la línea 29 se imprime el valor del producto matricial.

```
1  %Programa para realizar el producto
2  %matricial de matrices cuadradas A(m,m) x B(m,m) = C(m,m)
3  %m = numero de filas de A
4
5
6  % Datos de entrada
7  disp('Introducir el valor del numero de renglones');
8  m = input('m ');
9
10 A = randi(m,m);
11 B = randi(m,m);
12 disp('La matriz A = ');
13 disp(A);
14 disp('La matriz B = ');
15 disp(B);
16
17 C = zeros(m);
18
19 for i1 = 1: m
20     for i2 = 1:m
21         for i3 = 1:m
22             C(i1,i2) = C(i1,i2) + A(i1,i3)*B(i3,i2);
23         end
24     end
25 end
26
27 %Dato de Salida
28 disp('C = A x B ');
29 disp(C);
```

El siguiente ejemplo es una regresión lineal de un conjunto de parejas ordenadas $[x_i, y_i]$ que se ajustan a la ecuación de una recta definida por $y_i = a_1 * x_i + a_0$, donde los coeficientes de la ecuación de la recta se determinan por las siguientes ecuaciones:

$$a_1 = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2} \tag{2.5}$$

$$a_0 = \frac{\sum y_i}{n} - a_1 \frac{\sum x_i}{n} \tag{2.6}$$

Como se observa en las ecuaciones 2.5 y 2.6, se necesita realizar sumas de forma secuencial para lo cual podemos usar un ciclo for.

El siguiente programa muestra la forma de llevar a cabo el cálculo de los coeficientes de una regresión lineal. En las líneas 4 y 5 se proporciona la colección de datos $[x_i, y_i]$, en la línea 9 obtenemos la longitud del vector x que se asigna a la variable N que será nuestra condición de paro en el ciclo for. En las líneas 14 a 17 se realiza la declaración e inicialización de las variables donde se acumularan las sumas. En las líneas 18 a 23 calculamos las sumas mediante un ciclo for que se ejecuta N veces. En la línea 25 calculamos a_1 y en la línea 27 se determina a_0. Finalmente graficamos la ecuación de la recta.

```
1  clear all;
2  clc;
3  %Datos de entrada
4  x = [1 2 3 4 5 6 7 8 9 10];
5  y = [0.5 1.8 3.2 4.3 5.01 6.4 6.8 7.1 8.1 10.2];
6  plot(x,y,'o');
7  grid on;
8  %Determinamos la longitud del vector x
9  N = length(x);
10 %Determinamos la suma de los elementos y_i
11 %Determinamos la suma de los elementos x_i
12 %Determinamos la suma del producto (x_i)(y_i)
13 %Determinamos la suma de x_i*y_i
14 suma_y = 0;
15 suma_x = 0;
16 suma_xx = 0;
17 suma_xy = 0;
18 for i = 1:N
19     suma_y = suma_y + y(i);
20     suma_x = suma_x + x(i);
21     suma_xx = suma_xx +x(i)^2;
22     suma_xy = suma_xy + x(i)*y(i);
23 end
24 %Generamos el dato de salida (Coeficiente a_1)
25 a_1 = (N*suma_xy-suma_x*suma_y)/(N*suma_xx-suma_x^2);
26 %Generamos el dato de salida (Coeficiente a_1)
27 a_0 = suma_y/N-a_1*suma_x/N;
28 %Salidad de datos
29 disp('a_0 = ');
30 disp(a_0);
31
32 disp('a_1 = ');
33 disp(a_1);
34 %Graficamos el ajuste
35 hold on;
36 plot(x,a_0+a_1*x);
```

2.8.3 Estructura *while*

La estructura *while* repite un bloque de código mientras la expresión 1 sea verdadera, la estructura se muestra en el siguiente listado:

Listado 2.30: Estructura while

```
1 while(expresión 1)
2
3         bloque de código
4
5 end
```

El siguiente programa es un ejemplo sencillo que muestra la forma de configurar el ciclo *while*. En la línea 3 se declara e inicia una variable *idx*, esta variable llevará el conteo de las veces que se ejecuta el ciclo *while*, en la línea 5 se declara el ciclo *while* con la expresión condicional $idx < 9$, esto es mientras idx tenga valores menores que 9 se ejecutan las líneas 6 y 7, en la línea 6 se incrementa una vez el valor de *idx*, por cada pasada que realiza el ciclo *while*, la línea 7 imprime el contenido del valor de *idx*, finalmente, en la línea 8 setá el limitador del ciclo *while*.

```
1 % Imprimir los números del 1 al 9.
2
3 idx = 0;
4
5 while(idx < 9)
6     idx =  idx +1;
7     disp(idx);
8 end
```

El siguiente programa muestra el uso del ciclo *while* para realizar la lectura de un vector, sumar los elementos del vector y finalmente obtener el promedio de los elementos. En la línea 5 se lee la cantidad de elementos a leer, en las líneas 8 a 10 se encuentra el ciclo *while* con el código a ejecutar, que para el caso es la lectura del elemento indexado por la variable *idx*, el ciclo *while* leerá N elementos, posteriormente de las líneas 16 a 19, con otro ciclo while se realiza la suma de los elementos, en la línea 20 se obtiene el promedio y en la línea 21 se muestra el promedio.

```
1  %Programa que calcula el promedio de n numeros,
2  %utilizando la instrucción while().
3  clear all;
4  %Se lee la cantidad de elementos que seran promediados
5  N = input('Cantidad de valores a promediar = ');
6  idx = 0;
7  %Se introduce cada elemento.
8  while(idx < N)
9   idx = idx +1;
10  disp(['Elemento(' int2str(idx) ')=']);
11  A(idx) = input('');
12 end
13 %Obtenemos la suma de los elementos
14 suma = 0;
15 idx = 0;
16 while (idx < N)
17  idx = idx +1;
18  suma = suma +A(idx);
19 end
20 prom = suma/N;
```

```
21 disp(['Promedio = ' num2str(prom)]);
```

2.9 Problemas Propuestos

2.9.1 Matrices

- Determine la solución del siguiente sistema de ecuaciones lineales

$$\begin{aligned} -x_1 + 2x_2 + x_4 &= 10 \\ x_1 + x_2 - x_3 + 2x_4 - x_5 &= 5 \\ x_3 - x_4 + x_5 &= 2 \\ x_1 + x_2 + 2x_3 + 3x_4 + 2x_5 &= 1 \\ -x_1 - x_2 - x_3 &= 1 \end{aligned} \tag{2.7}$$

- Encuentre el determinante e inversa de la matriz A.

$$A = \begin{bmatrix} 1 & -1 & 1 & 1 \\ 2 & 1 & 0 & 2 \\ 1 & -1 & 3 & 0 \\ 0 & 1 & -1 & 1 \end{bmatrix} \tag{2.8}$$

- Genere una matriz de 100 filas y 8 columnas con números aleatorios enteros en el rango de 0 a 100.
- Genere una matriz de 100 filas y 12 columnas de forma aleatoria con unos y ceros.

2.9.2 Graficación

- Grafique la función matemática $y = \dfrac{2x-1}{x^2-4}$ en el rango $[-40, 40]$, adicione un título a la gráfica y las correspondientes etiquetas al eje x e y.
- Grafique la función matemática $y = cos(\frac{1}{2}x) + 2sin(4x)$ en el rango de $[-4\pi, 4\pi]$, adicione un título a la gráfica y sus correspondientes etiquetas en los ejes x e y.

2.9.3 Secuencia $if - elseif - else$

- Usando la secuencia $if - else - if$ realice un programa para ver si un número es positivo o negativo.
- Usando la secuencia $if - else - if$ realice un programa para encontrar el mayor de tres números.
- Usando la secuencia $if - else - if$ realice un programa para ver si una persona es un niño, un joven, un adulto o un anciano dependiendo de su edad.

2.9.4 Ciclo for

- Escriba una programa para calcular el factorial de un número usando el ciclo for
- Escriba un programa usando el ciclo for para determinar el valor de y, si proporcionamos un valor determinado de N.

$$y = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{N^2} \tag{2.9}$$

2.9.5 Ciclo *while*

- Usando el ciclo *while* escriba un programa para encontrar los números primos contenidos en el intervalo de $[0, N]$.
- Usando el ciclo *while* escriba un programa para encontrar el valor de la serie dado un valor de N.

$$y = \sum_{k=1}^{N} \frac{1}{2^k} \tag{2.10}$$

Capítulo 3

Principios de Algoritmos Genéticos

3.1 Algoritmos Evolutivos Computacionales

Los orígenes de los Algoritmos Evolutivos (AE) tienen sus inicios entre los años 50's y 70's. La Tabla 3.1 muestra la clasificación existente de los AE durante este periodo [24].

Tabla 3.1: Fundadores de la Computación Evolutiva

Técnica	Fundador	Año
Programación Evolutiva	Fraser y Box	1957
Estratégias de Evolución	L.J. Fogel, A. J. Owens	1966
Algoritmos Genéticos	John Holland	1975

Por otro lado, la Computación Evolutiva (CE) es un campo reconocido recientemente y el término fue inventado en 1991 [1]. La CE representa el esfuerzo de unir a los investigadores que tienen diferentes acercamientos con la simulación de los diferentes aspectos de la evolución. Todas estas técnicas, Programación Evolutiva (PE), Estrategias de Evolución (EE) y Algoritmos Genéticos (AG), tienen muchos aspectos comunes, tales como la población de individuos, mecanismos de evolución sobre la población tales como selección, cruce, mutación, etc. Así, el contenido de este libro está enfocado al análisis, desarrollo y aplicación de los AG.

Las primeras investigaciones con AG fueron realizadas en la Universidad de Michigan por John Holland junto con sus colegas y estudiantes en 1975, Holland fue en primero en intentar desarrollar una base teórica para los AG a través del teorema de esquemas y analizar la posibilidad de las capacidades intrínsecas del paralelismo [17]. Sin embargo, uno de los estudiantes de Holland fue quien logró dar una aplicación de los AG en el control de flujo en gasoductos [14].

En la actualidad, a esta parte de la inteligencia artificial se le llama computación evolutiva y tiene una gran cantidad se esquemas que explotan el comportamiento natural de los seres vivos, tales como optimización por colonia de hormigas, cúmulo partículas, parvadas de pájaros, enjambre de abejas, forraje de bacterias, algoritmos genéticos, etc.

El enfoque del contenido de este libro abordará solo la parte de los algoritmos genéticos, en las siguientes secciones de este capítulo se mostrarán los fundamentos básicos de los algoritmos genéticos, la forma de programar sus operadores y la aplicación a problemas de optimización de funciones matemáticas sencillas usando Matlab®.

3.1.1 Definición

Los AG son técnicas de búsqueda y optimización basados en los principios de la genética y selección natural. Los AG están compuestos de un conjunto de individuos y diferentes tipos de reglas que aplican directamente sobre los individuos [15]. La figura 3.1 muestra las partes que integran un AG.

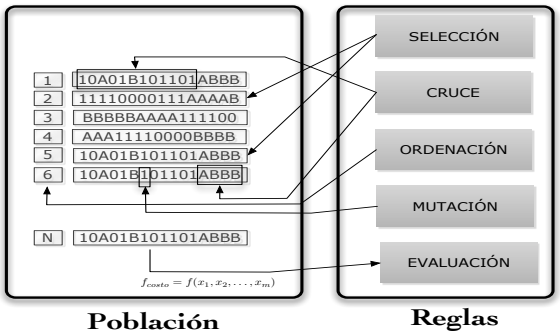

Figura 3.1: Elementos que integran un Algoritmo Genético

La Fig. 3.2 es una diagrama de flujo que muestra la estructura y orden de ejecución de un AG.

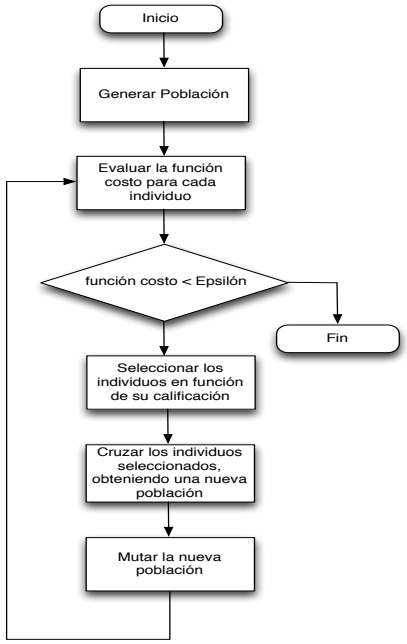

Figura 3.2: Estructura de ejecución de un Algoritmo Genético

La población es un conjunto de individuos y cada individuo se representa por una cadena de símbolos, letras y/o números, la forma de generar estos individuos es de forma totalmente aleatoria, después, esta cadena es evaluada en una función costo con la finalidad de obtener una calificación para cada individuo, mediante la calificación otorgada al individuo, se ordena y selecciona la población para obtener un conjunto

con los individuos mejor calificados, posteriormente mediante el operador de cruce seleccionamos de forma aleatoria los segmentos de información que van a compartir para generar nuevos individuos, cuando se termina de hacer el cruce de todos los individuos tenemos una nueva población que deberá tener una mejor calificación en la siguiente evaluación, sin embargo, antes de someterlo a una nueva evaluación es necesario mutar de forma aleatoria algunos individuos de la población, la mutación consiste en seleccionar de forma aleatoria algunos individuos y posteriormente en su cadena de genes cambiar la información. La descripción anterior es la forma de realizar y ejecutar un AG. La terminología que se usa para definir las partes que forma una población es la siguiente:

Tabla 3.2: Conceptos de los elementos que integran una población

Término	Descripción
Alelo	Cada uno de los estados distintos que puede presentar un gen en una misma posición.
Gen	Es el valor de un alelo dentro de un arreglo.
Cromosoma	Es una colección de genes en forma de arreglo.
Posición	Es el lugar que ocupa un gen dentro del cromosoma.
Índice	Es la posición que tiene el individuo dentro de la población

La siguiente Fig. 3.3 muestra los términos de alelo, gen, cromosoma, posición e índice en una población.

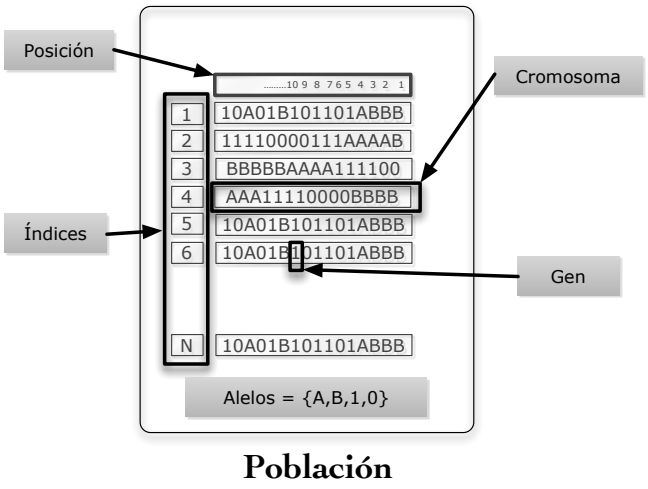

Figura 3.3: Alelo, gen, cromosoma, posición e índice en una población

Como todo algoritmo de solución tiene sus pros y contras, sin embargo, la justificación para aplicar estos algoritmos a la solución de problemas en ingeniería es que han tenido éxito en aquellas aplicaciones en donde las herramientas matemáticas gradientales de optimización han fracasado. A continuación se enumeran las ventajas y desventajas de los AG.

