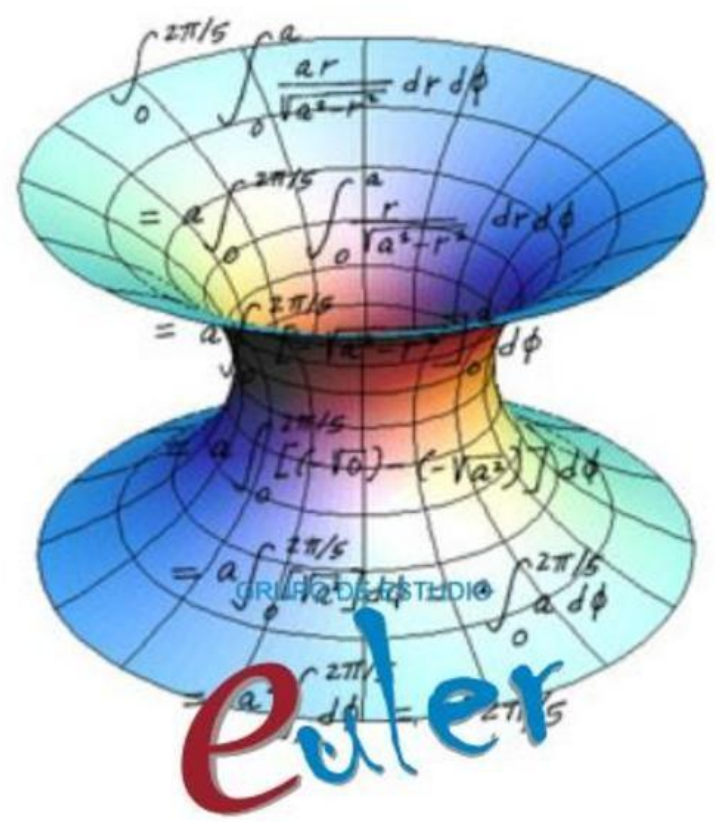
MÉTODOS NUMÉRICOS EN JAVASCRIPT



HERNAN PEÑARANDA V.

materias-hpv.comli.com

Sucre, 1/2013

1. EXPRESIONES MATEMÁTICAS

En la asignatura se estudian métodos que permiten resolver problemas matemáticos mediante la ejecución de cálculos aritméticos simples, dichos métodos se conoce con el nombre de métodos numéricos.

El que sea posible resolver problemas matemáticos complejos, con simples cálculos