Ventajas

1. Implementación computacional. Las operaciones computacionales para realizar un AG es a través de operaciones aritméticas, lógicas y de ordenamiento sencillas que son fáciles de implementar.

2. Información a priori. Los AG no necesitan información a priori del sistema que se desea optimizar. El AG genera múltiples soluciones de forma aleatoria y si algunas de ellas mejoran, entonces, son soluciones factibles que se tomarán en cuenta para evolucionar la población.

3. Optimización de sistemas no lineales. El AG realiza un amplia exploración en el rango de búsqueda sin importar que el sistema sea de funciones discontinuas, que no tengan derivada o que sea no convexo.

4. Paralelismo. Los AG por su naturaleza son factibles de implementarse en clusters de cómputo paralelo, de forma tal, que el tiempo de cómputo para evaluar la evolución de un individuo es el mismo que para evaluar toda la población.

Desventajas

1. Función costo. La única forma para evaluar el desempeño de las evoluciones en el AG es a través de una función de evaluación o función costo, en ciertas aplicaciones no es sencillo establecer una función costo para optimizar un sistema multivariable.

2. Reglas. No existen reglas para determinar el número de individuos en una población, qué tipo de selección aplicar o cómo realizar la mutación.

3. Aplicaciones. Los AG genéticos tienen una amplia gama de aplicaciones donde han logrado optimizar sistemas con éxito, pero esto no quiere decir que se puedan utilizar en cualquier aplicación y que logren un mejor desempeño que los métodos analíticos matemáticos.

4. Programación serie. Cuando los AG genéticos se programan en plataformas de procesamiento serie los AG no son tan rápidos en su tiempo de ejecución y por lo tanto, es difícil implementarlos en aplicaciones en línea.

5. Convergencia prematura. Es probable que un individuo con un alto desempeño propague su código genético desde el arranque de la simulación, de tal forma, que se pierde diversidad en la población y el AG cae un punto óptimo local.

3.2 Algoritmos Genéticos Binarios

Básicamente un AG está integrado por una población de individuos y un conjunto se operadores que actúan sobre la población para ir obteniendo nuevas generaciones de individuos, con la finalidad de optimizar algún recurso. Los valores que puede tomar cada objeto dentro del chromosoma se le llama alelo y para el caso de los binarios es uno y cero, $alelo = \{1, 0\}$. La población está constituida por un conjunto de individuos y cada individuo se representa mediante un cromosoma equivalente a un número binario, a la cadena de unos y ceros se le llama **chromosoma**. En las siguientes secciones se describirán los AG binarios, se realizarán aplicaciones y se implementarán programas en Matlab® para optimizar funciones sencillas.

3.2.1 Población

La población es una colección de individuos que se generan de forma aleatoria y cada individuo tiene información que se encuentra codificada en su cromosoma, esta información contiene soluciones del problema a optimizar. Cuando la evolución del AG termina, el código genético de cada individuo es una solución que se encuentra en el óptimo o cercano al óptimo. El siguiente código de Matlab® es una función ($Inicializar\_pob$) que genera una población de forma aleatoria usando el sistema binario. Los argumentos que recibe la función son el número de individuos en la variable num_ind y la longitud del cromosoma en la variable $long\_crom$, por lo tanto, la función $rand(num\_ind, long\_crom)$ genera una matriz con num_ind renglones y $long\_crom$ columnas de números aleatorios en el intervalo de $[0, 1]$, esta matriz entra como argumento de la función

round() que redondea hacia el entero más próximo obteniendo así la matriz con elementos en unos y ceros, finalmente se asigna a la variable *población* la salida de la función.

```
1 function [ poblacion ] = Inicializar_pob(num_ind, long_crom)
2 %Función que inicia población
3 %Conjunto de alelos = {1,0}.
4 %num_ind = número de individuos en la población.
5 %long_crom = Longitud del cromosoma de cada individuo
6     poblacion = round(rand(num_ind, long_crom));
7 end
8
```

3.2.2 Codificación/Decodificación

Cada individuo se encuentra representando por su cromosoma que equivale a una cadena de unos y ceros, directamente puede ser interpretado como un número binario que tiene su equivalente en el sistema numérico decimal. Por lo tanto, el cromosoma esta codificado en el sistema numérico binario, pero además, cada individuo puede mapear un valor en el universo de discurso del sistema a optimizar y para conocer el valor que tiene cada individuo en el sistema se debe realizar un proceso de decodificación, para el caso, la decodificación es el proceso de conversión del sistema binario a el sistema decimal y posteriormente un escalamiento al universo de discurso del sistema que se desea optimizar. Ejemplo: Se tienen 6 individuos con cromosomas de 5 bits en un universo de discurso del sistema a optimizar entre [-1, 1]. Suponemos que la población está representada de la siguiente forma, Eq. 3.1:

$$poblacion_{(binaria)} = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{3.1}$$

Ahora se realiza la conversión de binario a decimal de cada fila de la Eq. 3.1 para obtener el valor de cada individuo en forma decimal.

$$poblacion_{(decimal)} = \begin{pmatrix} 23 \\ 27 \\ 15 \\ 24 \\ 19 \\ 14 \end{pmatrix} \tag{3.2}$$

Finalmente, realizamos una conversión a la escala deseada en la que se encuentra el universo de discurso de sistema a optimizar. El intervalo en decimal que podemos representar con 5 bits es de $valor_{dec} = [min_{dec}, max_{dec}] = [0, 31]$, y lo transformamos a el intervalo del sistema a optimizar $valor_{float} = [min_{float}, max_{float}] = [-1, 1]$, así el valor codificado $valor_{float}$ de cada individuo puede expresarse por la Ec. 3.3

$$valor_{float} = \frac{max_{float} - min_{float}}{max_{dec} - min_{dec}} (valor_{dec} - min_{dec}) + min_{float} \tag{3.3}$$

Sustituyendo los valores de los intervalos

$$valor_{float} = \frac{2}{31} (valor_{dec}) - 1 \tag{3.4}$$

Así la población codificada queda definida por

$$poblacion_{(decodificada)} = \begin{pmatrix} 0.4838 \\ 0.7419 \\ -0.0322 \\ 0.5483 \\ 0.2258 \\ -0.09677 \end{pmatrix} \tag{3.5}$$

Así la Ec. 3.5 es es el conjunto de valores que tienen los individuos en el universo de discurso del sistema a optimizar, sin embargo, el proceso de evolución de la población se lleva a cabo mediante operaciones realizadas directamente sobre la población representada por código binario, Ec. 3.1.

3.2.3 Función Costo

La **función costo** puede representarse de diversas formas, la más convencional es una función matemática, sin embargo, cualquier forma que permita evaluar el valor de cada individuo para determinar su desempeño dentro de la función objetivo puede considerarse una función costo.

La función costo proporcionará una forma cuantitativa del desempeño de cada individuo para llevar a cabo la optimización del sistema solicitado. La función costo se determina con el valor decodificado y escalado de cada individuo. Ejemplo: Determinar los valores de la función costo $f(x) = x^2$ de los individuos definidos en la Eq. 3.5.

$$f(x) = x^2 = \begin{pmatrix} (0.4838)^2 \\ (0.7419)^2 \\ (-0.0322)^2 \\ (0.5483)^2 \\ (0.2258)^2 \\ (-0.09677)^2 \end{pmatrix} = \begin{pmatrix} 0.2340 \\ 0.5504 \\ 0.0010 \\ 0.3006 \\ 0.0509 \\ 0.0093 \end{pmatrix} \tag{3.6}$$

La Eq. 3.6 nos proporciona una forma cuantitativa del desempeño de cada individuo en el proceso de optmización.

Si el objetivo es maximizar, entonces los mejores individuos, son aquellos que tienen el valor más grande en un función costo y estos individuos son los que tendrán mayor oportunidad de propagar su información genética a las siguientes generaciones. De forma contraria, si el objetivo es minimizar, los individuos que tendrán mayor probabilidad de propagar su código genético serán los que tengan en valor más pequeño en su función costo.

A continuación se muestran algunos tipos de funciones costo en dos dimensiones que se usan para probar el desempeño de los AG.

Función costo con un óptimo global. La forma ideal de una función objetivo es que esta tenga un máximo o minímo global en el universo de discurso donde es factible la solución, la Fig. 3.4(a) descrita por la Ec. 3.7 contiene un mínimo global y los cambios son suaves para acercarse desde cualquiera de los punto x_1, x_2 al mínimo global.

$$f(x_1, x_2) = x_1^2 + x_2^2 \tag{3.7}$$

Función costo con un óptimo global y diferentes óptimos locales. La Eq. 3.8 es una función que tiene un máximo global y distintos puntos con un máximo o mínimo global, la Fig. 3.4(b) es la representación de esta ecuación. Cuando la población de un AG se inicia con pocos individuos es muy probable quedar atrapado en un mínimo o máximo local en este tipo de funciones, si algunos individuos caen en puntos cercanos al máximo global, es probable que propaguen sus cromosomas a las siguientes generaciones de individuos y se acerquen al óptimo global.

$$f(x_1, x_2) = \frac{sin(\sqrt{x_1^2 + x_2^2})}{\sqrt{x_1^2 + x_2^2}} \tag{3.8}$$

Función costo con distintos óptimos globales. La Eq. 3.9 es una función que tiene óptimos globales en distintos puntos, la Fig. 3.5(a) es la gráfica de este tipo de función costo. En este tipo de funciones los individuos en un AG, podrían agruparse en distintos puntos ya que existen varios óptimos globales o puede presentarse el caso que el AG no logre la convergencia a ningún punto.

$$f(x_1, x_2) = sin^2(x_1) + cos^2(x_2) \tag{3.9}$$

Función costo con diferentes óptimos locales cercanos. La Eq. 3.10 es una función con un punto óptimo máximo global y diferentes puntos óptimos locales y todos los puntos en su óptimo global y local se encuentran cercanos, la Fig. 3.5(b) muestra la gráfica de esta función. La convergencia prematura sucede cuando un subconjunto de la población de individuos quedan aglutinados en un mínimo o máximo local, que posteriormente cuando evoluciona el AG, estos individuos dominan a la población evitando que se exploren dominios donde posiblemente se encuentre el máximo o mínimo global. Para evitar este tipo problema se trata de hacer una distribuciónn uniforme de los individuos en todo el domino de la función costo.

$$f(x_1, x_2) = x_2 sin^2(2x_1) + 2x_1 cos^2(2.5x_2) \tag{3.10}$$

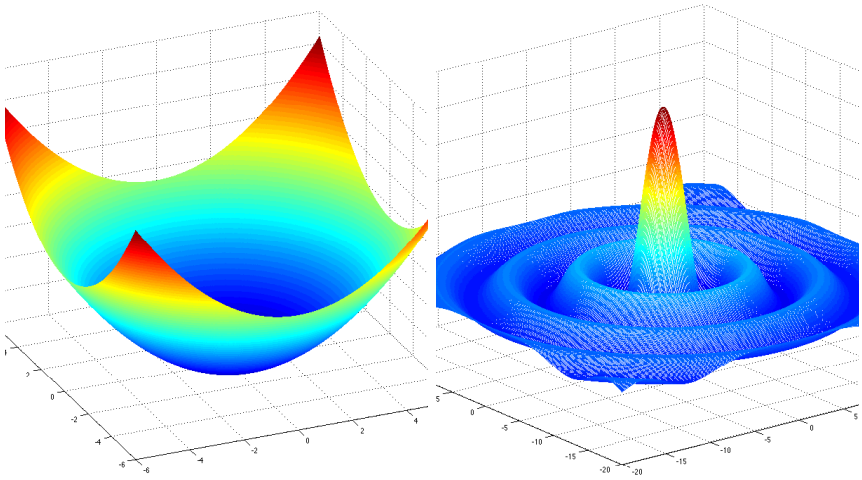

Figura 3.4: (a) $f(x_1, x_2) = x_1^2 + x_2^2$, (b) $f(x_1, x_2) = \frac{sin(\sqrt{x_1^2+x_2^2})}{\sqrt{x_1^2+x_2^2}}$

3.2.4 Selección

Este es el operador que tiene mayor impacto en el proceso de optimización de sistemas, ya que el operador selección es el encargado de transmitir el código genético de los individuos mas aptos a las futuras generaciones con la finalidad de llevar a cabo el objetivo de optimización, sin embargo, esto no quiere decir que sólo se tome en cuenta los mejores, ya que al hacer esto, el algoritmo rápidamente pierde diversidad y difícilmente

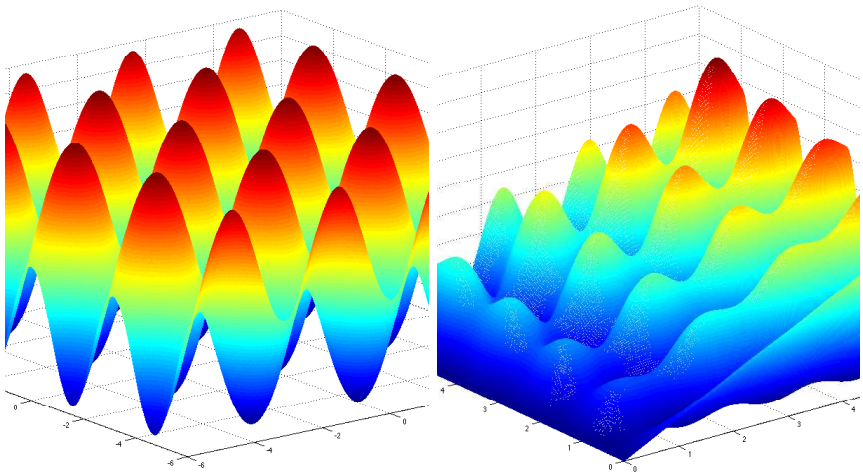

Figura 3.5: (a) $f(x_1, x_2) = sin(x_1)^2 + cos(x_2)^2$, (b) $f(x_1, x_2) = x_2 sin^2(2x_1) + 2x_1 cos^2(2.5x_2)$

Tabla 3.3: Población de 6 individuos con 8 cromosomas

Índice	Individuo (binario)	Decodificacion x(decimal)	Función costo f(x)
1	11110101	245	0.0419
2	10010001	145	8.9083
3	10111110	190	2.1490
4	01101011	107	8.3828
5	10100111	167	0.4422
6	00100110	38	0.3916

podría alcanzar el óptimo global. Existen diferentes estrategias de selección que han sido investigadas y probadas con la finalidad de lograr que las futuras generaciones mejoren a los papás y como consecuencia lograr una rápida convergencia al óptimo. A continuación se mostrarán algunos métodos de selección y para ilustrarlos se usará una población de 6 individuos con 8 cromosomas en un universo de discurso de $[0, 255]$ que es el intervalo que podemos representar con 8 bits, la función costo está definida por:

$$f(x) = Ae^{-\frac{(x-b)^2}{c^2}} \tag{3.11}$$

Donde $A = 10$, $b = 128$ y $c = 50$ son constantes, La siguiente tabla muestra los seis individuos con sus cromosomas de 8 bits, su valor en decimal y el valor que tiene el individuo al ser evaluado en la Ec. 3.11. La Fig. 3.6 muestra la función costo y la ubicación de cada individuo de la Tabla 3.2.4 en la función.

Selección directa - El método de la selección directa es realizar un cruce desde el índice inferior al índice superior, este método permite que individuos poco adaptados se relacionen con individuos mejor adaptados, sin embargo, se puede ordenar a la población desde individuos bien adaptados hasta los mal

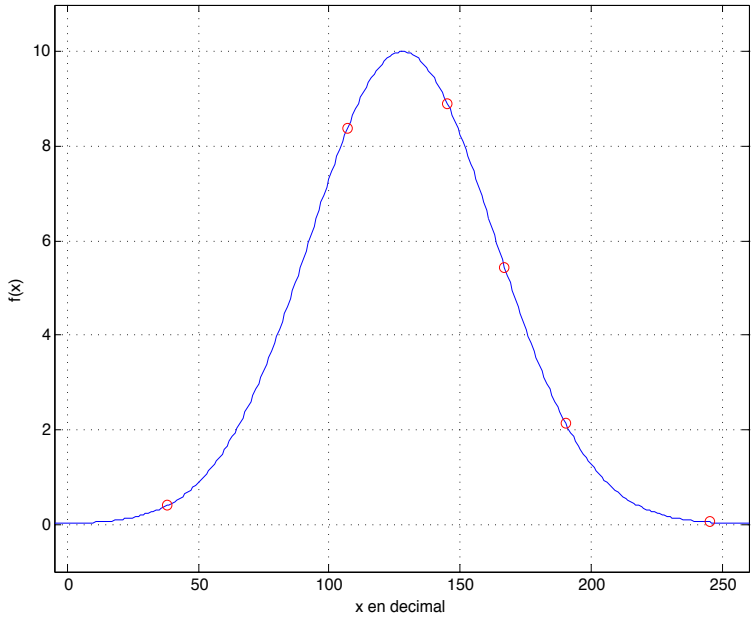

Figura 3.6: Seis individuos evaluados en la función $f(x) = Ae^{-\frac{(x-b)^2}{c^2}}$

adaptados y realizar la selecciones de arriba a abajo y de esta forma solo habrá cruces entre los mejores ponderados y cruces entre los individuos con mala calificación en su función costo. Si tomamos como referencia la Tabla 3.2.4 aplicando el primer método sin ordenación la selección de parejas a cruzar serían, el individuo 1 con 2, 3 con 4 y 5 con 6. Así el individuo 1 tiene un bajo factor de adaptación 0.0419 y se cruzará con el individuo 2 que tiene un mejor factor de adaptación 2.1490. Algo similar sucede con los individuos 3 y 4, donde se da un cruce entre individuos altamente adaptados y finalmente 5 con 6 tienen una adaptación regular.

Selección por torneo Se seleccionan de forma aleatoria dos individuos de la población, posteriormente se genera un número aleatorio entre $[0, 1]$ y se pone un umbral, por ejemplo $umbral = 0.7$ y aplicamos la siguiente regla condicional:

$$\begin{array}{l} \text{Si aleatorio} < \text{umbral} \\ \text{entonces (seleccionamos al mejor adaptado)} \\ \text{else (seleccionamos al peor adaptado)} \end{array} \tag{3.12}$$

Este método de selección tiene cierto grado de elitismo ya que le da mayor probabilidad de selección a los mejor adaptados (0.7) y le da menos probabilidad a los mal adaptados (0.3). Este método no tiende a producir una convergencia prematura y es fácil de implementar computacionalmente.

Selección proporcional - En el método de selección proporcional, o de la ruleta, cada indivividuo x_i tiene una probabilidad de cruce proporcional a la calificación que tiene en la función costo $f(x_i)$. Donde la probabilidad de selección se obtiene por:

$$P(x) = \frac{f(x_i)}{\sum f(x_i)} 100\% \tag{3.13}$$

La Tabla 3.2.4 en su última columna tiene la aplicación de la Eq. 3.13, así el individuo 1 tiene la probabilidad de selección del 0.21%, el individuo 2 tiene la probabilidad de selección de 10.58% y así sucesivamente.

Posteriormente, se genera un vector de números aleatorios en el rango $[0, 100\%]$ de longitud equivalente a número de individuos en la población y se ordenan de forma ascendente; ahora se compara la probabilidad de selección del individuo con la probabilidad del vector aleatorio ascendente. Si el valor del vector aleatorio cae en el intervalo del individuo, entonces seleccionamos dicho individuo. El siguiente programa muestra la forma de implementar el operador selección usando el método de la ruleta.

Tabla 3.4: Probabilidad de selección para la población de 6 individuos con 8 cromosomas

Índice	Individuo (binario)	Decodificación x(decimal)	Función costo f(x)	Probabilidad de selección P(x)
1	11110101	245	0.0419	0.21 %
2	10111110	190	2.1490	10.58 %
3	10010001	145	8.9083	43.85 %
4	01101011	107	8.3828	41.26 %
5	10100111	167	0.4422	2.17 %
6	00100110	38	0.3916	1.93 %
			$\sum f(x_i) = 20.3158$	

```
function [PN] = seleccion(P, long)
%Determinamos la probabilidad de selección de cada individuo
    P(:,long+3) = P(:,long+2)/sum(P(:,long+2));
%Determinamos la suma parcial de las  probabilidades acumuladas
    P(:,long+4) = cumsum(P(:,long+3));

    [M N] = size(P);
    alea = sort(rand(M,1));
    i1 = 1;
    i2 = 1;

    while (i1 <= M)
        if(i2 > M)
            break;
        end
        while(alea(i2) < P(i1,long+4) )
            PN(i2,:) = P(i1,:);
            i2 = i2 +1;
            if i2 > M
                break;
            end
        end
        i1 = i1 +1;
    end
end
```

3.2.5 Cruce

El operador cruce es la acción de seleccionar dos individuos llamados padres y aparearlos para obtener nuevos individuos llamados hijos. Como consecuencia los hijos son el resultado de una combinación de cromosomas de los padres y así cada hijo contiene información genética de los papás. La Fig. 3.7 muestra una población

de n individuos, con una longitud en sus cromosomas de 12. Para el ejemplo, se muestra el contenido genético de 7 individuos, mediante una técnica de selección se toman dos individuos que funcionarán como papás, en este caso los papás 2 y 5, despúes, de forma aleatoria se selecciona el punto de cruce que es 6. El punto de cruce es un valor desde uno hasta la longitud del cromosoma menos uno. El punto de cruce nos permite partir el cromosoma en dos partes para cada papá y entonces tener cuatro cadenas de cromosomas que se intercambian como se muestra en el figura 3.7, para generar dos hijos, cada hijo tiene información cromosómica de los papás. La forma clásica de un AG es usando un punto de cruce que se genera de forma aleatoria, sin embargo, es posible usar dos o más puntos de cruce. Por otra parte, existen diferentes estrategias para realizar la seleción de individuos, en el siguiente punto se muestran dichas técnicas. La forma de implantar el operador cruce en Matlab®, se reporta en el siguiente código.

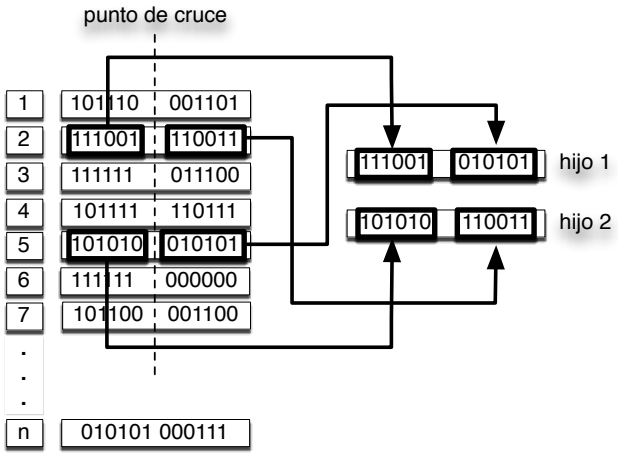

Figura 3.7: Apareamiento de dos individuos que generan dos hijos

En el siguiente código, la función $randperm(M)$ genera un vector permutado con números aleatorios entre 1 y el número de individuos M y los asigna como vector a la variable t. La función $reshape(t, M/2, 2)$ reordena el vector t en una matriz de $M/2$ filas por 2 columnas. De esta forma tenemos por cada fila de la matriz los índices de los padres que se cruzarán.

Dentro de un ciclo for realizamos el cruce de los individuos, inicialmente generamos un número aleatorio entre 1 y $long - 1$ en la variable $pcruce$ y esta variable representa el punto de cruce para cada fila de la población. Se usan cuatro variables auxiliares para guardar los segmentos de los cromosomas de cada papá y después se recombinan en la variable PNC que representa la población nueva cruzada. Ya que en cada pasada del ciclo for se guardan dos individuos hijos, es necesario incrementar el índice de PNC en dos que para el caso es la variable i_1

```
1  function [PNC] = cruce(PN,long)
2
3  [M N]=size(PN);
4  PNC = zeros(M,long+4);
5
6  t = randperm(M);
7
8  fc= reshape(t,M/2,2);
9  i1 = 1;
10 for i = 1:M/2
11     %Probabilidad de cruce
12     %probc = 0.5;
```

```
13
14      %generamos un punto de cruce de forma
15      %aleatoria desde 1 hasta long-1
16      pcruce = randi(long-2)+1;
17
18      aux1 = PN(fc(i,1),1:pcruce);
19      aux2 = PN(fc(i,1),pcruce+1:long);
20      aux3 = PN(fc(i,2),1:pcruce);
21      aux4 = PN(fc(i,2),pcruce+1:long);
22
23      PNC(i1,1:long) = [aux1 aux4];
24      PNC(i1+1,1:long) = [aux3 aux2];
25      i1 = i1+2;
26  end
27  end
```

3.2.6 Mutación

La mutación es una forma de explorar nuevas alternativas de lograr individuos que produzcan nuevas soluciones, la mutación se realiza modificando de forma aleatoria los genes de una población en un cierto porcentaje. Por ejemplo si tenemos una población de 10 individuos con cromosomas de 8 bits, entonces tenemos 80 genes que integran la población y si consideramos un porcentaje de mutación del 5%, debemos cambiar de forma aleatoria a cuatro genes en la población.

El siguiente código muestra la implantación del operador mutación en Matlab®, la función *mutacion* recibe como argumento PN = Población Nueva o la nueva generación de individuos, $long$ = longitud de los cromosomas y $prob$ = porcentaje de mutación. $totgenes$ = total de genes en la población y se obtiene por el producto del número de individuos y la longitud del cromosoma. En *genesmut* se obtiene la cantidad de genes que se van a mutar en la población y de esta forma se genera un ciclo for para contabilizar el número de mutaciones. Dentro del ciclo for, generamos dos números aleatorios, uno de ellos, entre 1 y el total de individuos y el otro, entre uno y la longitud del cromosoma, mediante el comando $randi()$ y asignados a las variables fil y col, dichas variables serán el índice para modificar un gen dentro de la población, finalmente se realiza la negación del gen y se repite el for el número de veces determinado en la variable *genesmut*.

```
1  function [ PN ] = mutacion(PN,long,prob)
2
3  %Se determina el número de genes que tiene la población
4  [M N] = size(PN);
5
6  totgenes=M*long;
7
8  genesmut = round(totgenes*prob/100);
9
10 for i = 1:genesmut
11  fil = randi([1 M]);
12  col = randi([1 long]);
13  %Se realiza la negación del gen seleccionado
14  PN(fil,col) = 1-PN(fil,col);
16 end
```

3.2.7 Algoritmo Genético Binario

El siguiente programa es la integración de los operadores genéticos descritos anteriormente para llevar a cabo el proceso de optimización de una función costo.

```
function y = ruleta(nump,long,xini,xfin,numt,prob,fun)

% nump     -> numero de individuos
% long     -> longitud de cromosoma
% xini     -> limite inferior
% xfin     -> limite superior
% numt     -> numero de iteraciones maximo
% prob     -> probabilidad de mutacion
% fun      -> funcion a maximizar


%Graficar función en el intervalo
x=xini:0.001:xfin;
z=fun(x);

% Inicializar población nueva en ceros
PN=zeros(nump,long+4);

%Población aleatoria
P=round(rand(nump,long+4));


for k=1:numt

    %Decodificar los individuos de la población
      totcomb =2^(long)-1;
      valordecimal = bin2dec(num2str(P(:,1:long)));
      P(:,long+1)=valordecimal/(totcomb)*(xfin-xini)+xini;

    %Calcular la función costo para cada individuo
      P(:,long+2)=fun(P(:,long+1));


    %Graficamos la función a optimizar
    plot(x,z,'r-')
    hold on

    %Graficamos la posición de los individuos
    plot(P(:,long+1),P(:,long+2),'bo')
    hold off
    pause;

    %Realizamos la selección de la Población
    %usando el método de la ruleta
    PN = seleccion(P,long)

    %Realizamos el cruce de la población
    PN = cruce(PN,long);


    %Realizamos la mutación de la población
    PN = mutacion(PN,long,prob);

    %Actualizamos la población
    %con la nueva generación
    P = PN;
end
```

3.3 Pruebas y Resultados

En esta sección se reportan dos casos de optimización de funciones en dos dimensiones con el método del torneo. Es recomendable realizar diferentes pruebas ajustando el número de individuos, el porcentaje de mutación, el tipo de función, etc. para tomar experiencia en la optimización usando AG. A continuación se reportan dos ejemplos, el primero es una función suave con un máximo global y la segunda es una función suave con múltiples máximos locales, un máximo global y los puntos máximos se encuentran cercanos entre si.

Ejemplo 1: La función Gaussiana es una función de costo sencilla que sólo contiene un máximo global, lo función está definida por:

$$f(x) = Ae^{-\frac{(x-b)^2}{c^2}} \tag{3.14}$$

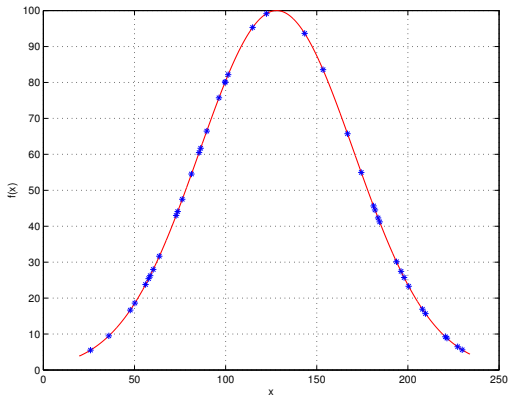

Figura 3.8: Población inicial distribuida en el universo de discurso de la función a optimizar

Donde las constantes de la función tiene los siguientes valores $A = 100$, $b = 128$ y $c =$, la función tiene el máximo global en $x = 128$ y $f(x) = 100$. Los parámetro para el AG binario son los siguientes: numero de individuos $= 40$, intervalo de búsqueda $= [20, 234]$, longitud del cromosoma $= 8$, número de evoluciones $=$ 100 y porcentaje de mutación $= 1\%$.

La Fig. 3.8 muestra el arranque de la simulación, todos los individuos se encuentran dispersos en el universo de discurso, posteriormente cuando el AG ha realizado 25 evoluciones los individuos se empiezan a agrupar alrededor el máximo global (Fig. 3.9).

En la evolución 50 (Fig. 3.10) continua la agrupación de individuos en el punto del máximo global la dispersión ha disminuido.

Finalmente, en la evolución 75 la mayoría de los individuos de la población ya se encuentran agrupados en un extremo del máximo global (Fig. 3.11).

Ejemplo 2: El siguiente ejemplo realiza la optimización de la función 3.15 que tiene un conjunto de diferentes máximos locales y un máximo global como puede verse en la Fig. 3.12.

$$y = Ae^{-\frac{(x-b)^2}{c^2}} cos^2(wx) \tag{3.15}$$

Donde las constantes de la función tiene los siguientes valores $A = 10$, $b = 0$, $c = 20$ y $w = \frac{1}{2}$. Los parámetro para el AG binario son los siguientes: numero de individuos $= 80$, intervalo de búsqueda $= [-40, 40]$, longitud del cromosoma $= 14$, número de evoluciones $= 100$ y porcentaje de mutación $= 0.2\%$.

La Fig. 3.12 muestra la población inicial distribuida en el universo de discurso $[-40, 40]$. Posteriormente, después de 15 evoluciones la mayor parte de los individuos se agrupan alrededor del máximo global y un

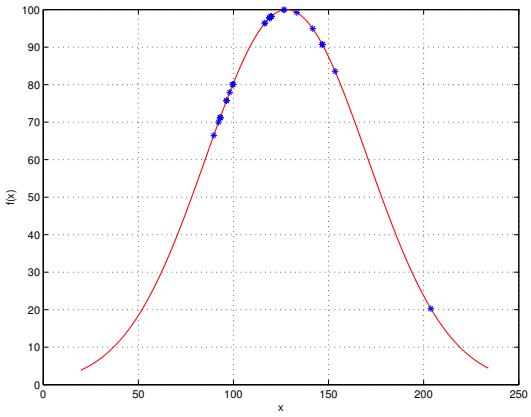

Figura 3.9: Población evolucionada después de 25 generaciones

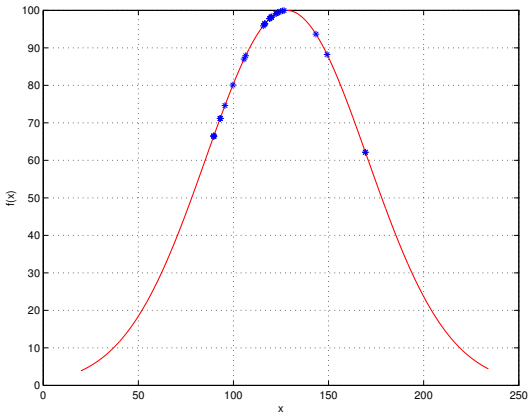

Figura 3.10: Población evolucionada después de 50 generaciones

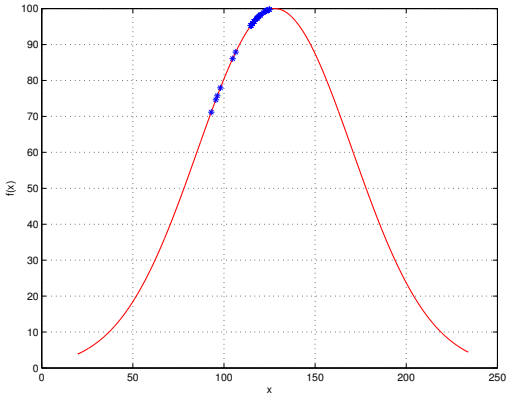

Figura 3.11: Población evolucionada después de 75 generaciones

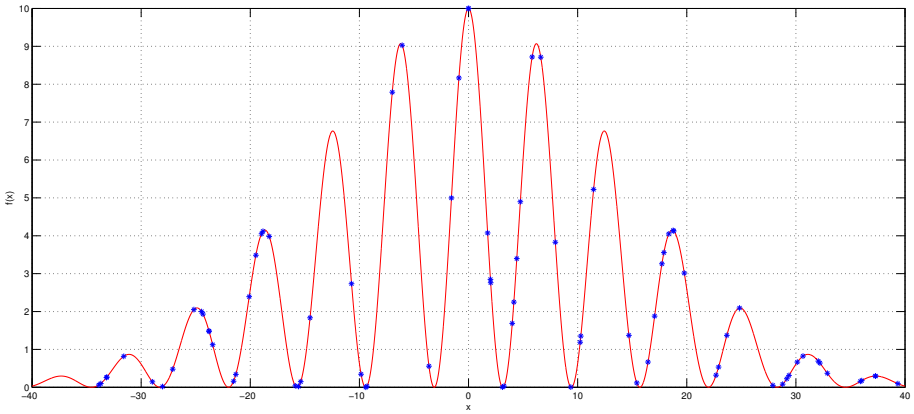

Figura 3.12: Población inicial distribuida en el universo de discurso de la función a optmizar

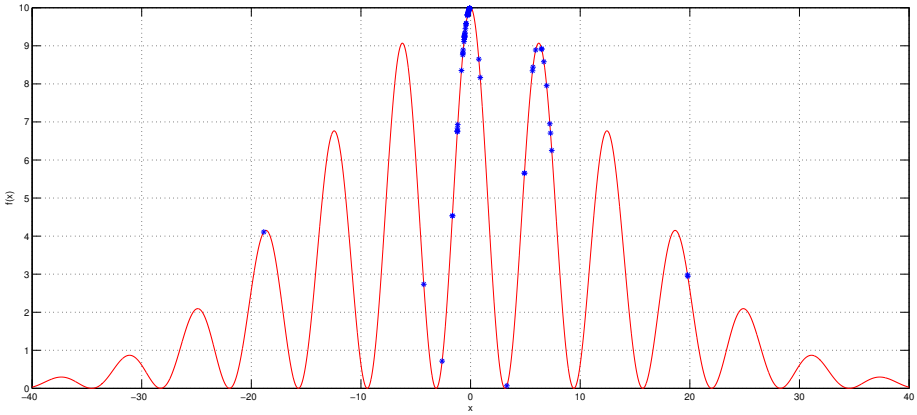

Figura 3.13: Evolución de la población en la generación 15

máximo local y existen otros individuos que aún se encuentran alejados del máximo global pero cercanos a otros máximos locales (Fig. 3.13). Para la evolución 30 la mayor parte de la población ya se encuentra agrupada en el máximo global con muy pocos individuos dispersos (Fig. 3.14).

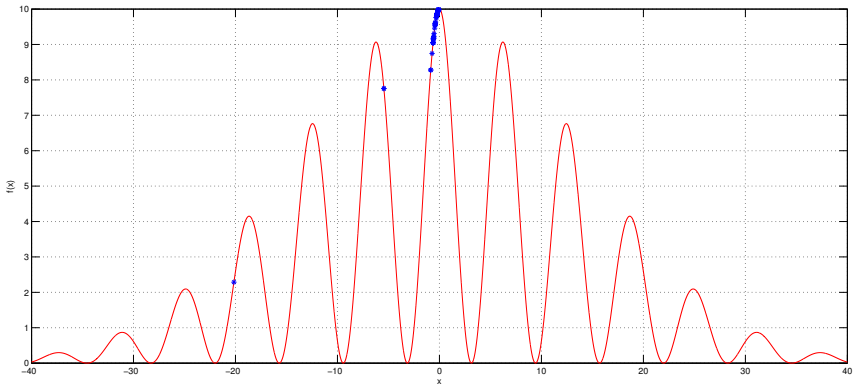

Figura 3.14: Evolución de la población en la generación 30

Finalmente, la evolución 45, los individuos de la población se encuentran fuertemente agrupados en el máximo global (Fig. 3.15). Podemos decir que se ha identificado el máximo global y que los individuos que tenían mejor capacidad de maximizar la función propagaron su información a los otros individuos ocasionando que la mayor parte de ellos convergieran al máximo global.

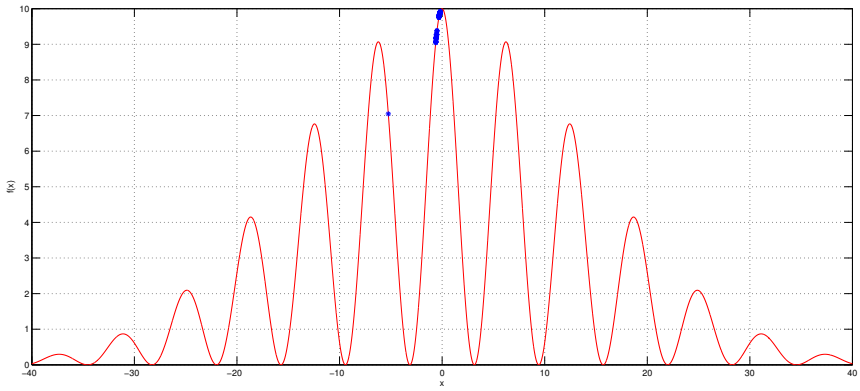

Figura 3.15: Evolución de la población en la generación 45

Capítulo 4

Problema de Empacado en Contenedores

En este capítulo se abordará el problema de empacado de contenedores (bin parking) y su solución por medio de algoritmos genéticos. La idea es utilizar todas las herramientas previamente expuestas sobre Matlab® y algoritmos genéticos con el fin de abordar un problema cuya formulación es muy sencilla pero para el cual es imposible calcular en un tiempo razonable una solución cuando el número de elementos involucrados aumenta. Se verá como plantear y modificar el modelo clásico de un algoritmo genético para dar solución al problema mencionado anteriormente, aplicando varias funciones que ofrece Matlab® con el fin de obtener un algoritmo sencillo y de fácil implementación.

4.1 Descripción del Problema de Empacado en Contenedores

El problema de empacado en contenedores (bin-packing problem) consiste en empacar n objetos en a lo más un número n de contenedores. En la versión unidimensional del problema, los contenedores se consideran homogéneos y con una capacidad de espacio limitado, sin tomar en cuanta la geometría ni de los objetos ni de los contenedores. Cada objeto ocupa un espacio $p_i > 0$ y cada contenedor tiene una capacidad limitada c. El problema es encontrar la mejor asignación de objetos en los contenedores de tal forma que no se exceda la capacidad de cada contenedor y se minimize el número de contenedores utilizado.

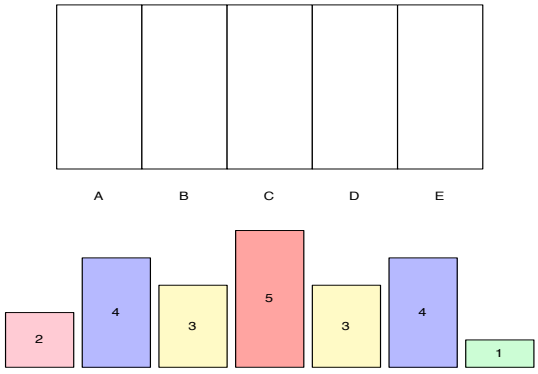

Figura 4.1: Contenedores y objetos para ilustrar el problema de empacado en contenedores

Para ilustrar el problema de forma gráfica, tomemos la Fig. 4.1 con 5 contenedores (A,B,C,D,E) y 7 objetos cuyos pesos están indicados por el número en cada uno de ellos. Un posible empacado de los objetos se muestra en la Fig. 4.2.

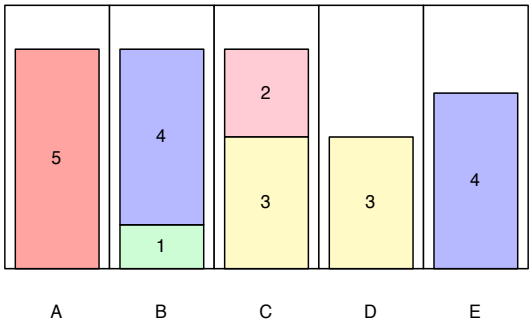

Figura 4.2: Primera opción de empacado de los objetos en los contenedores

Es claro que el empacado de la Fig. 4.2 no es único ni es óptimo en el sentido de que se pueden conseguir otros acomodos usando un menor número de contenedores, como el que se muestra en la Fig. 4.3.

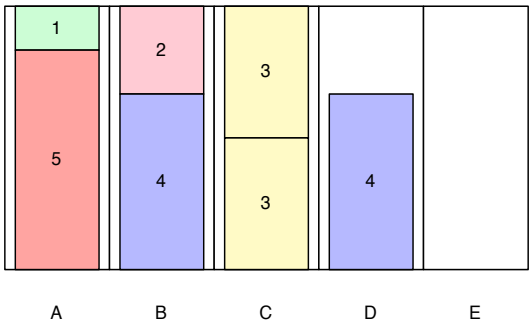

Figura 4.3: Opción más eficiente de empacado de los objetos en los contenedores

Una formulación matemática del problema de empacado en contenedores tomada de [13] es la siguiente:

$$\min z(\mathbf{y}) = \sum_{i=1}^{n} y_i \tag{4.1}$$

sujeto a:

$$\sum_{j=1}^{n} p_j x_{ij} \leq c y_i, \qquad i \in \mathbb{N} = \{1, 2, \ldots, n\} \tag{4.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j \in \mathbb{N} \tag{4.3}$$

$$y_i = 0 \text{ ó } 1, \qquad i \in \mathbb{N} \tag{4.4}$$

$$x_{ij} = 0 \text{ ó } 1, \qquad i, j \in \mathbb{N} \tag{4.5}$$

donde

$$y_i = \begin{cases} 1 & \text{si se usa el contenedor } i \\ 0 & \text{cualquier otro caso} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{si se asigna el objeto } j \text{ al contenedor } i \\ 0 & \text{cualquier otro caso} \end{cases}$$

En este caso, las variables x_{ij} y y_i son variables binarias que indican el uso o no de un recurso, llámese contenedores u objetos respectivamente. De este modo, la Ec. 4.1 implica tomar el mínimo de contenedores necesarios para empacar todos los objetos. La Ec. 4.2 es una restricción que revisa que el espacio ocupado por los objetos asignados a un contenedor no rebase la capacidad del mismo y la Ec. 4.3 examina que un objeto sea empacado en uno y solo un contenedor.

De esta forma, las variables x_{ij} definen una matriz X en donde cada columna tiene una única entrada igual a 1 y el resto igual a 0; es decir, X es una matriz $0-1$.

4.2 Relevancia del Problema de Empacado en Contenedores

A pesar de su sencillez, la naturaleza combinatoria del problema de empacado en contenedores lo hace un problema NP-duro, es decir, que no existe un algoritmo que pueda resolver todas las instancias del problema en un tiempo polinomial con respecto al número de objetos de entrada [9].

El problema de empacado en contenedores tiene gran relevancia en el llenado de contenedores de transporte, en la carga de camiones de carga con capacidad de peso, en la creación de archivos de respaldo en medios portátiles, en la distribución de tareas en procesadores homogéneos, en la organización de archivos de computadora en bloques de memoria de tamaño fijo, o en el diseño de circuitos.

Con el fin de solucionar instancias del problema para un número grande de objetos, se han establecido varios algoritmos que funcionan siguiendo estrategias muy sencillas, como son:

Algoritmo del ajuste siguiente (AS): En este algoritmo, se enlistan los objetos, el primero se asigna al contenedor 1, el resto de los objetos se asigna al contenedor actual si cabe; de otro modo, se asigna al siguiente contenedor. El nuevo contenedor se convierte en el contenedor actual.

Algoritmo del primer ajuste (PA): En este algoritmo, se enlistan los objetos como los contenedores. El primer objeto se asigna al contenedor 1, el resto de los objetos se asigna al contenedor con el menor índice donde quepa. Si el objeto no cabe en ningún contenedor; se crea uno nuevo donde se asigna. El nuevo contenedor se indexa incrementando en uno el mayor índice que se tenga.

Algoritmo del mejor ajuste (MA): Este algoritmo es una modificación del AS en el cual se asigna el objeto en curso al contenedor que tenga el mínimo espacio disponible capaz de albergar al objeto. En caso de empate entre dos o más contenedores, se elige aquel que tenga el menor índice.

Con el fin de mejorar estos algoritmos, como paso previo se puede reordenar a los objetos dependiendo del espacio que ocupen en forma descendente:

$$p_1 \geq p_2 \geq \cdots \geq p_n$$

y sobre este arreglo ordenado de objetos, aplicar los algoritmos AS, PA o MA. En este caso, los algoritmos modificados se denominan ajuste siguiente decreciente (ASD), primer ajuste decreciente (PAD) y mejor ajuste decreciente (MAD), respectivamente.

Los algoritmos anteriores se utilizan tanto para probar la efectividad de nuevos métodos al comparar resultados, como al incluirlos como parte del proceso de un algoritmo genético para mejorar su desempeño. En este caso, utilizaremos el MAD par implementar un algorimto genético con el fin de solucionar el problema de empacado en contenedores.

4.3 Representación Genética del Problema

Usualmente se han utilizado tres esquemas en la aplicación de algoritmos genéticos para representar problemas de empacado; estos son:

- Representación basada en contenedores.
- Representación basada en objetos.
- Representación basada en grupos.

4.3.1 Representación Basada en Contenedores

En esta perspectiva, cada cromosoma codifica la membrecía de los objetos en los contenedores; es decir, el índice de un gen representa al objeto y su valor corresponde al empaque en donde está colocado el objeto. Por ejemplo, un cromosoma 432214 codifica el empaque donde el primer y el sexto objeto están en el contenedor 4, el segundo en el contenedor 3, el tercero y cuarto en el contenedor 2 y el quinto en el contenedor 1. Un ejemplo se presenta en la Fig. 4.4.

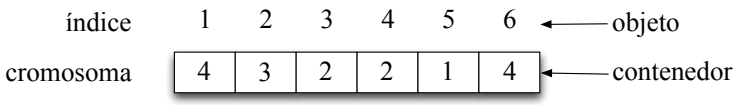

Figura 4.4: Ejemplo de la representación basada en contenedores

En esta representación, los cromosomas tienen una longitud constante igual al número de objetos, lo que permite una implementación directa de los operadores genéticos. Sin embargo, el manejo de la información de esta manera tiene severas desventajas [11]; por ejemplo, esta codificación es altamente redundante ya que el costo de la solución depende del número de contenedores utilizados más que de la numeración de los objetos. De este modo, los cromosomas 123144 y 314322 codifican la misma solución ya que en ambos casos, el primer y el cuarto objeto son asignados a un contenedor, el quinto y sexto objeto también comparten el mismo contenedor y el segundo y el cuarto objeto se empacan en contenedores individuales.

Bajo este esquema, el número de cromosomas que representan a la misma solución (es decir, la redundancia de las soluciones) crece exponencialmente con el número de empaques. Así el tamaño del espacio que el algoritmo genético tiene que explorar es mucho mayor que el espacio real en donde se encuentran las soluciones, lo que disminuye la efectividad del proceso.

Este problema se ve agravado al aplicar de manera directa las operaciones genéticas; en particular, el cruce de soluciones. Generalmente, un cruce de individuos idénticos debe producir una solución que tenga información equivalente a la de los padres. Sin embargo, en este caso el cruce de individuos puede generar

soluciones las cuales no correspondan al contexto del problema. Por ejemplo, tomemos el cruce de la Fig. 4.5.

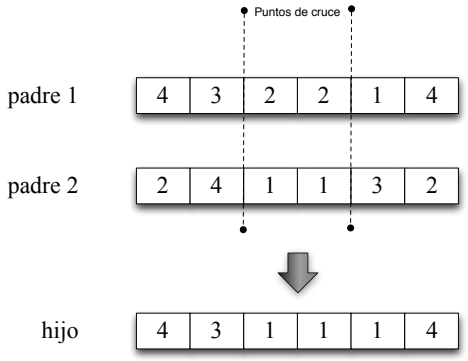

Figura 4.5: Cruce de soluciones tomando una representación basada en contenedores

En este ejemplo, cada padre tiene cuatro contenedores, pero el hijo resultante solo tiene tres. Esta operación puede llevar a producir soluciones que no sean factibles para el problema, en el sentido de que muchos objetos pueden asignarse a un solo contenedor, excediendo su capacidad.

4.3.2 Representación Basada en Objetos

Otro tipo de representación es codificar las permutaciones de objetos y después aplicar un descodificador para obtener la solución correspondiente. Supongamos que tenemos seis objetos numerados del 1 al 6. Una posible codificación sería 231654. Para descodificar la solución, tomemos la permutación de izquierda a derecha y asignemos los objetos a cada contenedor conforme estos se vayan llenando; de este modo, la solución puede quedar dividida de la siguiente manera:

$$231 \mid 65 \mid 4$$

En este caso tenemos tres contenedores, el primero con los objetos 2, 3 y 1; el segundo con los objetos 6 y 5; y por último el tercero con el objeto 4. Sin embargo, cualquier otra permutación que conserve el orden de los contenedores (como 312564 ó 465231) codifica la misma solución. Al igual que en el caso anterior, el grado de redundancia crece exponencialmente con el número de empaques, lo que disminuye la efectividad del algoritmo genético.

Por último, dado que la descodificación toma las permutaciones de izquierda a derecha, el acomodo de un objeto depende seriamente de la "cabeza" del cromosoma; es decir, del objeto más a la izquierda, lo que sesga la generación de buenas soluciones. Un punto a favor de este esquema es que nunca se generan soluciones no factibles.

4.3.3 Representación Basada en Grupos

Las dos representaciones anteriores no son adecuadas para problemas de grupos como el de empaque en contenedores ya que sus cromosomas están basados en objetos mas que en grupos de ellos. Por lo tanto, los cromosomas que se tomarán en esta sección serán basados en grupos de objetos. Tomemos el cromosoma 432214 presentado en la Sección 4.3.1, el cual indica que el contenedor 1 tiene al objeto 5, el contenedor 2 tiene a los objetos 3 y 4, el contenedor 3 tiene al objeto 2 y el contenedor 4 tiene a los objetos 1 y 6.

Un cromosoma adecuado debe representar contenedores mas que objetos individuales. Para ilustrar este punto, utilizaremos letras para representar a dichos contenedores de la siguiente manera:

$$A = \{5\} \quad B = \{3, 4\} \quad C = \{2\} \quad D = \{1, 6\}$$

Así el cromosoma que representa al empacado anterior es $ABCD$. Los operadores genéticos se aplicarán ahora sobre cromosomas que representan grupos de objetos (un grupo para cada contenedor). Esto implicará utilizar cromosomas de longitud variable.

4.4 Operaciones Genéticas para la Representación Basada en Grupos

Las operaciones que aquí se presentan se basan en los trabajos presentados en [11] y [13].

4.4.1 Cruce de Cromosomas

El cruce de cromosomas se ilustra en la Fig. 4.6.

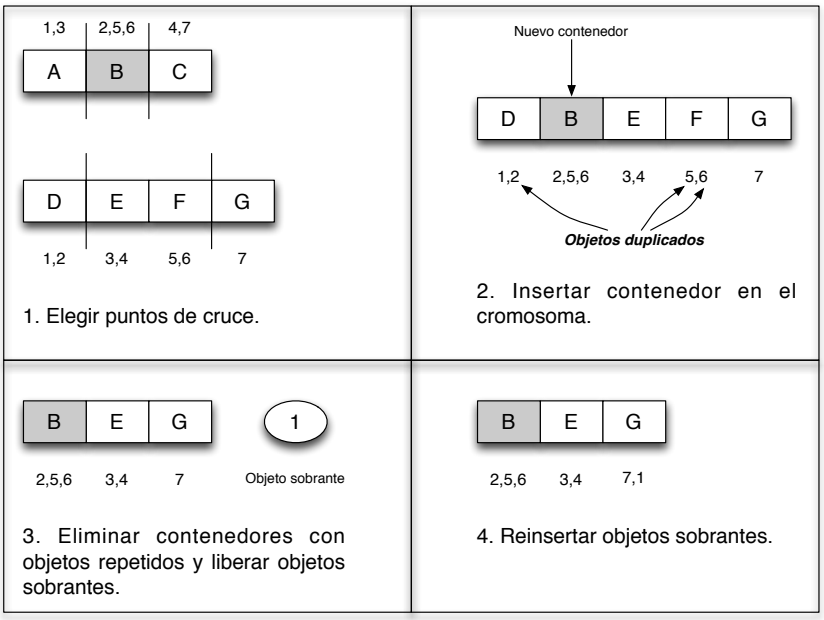

Figura 4.6: Cruce de soluciones tomando una representación basada en grupos

El procedimiento se describe a continuación:

1. Seleccionar dos secciones de cruce de forma aleatoria en cada cromosoma.
2. Generar un nuevo hijo insertando los contenedores de la sección de cruce del primer padre en el sitio de cruce inicial del segundo padre.
3. Detectar los objetos duplicados que se tengan en los contenedores originales del segundo padre.
4. Eliminar los contenedores originales con objetos duplicados y liberar los objetos sobrantes.

5. Reinsertar los objetos sobrantes en los contenedores del nuevo hijo utilizando alguna de las heurísticas expuestas en la Sección 4.2.

6. Aplicar nuevamente los pasos 2 al 5 intercambiando el papel de los cromosomas padres.

4.5 Mutación

La mutación de cromosomas debe actuar sobre contenedores y no con objetos individuales. Aquí tomaremos una mutación donde cada cromosoma mutará con una probabilidad dada. En caso de mutación, se seleccionará un contenedor al azar, se eliminará del cromosoma y los objetos que lo conforman se reinsertarán a él utilizando la técnica MAD. Este proceso se ejemplifica en la Fig. 4.7.

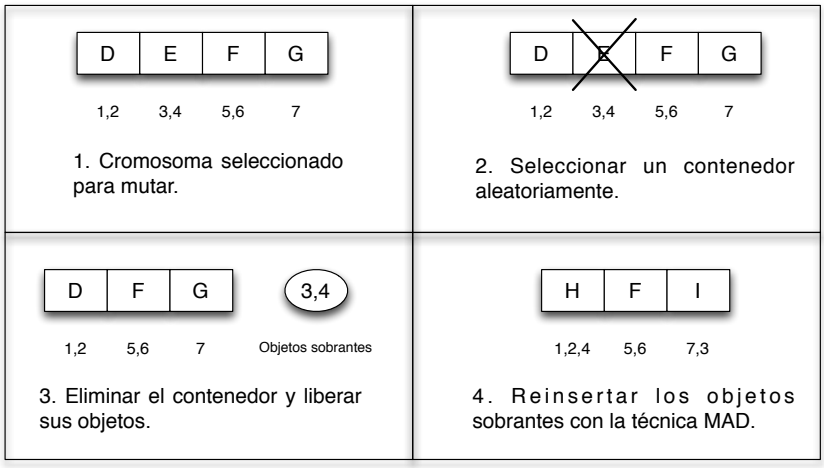

Figura 4.7: Mutación de una solución basada en grupos

Es posible que la mutación de un cromosoma vuelva a generar el mismo cromosoma, esto será mas probable mientras se tenga una mejor solución para el problema, por lo que la aplicación de este operador será fundamental para mejorar las soluciones en los pasos iniciales del algoritmo genético.

4.6 Función de Ajuste

La forma más rápida de evaluar una solución es contando el número de contenedores utilizados por ésta. Falkenaur [11] define la siguiente función de ajuste para calificar el rendimiento de una solución.

$$f = \frac{\sum_{i=1}^{N}(F_i/C)^k}{N} \tag{4.6}$$

donde N es el número de contenedores de la solución, F_i es la suma de tamaños de los objetos en el contenedor i, C es la capacidad del empaque, y k es una constante de ponderación ($k > 1$). La constante k favorece a los contenedores completamente llenos (dejándolos con valor unitario) y califica mal a contenedores llenados de forma parcial.

Esta función es preferible a buscar simplemente el mínimo de contenedores, ya que se evalúa también que tan llenos están los contenedores. Así las mejores soluciones serán aquellas que tengan contenedores mejor ocupados en comparación con otras que aunque tengan una gran cantidad de contenedores llenos, posean también contenedores poco aprovechados.

4.7 Implementación del Algoritmo Genético en Matlab®

En esta sección se explicará la implementación de los conceptos descritos anteriormente en ®, siguiendo la estructura descrita en el diagrama de flujo de la Fig. 4.8.

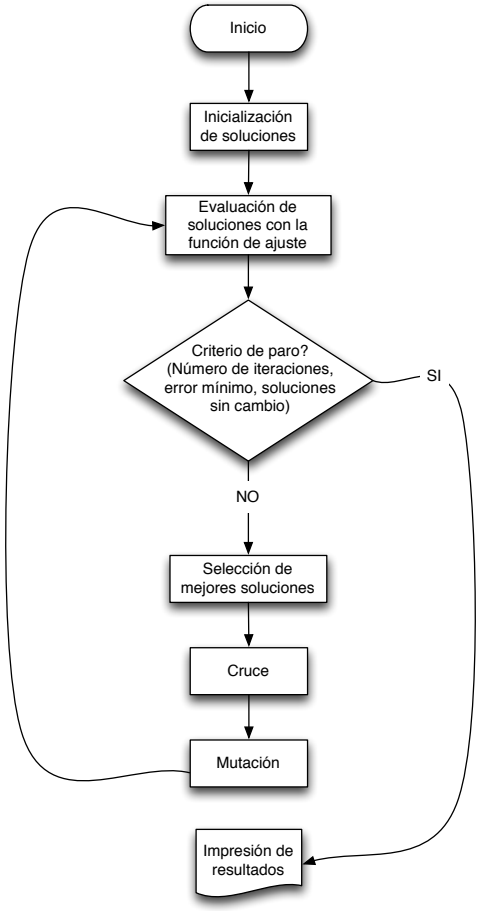

Figura 4.8: Diagrama de flujo a seguir para la implementación del algoritmo genético en Matlab®

Los parámetros de entrada del algoritmo serán:

- Arreglo *pesos* que guarda los pesos o dimensiones de los objetos a empacar.
- Escalar *capacidad* que indica la capacidad máxima de cada contenedor.
- Escalar *numind* que indica el número de individuos a manejar en el algoritmo genético.
- Escalar *numiter* que indica el número de iteraciones que realizará el algoritmo genético.

Para manejar cada individuo en Matlab®, se tomará una matriz en donde los renglones representan cada contenedor y las columnas tendrán el índice de los objetos en cada contenedor. Las dos últimas columnas tendrán el número de objetos en el contenedor, y el peso total que ocupan los objetos en el mismo (aclarando

nuevamente, *peso* indica la variable que restringe la capacidad de cada contenedor, ya sea peso, tamaño, dimensión o costo).

En las siguientes funciones, los comentarios aparecen sin acentos dado que no en todas las plataformas el editor de Matlab® los maneja de forma adecuada.

4.7.1 Inicialización de Soluciones

La siguiente función genera `numind` individuos donde cada uno representa una solución aleatoria al problema. Otros parámetros de la función son el número de objetos a empacar (`numobj`), un vector renglón con los pesos de los objetos y la capacidad máxima que puede tener cada contenedor.

La función primero genera como un vector de ceros a la población, la cual es un arreglo tridimensional, donde cada matriz es un individuo. También se declara un vector renglón que guardará el número de contenedores utilizados por cada individuo. Note que este vector tiene un número de columnas igual al doble de individuos a generar. Este espacio adicional será utilizado posteriormente por el algoritmo genético.

Para cada individuo se produce un orden aleatorio de pesos, y se van inicializando contenedores conforme se van llenando con el orden generado. Al final, la función regresa dos arreglos, la población inicial y el vector que contiene el número de contenedores utilizados en cada individuo.

```
1  function [poblacion numbins]=inicializarPoblacion(numind,numobj,pesos,capacidad)
2  %Inicializa poblacion de soluciones iniciales
3
4  %numind -> Numero de individuos a generar
5  %numobj -> Numero de objetos a empacar
6  %pesos -> Vector renglon con los pesos de cada objeto
7  %capacidad -> Capacidad de cada contenedor
8  %poblacion -> Arreglo de matrices, cada matriz guarda un individuo
9
10 %Inicializar poblacion inicial
11 poblacion=zeros(numobj,numobj+2,numind);
12 %Inicializar numero de contenedores usados
13 numbins=zeros(1,numind*2);
14 %Ciclo para generar poblacion inicial aleatoria
15 for i1=1:numind
16     %Permutacion para acomodar objetos en contenedores
17     orden=randperm(numobj);
18     %Inicializar contenedores genericos y arreglo de pesos
19     %La penultima columna dice cuantos objetos tiene el contenedor
20     %La ultima columna dice el peso contenido en el contenedor
21     contenedor=zeros(numobj,numobj+2);
22     %Indices para elegir contenedor y donde poner el objeto
23     indBin=1;
24     indObj=1;
25     for i2=1:numobj
26         %Checa si el objeto ya no cabe en el contenedor
27         if contenedor(indBin,numobj+2)+pesos(1,orden(1,i2)) > capacidad
28             %Pasar al siguiente contenedor y empezar a llenarlo desde el inicio
29             indBin=indBin+1;
30             indObj=1;
31         end
32         contenedor(indBin,numobj+2)=contenedor(indBin,numobj+2)+pesos(1,orden(1,i2));
33         contenedor(indBin,numobj+1)=contenedor(indBin,numobj+1)+1;
34         contenedor(indBin,indObj)=orden(1,i2);
35         indObj=indObj+1;
36     end
37     %Actualiza numero de contenedores usados
38     numbins(1,i1)=indBin;
39     %Agrega individuo a la poblacion
40     poblacion(:,:,i1)=contenedor;
41 end
```

4.7.2 Evaluación de Soluciones con la Función de Ajuste

La siguiente función toma la Ec. 4.6 para aplicarla sobre todos los individuos de la población de soluciones para el problema de empaque de contenedores. Esta función regresa un vector renglón donde cada elemento es la calificación asociada al individuo correspondiente.

```
function [calif]=calificarSolucion(poblacion,capacidad,factor,numbins,numobj)
%Funcion de ajuste para el problema de empaque
%Se toma de la Ec. 1.6

%poblacion -> Arreglo tridimensional donde cada individuo es una matriz
%                    La penultima columna dice cuantos objetos tiene el empaque
%                    La ultima columna dice el peso del empaque
%capacidad -> Capacidad maxima de cada contenedor
%factor -> Factor de amplitud para la funcion de ajuste
%numbins -> Numero de empaques usados en la cada solucion
%numobj -> Numero de objetos

%calif -> Arreglo que regresa la funcion con las calificaciones de cada
%            individuo

%Tamano de la poblacion
aux=size(poblacion,3);
%Inicializar calificacion de cada individuo
calif=zeros(1,aux);
%Ciclo para calcular funcion de ajuste
for j=1:aux
    calif(1,j)=(sum((poblacion(1:numbins(1,j),numobj+2,j)./capacidad).^factor))/numbins(1,j);
end
```

4.7.3 Selección de Mejores Soluciones

El algoritmo genético funcionará de la siguiente manera. Cada población se depurará haciendo un torneo entre sus individuos. Con esta población depurada, se generará otra del mismo tamaño por medio de las operaciones de cruce y mutación. Después de este proceso, ambas poblaciones quedarán unidas y sus individuos (padres e hijos) se ordenarán según su calificación siguiendo la función de ajuste. Una vez ordenadas, se desechará la peor mitad y solo nos quedaremos con la mitad de los individuos; es decir, con el número original de soluciones.

Esta selección en dos pasos se hace por las siguientes razones:

- Se hace un torneo con las soluciones originales para que las mejores tengan la probabilidad de transmitir su información un mayor número de veces, pero sin acaparar la generación de nuevos hijos.
- Se duplica el número de individuos para tener una mayor diversidad de soluciones.
- Se eligen solo a los mejores individuos que pertenecen tanto a la población anterior como a la nueva, con el fin de no perder soluciones buenas.

```
function [nuevapob nuevabins]=torneo( numind,numobj,poblacion,calif,numbins)
%Torneo, el mejor pasara dos veces y el peor no sera tomado en cuenta

%numind -> Numero de individuos en la poblacion
%numobj -> Numero de objetos
%poblacion -> Poblacion de individuos
%calif -> Vector renglon con calificaciones de los individuos
%numbins -> Numero de mpaques en cada solucion
```

```
9
10 %Hacer listas de encuentros, en dos partes aleatorias
11 lista=zeros(numind,2);
12 %Llenar primera parte del arreglo reordenando una permutacion de
13 %individuos en una matriz de dos columnas
14 lista(1:numind/2,:)=reshape(randperm(numind),numind/2,2);
15 %Segunda parte
16 lista((numind/2)+1:numind,:)=reshape(randperm(numind),numind/2,2);
17 %Inicializar poblacion nueva, tendra el doble que la poblacion inicial
18 nuevapob=zeros(numobj,numobj+2,2*numind);
19 %Nuevo numero de empaques por solucion ganadora despues del torneo
20 nuevabins=zeros(1,numind*2);
21 %Hacer torneo y dejar nuevapob con mejores
22 for i1=1:numind
23     if(calif(1,lista(i1,1))>calif(1,lista(i1,2)))
24         aux=lista(i1,1);
25         else
26             aux=lista(i1,2);
27         end
28         nuevapob(:,:,i1)=poblacion(:,:,aux);
29         nuevabins(1,i1)=numbins(1,aux);
30     end
31 end
32
```

4.7.4 Cruce de Soluciones

La siguiente función realiza el cruce de dos soluciones siguiendo el proceso descrito en la Fig. 4.6. La función recibe una población con el doble de tamaño que la población inicial, así que se tomará la mitad ya inicializada con soluciones para crear otra mitad con nuevos hijos. Esto se realiza en un ciclo en el cual por cada iteración se generarán dos nuevos hijos (línea 22).

Primero se crea una lista aleatoria de cruces entre todos los padres de la población. Una vez hecha la lista, para cada pareja de padres se calculan dos lugares de cruce por padre, cuidando de no tomar una solución completa en este proceso (líneas 24 a la 35).

Una vez seleccionados aleatoriamente los lugares de cruce, se inicializan dos individuos, uno temporal que tendrá el producto del cruce y otro definitivo el cual quitará del temporal los empaques y objetos repetidos. Primero se elige cual de los dos padres será el inicial o el final dependiendo del valor de la variable del ciclo. Se toma la parte izquierda del segundo padre que está fuera de la sección de cruce, se insertan los empaques en la sección de cruce del primer padre y por último se añaden los empaques del segundo padre que están a la derecha de su sección de cruce para formar al hijo temporal (líneas 38 a la 61).

Después tomando el hijo temporal, se crean dos arreglos, uno con empaques completos no usados en las sección de cruce y otro con objetos que quedaron sueltos, ya que pertenecían a empaques parcialmente contenidos en la sección de cruce. Estos arreglos definirán al hijo definitivo, tomando los empaques completos del primer arreglo y la sección de cruce. Se guarda además el número de empaques hasta ahora utilizados en el hijo definitivo (líneas 67 a la 124).

Como siguiente paso, se toma el vector de objetos sueltos y se acomodan en los empaques del hijo definitivo utilizando la heurística MAD (líneas 127 a la 151).

Por último, los objetos sueltos que no se pudieron acomodar se integran en empaques nuevos, tantos como sea necesarios utilizando la heurística PA, y actualizando el número de empaques utilizados en el nuevo individuo (líneas 154 a la 175).

Una vez que se tiene completo al hijo definitivo, se agrega en la segunda mitad de la población, se actualiza el número de individuos y el número de empaques utilizados por la solución en los vectores correspondientes (líneas 177 a la 179), con esto termina el ciclo de la función de cruce.

Al final, la función devuelve la población completa y el arreglo con el número de empaques utilizados por cada individuo (líneas 184 y 185).

```
function [pob bins]=cruce(numind,nuevabins,nuevapob,pesos,capacidad)
%Cruce de contenedores

%numind  -> numero de individuos
%nuevabins -> numero de empaques por solucion
%nuevapob -> poblacion nueva, tendra el doble que la poblacion inicial
%pesos -> vector renglon con pesos de los objetos
%capacidad -> capacidad maxima de cada empaque

%pob -> poblacion final despues de cruce
%bins -> vector renglon con empaques usados por cada individuo despues de
%            cruce

%Numero de objetos
numobj=size(pesos,2);
%Hacer lista de cruces, primera aleatoria de 1 hasta numpob/2
%y segunda aleatoria de numpob/2+1 hasta numpob
lista=reshape(randperm(numind),numind/2,2);
%Contador para agregar hijos a nuevapob
conthijo=numind+1;
%Ciclo para cruce
for i1=1:numind/2
    %Ciclo para calcular seccion de cruce de 2 hijos
    liminf=zeros(1,2);
    limsup=zeros(1,2);
    for i2=1:2
        %Calcular lugares de cruce para cada papa
        %randi solo acepta argumentos positivos
        liminf(i2)=randi(nuevabins(1,lista(i1,i2)));
        limsup(i2)=randi(nuevabins(1,lista(i1,i2))-(liminf(i2)-1))+liminf(i2)-1;
        %No tomar todos los empaques
        if (liminf(i2)==1) && (limsup(i2)==nuevabins(1,lista(i1,(i2))))
            limsup(i2)=limsup(i2)-1;
        end
    end
    %Hijos
    %Ciclo para 2 hijos
    for i2=1:2
        %Inicializar hijo temporal y final
        hijot=zeros(numobj,numobj+2);
        hijo=zeros(numobj,numobj+2);
        %Seleccionar padre 1 y 2
        pini=mod(i2+1,2)+1;
        pfin=mod(i2,2)+1;
        %Formar hijo temp, tomar primera parte del padre fin
        for i3=1:liminf(pfin)-1
            hijot(i3,:)=nuevapob(i3,:,lista(i1,pfin));
        end
        %Guardar indice donde poner nuevo elemento
        auxnb=liminf(pfin);
        %Formar hijo temp, tomar seccion de cruce del padre ini e
        %insertarlo en el hijo
        for i3=liminf(pini):limsup(pini)
            hijot(auxnb,:)=nuevapob(i3,:,lista(i1,pini));
            auxnb=auxnb+1;
        end
        %Formar hijo temp, tomar segunda parte del padre fin
        for i3=liminf(pfin):nuevabins(1,lista(i1,pfin))
            hijot(auxnb,:)=nuevapob(i3,:,lista(i1,pfin));
            auxnb=auxnb+1;
        end
        %Checar hijo temp, eliminar bins ya contenidos y guardar
        %objetos que queden sueltos
```

```
64          %Se guarda una lista de bins completos, los incompletos se desarman
65          %y se guardan objetos en otra lista
66          %Hacer hijo final, numero de bins del hijo
67          numbinsaux=0;
68          %Matriz auxiliar con los empaques del padre inicial
69          Maux=hijot(liminf(pfin):liminf(pfin)+(limsup(pini)-liminf(pini)),:);
70          %Vector que guarda empaques no utilizados en la seccion del padre ini
71          vecbins=[];
72          %Vector que guarda objetos sueltos
73          vecobj=[];
74          %Contador para saber cuantos bins son validos de la primera
75          %parte de padre fin
76          contprim=0;
77          %Ciclo para tomar las dos partes de padre fin en hijo temp
78          for i3=1:2
79              %Elegir que indices tomar (parte inicial o final)
80              if i3==1
81                  auxini=1;
82                  auxfin=liminf(pfin)-1;
83              else
84                  auxini=liminf(pfin)+size(Maux,1);
85                  auxfin=auxnb-1;
86              end
87              %Tomar empaques del padre fin
88              for i4=auxini:auxfin
89                  %Empaque
90                  raux=hijot(i4,:);
91                  %Interseccion de empaque con los del padre ini
92                  aux1=[];
93                  for i5=1:size(Maux,1)
94                      aux1=[aux1 intersect(raux(1:raux(numobj+1)),...
95                          Maux(i5,1:Maux(i5,numobj+1)))];
96                  end
97                  %Interseccion vacia, se conserva el empaque
98                  if isempty(aux1)
99                      vecbins=[vecbins i4];
100                     %Contar si es de la primera parte
101                     if i3==1
102                         contprim=contprim+1;
103                     end
104                 %Interseccion no vacia, checar si se usaron solo algunos
105                 %objetos y los que no se usaron guardarlos
106                 else
107                     %Objetos no usados
108                     aux2=setdiff(raux(1:raux(numobj+1)),aux1);
109                     %Guardarlos
110                     vecobj=[vecobj aux2];
111                 end
112             end
113         end
114         %Ya tenemos los empaques sin usar en vecbins, agregarlos en el hijo final
115         %Actualizar numero de bins usados en el hijo final
116         numbinsaux=size(vecbins,2);
117         %Poner empaques no usados por la seccion del padre ini primera parte
118         hijo(1:contprim,:)=hijot(vecbins(1:contprim),:);
119         %Agregar al hijo final la seccion del padre ini
120         hijo(contprim+1:contprim+size(Maux,1),:)=Maux;
121         %Poner empaques no usados por la seccion del padre ini segunda parte
122         hijo(contprim+size(Maux,1)+1:size(Maux,1)+size(vecbins,2),:)=...
123             hijot(vecbins(contprim+1:numbinsaux),:);
124         numbinsaux=numbinsaux+size(Maux,1);
125         %Quedan los objetos sueltos en vecobj, agregarlos a los empaques existentes con MAD
126         %Orden de pesos de objetos sueltos, de mayor a menor
127         [listap indp]=sort(pesos(vecobj),2,'descend');
```

```
128         %Poner objetos sobrantes en los empaques
129         %Objetos que se pudieron acomodar
130         objacm=[];
131         for i3=1:size(listap,2)
132             %Orden de capacidades disponibles, de menor a mayor
133             [listac indc]=sort((hijo(1:numbinsaux,numobj+2))',2,'ascend');
134             for i4=1:size(listac,2)
135                 %El objeto cabe en el empaque
136                 if (listac(i4)+listap(i3)) < capacidad
137                     %Acomodar objeto
138                     hijo(indc(i4),hijo(indc(i4),numobj+1)+1)=vecobj(indp(i3));
139                     %Ajustar numero de objetos
140                     hijo(indc(i4),numobj+1)=hijo(indc(i4),numobj+1)+1;
141                     %Ajustar capacidad ocupada por objetos
142                     hijo(indc(i4),numobj+2)=hijo(indc(i4),numobj+2)+listap(i3);
143                     %Poner objeto como ya acomodado
144                     objacm=[objacm vecobj(indp(i3))];
145                     %Ajustar lista de capacidades disponibles en el empaque
146                     listac(i5)=listac(i5)+listap(i3);
147                     %Romper ciclo
148                     break
149                 end
150             end
151         end
152         %Hacer nuevos empaques con objetos que no se pudieron acomodar
153         %Sacar objetos que no se acomodaron
154         vecobj=setdiff(vecobj,objacm);
155         %Acomodar objetos en nuevos empaques
156         if size(vecobj,2)~=0
157             %Indice del objeto a acomodar en el empaque
158             indbin=1;
159             numbinsaux=numbinsaux+1;
160             for i3=1:size(vecobj,2)
161                 %El objeto no cabe en el empaque, definir nuevo
162                 if (hijo(numbinsaux,numobj+2)+pesos(vecobj(i3)))>capacidad
163                     numbinsaux=numbinsaux+1;
164                     indbin=1;
165                 end
166                 %Poner objeto
167                 hijo(numbinsaux,indbin)=vecobj(i3);
168                 %Actualizar numero de objetos en el empaque
169                 hijo(numbinsaux,numobj+1)=indbin;
170                 %Actualizar peso del empaque
171                 hijo(numbinsaux,numobj+2)=hijo(numbinsaux,numobj+2)+pesos(vecobj(i3));
172                 %Actualizar indice de objetos en el empaque
173                 indbin=indbin+1;
174             end
175         end
176         %Se tiene el hijo completo final, agregar
177         nuevapob(:,:,conthijo)=hijo;
178         nuevabins(conthijo)=numbinsaux;
179         conthijo=conthijo+1;
180     end
181 end
182 
183 %Regresar poblacion nueva y empaques usados
184 pob=nuevapob;
185 bins=nuevabins;
```

4.7.5 Mutación de Soluciones

La siguiente función realiza la mutación de cada solución con una probabilidad *probmut*. La función recibe la población resultante de la función cruce; es decir, una población con el doble de individuos especificados inicialmente. Primero se inicializa un hijo *mutado* compuesto por una matriz de ceros donde los renglones representan cada contenedor y contienen los índices de los objetos en cada contenedor. Las dos últimas columnas tendrán el número de objetos en el contenedor, y el peso total que ocupan los objetos (línea 21).

En el primer ciclo se revisa si un individuo se va a mutar o no generando un número aleatorio, en caso afirmativo el individuo se copia en la matriz *mutado*. Una vez copiado el individuo, se elige un empaque al azar, este empaque se saca del arreglo en forma de un vector y se actualiza el arreglo así como el número de empaques que actualmente tiene (líneas 29 a la 39).

Como siguiente paso, se toma el vector de objetos y se acomodan en los empaques del individuo *mutado* utilizando la heurística MAD (líneas 42 a la 66).

Los objetos que se pudieron acomodar se enlistan en el vector *objacm*, se calcula la diferencia con respecto al vector completo de objetos que se tomaron del individuo. Los objetos sobrantes (si existen) se integran en empaques nuevos, tantos como sea necesarios utilizando la heurística PA, y actualizando el número de empaques utilizados en el individuo *mutado* (líneas 69 a la 90).

Una vez acabado el proceso de mutación, el hijo *mutado* y el número de empaques actualizado que se usan en el mismo se reintegran al arreglo tridimensional que guarda la población y al vector con el número de empaques respectivamente (líneas 92 y 93). Al final, la función devuelve la población completa y el arreglo con el número de empaques utilizados por cada individuo (líneas 97 y 98).

```
1  function [pob bins]=mutacion(numind,probmut,nuevabins,nuevapob,pesos,capacidad)
2  %Mutacion de contenedores
3
4  %numind  -> numero de individuos
5  %probmut  -> probabilidad de mutacion
6  %nuevabins -> numero de empaques por solucion
7  %nuevapob -> poblacion nueva, tendra el doble que la poblacion inicial
8  %pesos -> vector renglon con pesos de los objetos
9  %capacidad -> capacidad maxima de cada empaque
10
11 %pob -> poblacion final despues de mutacion
12 %bins -> vector renglon con empaques usados por cada individuo despues de
13 %            mutacion
14
15 %Numero de objetos
16 numobj=size(pesos,2);
17 %Mutacion
18 %Se deshace un empaque con cierta probabilidad, sus elementos se
19 %reintegran con la heuristica MAD
20 %Inicializar hijo mutado
21 mutado=zeros(numobj,numobj+2);
22 %Ciclo para ver si hay mutacion
23 for i1=1:numind*2
24     %Checar probabilidad
25     if(rand <= probmut)
26         %Tomar individuo a mutar
27         mutado=nuevapob(:,:,i1);
28         %Tomar un empaque al azar
29         binale=randi(nuevabins(i1));
30         %Tomar numero de objetos en el empaque aleatorio
31         numaux1=mutado(binale,numobj+1);
32         %Tomar objetos del empaque
33         vecobj=mutado(binale,1:numaux1);
34         %Deshacer empaque, encimar los renglones siguientes en la matriz y
35         %actualizar numero de empaques usados en el individuo
36         numaux2=nuevabins(i1);
37         mutado(binale:numaux2,:)=mutado(binale+1:numaux2+1,:);
```

```
        nuevabins(i1)=nuevabins(i1)-1;
        numaux2=nuevabins(i1);
        %Insertar los objetos sueltos en los empaques siguiendo MAD
        %Orden de pesos de objetos sueltos, de mayor a menor
        [listap indp]=sort(pesos(vecobj),2,'descend');
        %Poner objetos sobrantes en los empaques
        %Objetos que se pudieron acomodar
        objacm=[];
        for i2=1:size(listap,2)
        %Orden de capacidades disponibles, de menor a mayor
        [listac indc]=sort((mutado(1:numaux2,numobj+2))',2,'ascend');
            for i3=1:size(listac,2)
                %El objeto cabe en el empaque
                if (listac(i3)+listap(i2)) < capacidad
                    %Acomodar objeto
                    mutado(indc(i3),mutado(indc(i3),numobj+1)+1)=vecobj(indp(i2));
                    %Ajustar numero de objetos
                    mutado(indc(i3),numobj+1)=mutado(indc(i3),numobj+1)+1;
                    %Ajustar capacidad ocupada por objetos
                    mutado(indc(i3),numobj+2)=mutado(indc(i3),numobj+2)+listap(i2);
                    %Poner objeto como ya acomodado
                    objacm=[objacm vecobj(indp(i2))];
                    %Ajustar lista de capacidades disponibles en el empaque
                    listac(i3)=listac(i3)+listap(i2);
                    %Romper ciclo
                    break
                end
            end
        end
        %Hacer nuevos empaques con objetos que no se pudieron acomodar
        %Sacar objetos que no se acomodaron
        vecobj=setdiff(vecobj,objacm);
        %Acomodar objetos en nuevos empaques
        if size(vecobj,2)~=0
            %Indice del objeto a acomodar en el empaque
            indbin=1;
            numaux2=numaux2+1;
            for i2=1:size(vecobj,2)
                %El objeto no cabe en el empaque, definir nuevo
                if (mutado(numaux2,numobj+2)+pesos(vecobj(i2)))>capacidad
                    numaux2=numaux2+1;
                    indbin=1;
                end
                %Poner objeto
                mutado(numaux2,indbin)=vecobj(i2);
                %Actualizar numero de objetos en el empaque
                mutado(numaux2,numobj+1)=indbin;
                %Actualizar peso del empaque
                mutado(numaux2,numobj+2)=mutado(numaux2,numobj+2)+pesos(vecobj(i2));
                %Actualizar indice de objetos en el empaque
                indbin=indbin+1;
            end
        end
        %Se tiene el hijo completo final, agregar
        nuevapob(:,:,i1)=mutado;
        nuevabins(i1)=numaux2;
    end
end
%Regresar poblacion nueva y empaques usados
pob=nuevapob;
bins=nuevabins;
```

4.7.6 Implementación Integrada

La siguiente función muestra la integración de todas las funciones anteriores. Primero se recibe como argumentos el número de individuos, número de iteraciones y el nombre del archivo que contiene los pesos, la capacidad del empaque y la solución óptima. Estos archivos de problemas son ampliamente usados para probar nuevas metodologías para resolver el problema de empaque en contenedores, se obtuvieron de la dirección electrónica http://www.wiwi.uni-jena.de/Entscheidung/binpp/bin1dat.htm; y contemplan cuatro parámetros:

n	Número de objetos.
w_j	Peso (tamaño) de los objetos para $j = 1, ..., n$.
c	Capacidad del empaque.
m	Solución optima (mínimo número de empaques para el problema propuesto).

El nombre de cada archivo de texto tiene la estructura "NxCyWz_v" donde x, y, z,v definen los siguientes parámetros:

Variable	Valor
x	1 para $n = 50$, 2 para $n = 100$, 3 para $n = 200$, 4 para $n = 500$.
y	1 para $c = 100$, 2 para $c = 120$, 3 para $c = 150$.
z	1 para $w_j \in [1, \ldots, 100]$, 2 para $w_j \in [20, \ldots, 100]$, 4 para $w_j \in [30, \ldots, 100]$.
v	A...T para 20 ejemplos de cada clase.

Así el archivo N1C1W1_A indica que el archivo es la primera instancia de un problema referente a una muestra de 50 objetos, con empaques de capacidad 100 y el peso de cada objeto se encuentra en el intervalo $[1, \ldots, 100]$.

Primero se toman los datos del archivo y se guardan en variables dentro del programa, como son el número de objetos, la capacidad de los empaques y la lista de pesos de los objetos. También se inicializan la probabilidad de mutación y el factor de ponderación para la función de ajuste (líneas 23 a la 32).

A continuación, se genera una población inicial aleatoria (línea 35) y se empieza el ciclo del algoritmo genético (línea 37). Dentro del ciclo, se califican las soluciones existentes (línea 39), y a partir de la segunda iteración se ordenan y según su calificación y se desecha la peor mitad (líneas 43 a la 51).

Si se llega al límite de iteraciones, se ordenan las calificaciones y el número de empaques utilizados por cada solución para obtener las características de la mejor solución en el primer elemento de estos arreglos (líneas 54 a la 60).

La última parte del ciclo realiza el torneo, cruce y mutación de las soluciones (líneas 65 a la 69). Al terminar el ciclo, la función regresa la población ya depurada con sus calificaciones y números de empaques utilizados por cada solución.

```
1  function [poblacion calif numero] = geneticoBinPacking(numind,numiter,nombre)
2  %Soluciona con un genetico Bin Packing
3
4  %[pob cal num]=geneticoBinPacking(60,50,'Benchmark/bin1data/N1C1W1_A.BPP')
5
6  %numind  -> numero de individuos
7  %numiter -> numero de iteraciones
8  %nombre -> nombre del archivo con pesos, capacidad de empaques y solucion
9  %                optima
10 %Formato del nombre del archivo NxCyMz indica que:
11 %x = 1 para n=50, 2 para n=100, 3 para n=200, 4 para n=500. \\
12 %y = 1 para c=100, 2 para c=120, 3 para c=150. \\
13 %z = 1 para pesos entre [1,...,100], 2 para pesos entre [20,...,100],
       4 para pesos entre [30,...,100]. \\
14 %v = A...T para 20 ejemplos de cada clase. \\
```

```
15
16 %poblacion -> poblacion de soluciones
17 %calif -> vector con calificaciones para cada solucion
18 %numero -> vector con el numero de empaques usados por cada solucion
19 %sol -> solucion optima conocida
20
21 %Toma los datos del bin packing de un archivo
22 %Numero de objetos
23 numobj = dlmread(nombre,'',[0 0 0 0]);
24 %Capacidad de contenedores
25 capacidad = dlmread(nombre,'',[1 0 1 0]);
26 %Arreglo de pesos
27 pesos = dlmread(nombre,'',[2 0 numobj+1 0]);
28 pesos=pesos';
29 %Probabilidad de mutacion, 1/numpob
30 probmut=1/numind;
31 %Parametro para ponderar funcion de ajuste
32 factor=2.0;
33
34 %Generar poblacion inicial
35 [poblacion numero]=inicializarPoblacion(numind,numobj,pesos,capacidad);
36 %Ciclo del algoritmo genetico
37 for i=1:numiter+1
38     %Calificar soluciones
39     calif=calificarSolucion(poblacion,capacidad,factor,numero,numobj);
40     %Tamano de la poblacion, al principio es n y despues 2n
41     aux=size(poblacion,3);
42     %A partir de la segunda iteracion, se quita la peor mitad
43     if (i>1)
44         %Ordenar calificaciones, ind guarda orden
45         [orden ind]=sort(calif,2,'descend');
46         %Inicializar poblacion nueva
47         nuevapob=zeros(numobj,numobj+2,aux);
48         %Reordenar la poblacion en problacion nueva
49         nuevapob(:,:,1:numind)=poblacion(:,:,ind(1,1:numind));
50         poblacion=nuevapob;
51     end
52     %Iteracion numiter+1, tomar la mitad superior de poblacion y salir del
53     %ciclo
54     if i==numiter+1
55         %Cortar calificaciones
56         calif=orden(1:numind);
57         %Reordenar numero de bins iusados
58         numero=numero(ind(1:numind));
59         break;
60     end
61     %Imprimir iteracion
62     text=sprintf('Iteracion %3d',i);
63     disp(text);
64     %Torneo para dejar los mejores en la poblacion
65     [poblacion numero]=torneo( numind,numobj,poblacion,calif,numero);
66     %Cruce
67     [poblacion numero] = cruce(numind,numero,poblacion,pesos,capacidad);
68     %Mutacion
69     [poblacion numero] = mutacion(numind,probmut,numero,poblacion,pesos,capacidad);
70 end
```

De los datos de prueba señalados anteriormente, se tomaron las primeras dos instancias de cada combinación con $x = 1, 2$, es decir, los 36 ejemplos de la forma "NxCyWz_A" y "NxCyWz_B". Los resultados se muestran en la Tabla 4.1. La tabla muestra el resultado óptimo conocido de cada ejemplo y el obtenido por el algoritmo genético. Para estos casos, se aplicó el proceso genético con una población de 60 individuos y 60 iteraciones; la selección de estos parámetros obedeció a buscar un equilibrio entre obtener mejores soluciones

en un tiempo de ejecución corto.

Tabla 4.1: Resultados obtenidos con el algoritmo genético para el problema de empaque de contenedores utilizando los datos de prueba en http://www.wiwi.uni-jena.de/Entscheidung/binpp/bin1dat.htm

Archivo	Óptimo	Obtenido	Tiempo (Seg.)
N1C1W1_A	25	26	107
N1C1W1_B	31	31	146
N1C1W2_A	29	29	127
N1C1W2_B	30	31	144
N1C1W4_A	35	35	179
N1C1W4_B	40	40	208
N1C2W1_A	21	21	79
N1C2W1_B	26	26	109
N1C2W2_A	24	25	110
N1C2W2_B	27	28	130
N1C2W4_A	29	30	142
N1C2W4_B	32	32	154
N1C3W1_A	16	17	55
N1C3W1_B	16	16	52
N1C3W2_A	19	19	68
N1C3W2_B	20	20	77
N1C3W4_A	21	21	80
N1C3W4_B	22	23	94
N2C1W1_A	48	49	417
N2C1W1_B	49	50	423
N2C1W2_A	64	64	644
N2C1W2_B	61	62	572
N2C1W4_A	73	73	744
N2C1W4_B	71	71	703
N2C2W1_A	42	42	311
N2C2W1_B	50	50	437
N2C2W2_A	52	53	452
N2C2W2_B	56	56	534
N2C2W4_A	57	58	555
N2C2W4_B	60	60	580
N2C3W1_A	35	36	252
N2C3W1_B	35	35	232
N2C3W2_A	41	42	314
N2C3W2_B	43	44	367
N2C3W4_A	43	44	356
N2C3W4_B	45	46	402

4.8 Apuntes Finales

En la Tabla 4.1 podemos observar que de las 36 instancias, en 17 casos la solución que se obtiene no es la óptima por solo un contenedor. Por supuesto, un aumento en el número de individuos y/o de iteraciones generará mejores resultados, con el costo de un mayor tiempo de ejecución.

Cabe hacer notar que la función de ajuste en la Ec. 4.6 califica mejor soluciones con contenedores que estén lo más ocupados posibles, y penaliza aquellas en donde la mayoría puedan estar llenas pero algunas

tengan un porcentaje alto de vacío, lo cual en algunos casos explica el porqué no se alcanza una solución óptima con respecto a minimizar el número de contenedores.

Sin embargo, consideramos que esto no es un problema ya que un buen método numérico debe encontrar buenas soluciones para cualquier instancia de un problema dado en un tiempo razonable, y no solo soluciones óptimas para unos cuantos problemas a expensas de un mayor tiempo de ejecución.

Capítulo 5

Conclusiones

5.1 Resultados en la Solución del Problema de Empacado en Contenedores

Los resultados obtenidos en el problema de empacado en contenedores demuestran experimentalmente que problemas combinatorios discretos que resultan intratables de resolver con una búsqueda exhaustiva o con técnicas clásicas, pueden ser tratados de manera satisfactoria con algoritmos evolutivos que sean modificados de la manera apropiada.

Así, en la mayoría de los casos, el proceso planteado en este libro logra alcanzar o estar muy cerca de la solución óptima. Una ventaja de este proceso es que solo necesitamos conocer la descripción del problema y una forma de calificar nuestras soluciones para poder implementar el algoritmo genético. No es necesario tener que conocer a profundidad las propiedades del espacio de soluciones como en el caso de métodos matemáticos clásicos.

De hecho, esta es una propiedad en la aplicación de dicho tipo de algoritmos, si el espacio de soluciones tiene propiedades muy complejas o difíciles de modelar, es complicado adaptar técnicas analíticas para encontrar una solución óptima, y es ahí donde la utilización de los algoritmos genéticos ha encontrado un área de desarrollo importante.

5.2 Ventajas de Matlab®

Como se pudo constatar en este libro, Matlab® ofrece un ambiente amigable para el desarrollo de aplicaciones de cálculo numérico, cuya implementación puede realizarse sin contar con amplios conocimientos en lenguajes de programación. Es necesario contar con conocimientos básicos acerca de los operadores aritméticos, relacionales y lógicos. Además, para llevar el control en el flujo del código, al igual que en los lenguajes de programación, en Matlab® existen estructuras de control que nos ayudan a definir la secuencia del flujo del programa.

Por otro lado, es de suma importancia que se identifiquen las operaciones y funciones predefinidas que ofrece Matlab®, y evitar la implementación de éstas por parte del lector. Tal es el caso de la multiplicación de matrices, en un lenguaje de programación como C, C++, Java o Delphi, esta operación matricial se tiene que implementar dentro del mismo código, sin embargo en Matlab® esta operación se realiza como si se estuvieran multiplicando dos números cualesquiera, sin necesidad de implementar el método de la multiplicación de matrices.

Con estos conocimientos básicos, es posible desarrollar algoritmos en este software, como el caso de los códigos que se programaron de algoritmos genéticos en el capítulo 2, además de la implementación del algoritmo genético para el problema del empacado en contenedores en el capítulo 3. Nos damos cuenta que

este tipo de programas de optimización pueden ser implementados de forma sencilla en Matlab®, y si el lector está interesado en continuar con la implementación de más algoritmos de optimización o de otra área, puede tomar este contenido como base y seguir descubriendo el potencial que ofrece este software de cálculo numérico.

5.3 Ventajas y Limitaciones de los Algoritmos Genéticos

Como todo algoritmo de solución tiene sus pros y contras, sin embargo, la justificación para aplicar estos algoritmos a la solución de problemas en ingeniería es que han tenido éxito en aquellas aplicaciones en donde las herramientas matemáticas gradientales de optimización han fracasado. A continuación se enumeran las ventajas y desventajas de los AG.

5.3.1 Ventajas

1. Implementación computacional. Las operaciones computacionales para realizar un AG es a través de operaciones aritméticas, lógicas y de ordenamiento sencillas que son fáciles de implementar computacionalmente.

2. Los AG no necesitan información a priori del sistema que se desea optimizar. El AG genera múltiples soluciones de forma aleatoria y si algunas de ellas mejoran, entonces, son soluciones que se tomarán en cuenta para evolucionar la población.

3. El AG realiza un amplia exploración en el rango de búsqueda factible sin importar que el sistema sea de funciones discontinuas, que no tengan derivada, que sea no convexo, o no lineal.

5.3.2 Desventajas

1. Función costo. La única forma para evaluar el desempeño de las evoluciones en el AG es a través de una función de evaluación o una colección de datos, en ciertas aplicaciones no es sencillo establecer una función costo para optimizar un sistema multivariable.

2. No existen reglas para determinar el número de individuos en una población, que tipo de selección aplicar o cómo realizar la mutación.

3. Programación serie. Cuando los AG genéticos se programan en plataformas de procesamiento serie los AG no son tan rápidos en su tiempo de ejecución y por lo tanto, es difícil implementarlos en aplicaciones donde es necesario realizar ajustes en tiempo real o en línea.

4. Es probable que un individuo con un alto desempeño propague su código genético desde el arranque de la simulación, de tal forma, que se pierde diversidad en la población y el AG cae un punto óptimo local.

5. Los AG genéticos tienen una amplia gama de aplicaciones donde han logrado optimizar sistemas con éxito, pero esto no quiere decir que se puedan utilizar en cualquier aplicación y que logren un mejor desempeño que los métodos analíticos matemáticos.

5.4 Trabajo Futuro Relacionado

Además del problema del empacado en contenedores abordado en este libro, los algoritmos genéticos también pueden ser aplicados en otros problemas de optimización. Se pueden atacar problemas de optimización combinatoria como el caso del problema del conjunto de cobertura, o problema del Set Covering (SCP), el cual consiste en encontrar aquellas soluciones que minimicen el costo de cubrir un conjunto de requerimientos;

dicho de otra manera, consiste en encontrar el mínimo de subconjuntos que contengan todos los elementos de un conjunto dado, con el fin de minimizar algún valor.

Otro problema interesante es el problema de la mochila, también conocido como el problema Knapsack, en el cual se necesita llenar una mochila seleccionando algunos objetos de un conjunto de varios de ellos. Existen n objetos diferentes disponibles y cada uno tiene un peso y un costo asociados. La mochila puede aguantar un peso máximo. El objetivo es encontrar un subconjunto óptimo de objetos que minimicen el costo total sujeto a la capacidad en peso de la mochila.

Existen problemas de optimización multiobjetivo, en los que no se tiene solamente una función objetivo, sino que se tiene un conjunto de funciones que se desean optimizar. En este caso también es posible implementar algoritmos genéticos para encontrar soluciones aceptables en este tipo de problemas.

Además, los algoritmos genéticos se están aplicando para encontrar soluciones en problemas de secuenciamiento de máquinas y/o de operaciones, en donde se busca la asignación de recursos limitados a través del tiempo para realizar tareas o actividades que satisfagan ciertos criterios o restricciones.

Otro de los problemas interesantes en ingeniería, que también puede ser abordado mediante los algoritmos genéticos, es el problema de transporte. Este es un problema básico de redes, en el cual no solo se considera la determinación del patrón de transporte óptimo, sino también puede incluirse el análisis de los problemas de secuenciamiento de producción, problemas de envío y problemas de asignación.

Bibliografía

[1] T. Back, D. B. Fogel, and T. Michalewicz. *Evolutionary Computation 1, Basic Algorithms and Operators*. Institute of Physics Publishing, 2000.

[2] R. Baldick. *Applied Optimization: Formulation and Algorithms for Engineering Systems*. Cambridge University Press, 2009.

[3] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms*. Wiley-Interscience, 2006.

[4] D. P. Bertsekas. *Nonlinear Programming, 2nd Edition*. Athena Scientific, 1999.

[5] D. P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II, 4th Edition: Approximate Dynamic Programming*. Athena Scientific, 2012.

[6] D. Bertsimas. *Introduction to Linear Optimization*. (Athena Scientific Series in Optimization and Neural Computation), Athena Scientific, 1997.

[7] S. Boyd. *Convex Optimization*. Cambridge University Press, 2004.

[8] E. G. Coffman, M. R. Garey, and D. S Johnson. Approximation algorithms for bin-packing–an updated survey. In *Algorithm Design for Computer System Design*, pages 49–106. 1984.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, third edition, 2009.

[10] T. Davidovic, M. Selmic, D. Teodorovic, and D. Ramljak. Bee colony optimization for scheduling independent tasks to identical processors. *J. Heuristics*, 18:549–569, 2012.

[11] E. Falkenauer. A new representation and operators for genetic algorithms applied to grouping problems. *Evol. Comput.*, 2(2):123–144, June 1994.

[12] A. Fuchs. *Nonlinear Dynamics in Complex Systems: Theory and Applications for the Life-, Neuro- and Natural Sciences*. Springer, 2012.

[13] M. Gen and R. Cheng. *Genetic Algorithms and Engineering Optimization*. John Wiley & Sons, Inc., 2000.

[14] D.E. Goldberg. *Computer-aided gas pipeline operation using genetic algorithms and rule learning*. PhD thesis, University of Michigan, 1983. Dissertation Abstracts International, 44(19) 1374B. University Microfilms No. 8402282.

[15] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[16] V. Hemmelmayr, V. Schmid, and C. Blum. Variable neighbourhood search for the variable sized bin packing problem. *Comput. Oper. Res.*, 39:1097–1108, 2012.

[17] John H. Holland. *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. The MIT Press, 1975.

[18] Q. Kang and H. He. A novel discrete particle swarm optimization algorithm for meta-task assignment in heterogeneous computing systems. *Microprocess. Microsy.*, 35:10–17, 2011.

[19] A. H. Kashan and B Karimi. A discrete particle swarm optimization algorithm for scheduling parallel machines. *Comput. Ind. Eng.*, 56:216–223, 2009.

[20] H.-W. Kim, J.-M. Yu, J.-S. Kim, H.-H. Doh, D.-H. Lee, and S.-H. Nam. Loading algorithms for flexible manufacturing systems with partially grouped unrelated machines and additional tooling constraints. *Int. J. Adv. Manuf. Technol.*, 58:683–691, 2012.

[21] D.S. Liu, K.C. Tan, S.Y. Huang, C.K. Goh, and W.K. Ho. On solving multiobjective bin packing problems using evolutionary particle swarm optimization. *Eur. J. Oper. Res.*, 190:357–382, 2008.

[22] K. E. Parsopoulos and M. N. Vrahatis. *Particle Swarm Optimization and Intelligence: Advances and Applications*. IGI Global, 2010.

[23] A. Ruszczynski. *Nonlinear Optimization*. Princeton University Press, 2006.

[24] S. Sumathi, T. Hamsapriya, and P. Surekha. *Evolutionary Intelligence, An Introduction to Theory and Applications with Matlab*. Springer-Verlag, 2008.

[25] R. Xu, H. Chen, and X. Li. Makespan minimization on single batch-processing machine via ant colony optimization. *Comput. Oper. Res.*, 39:582–593, 2012.

[26] Y. Zhang, F. Y. L. Chin, H.-F. Ting, and X. Han. Online algorithms for 1-space bounded multi dimensional bin packing and hypercube packing. *J. Comb. Optim.*, 2012. Available with full open access.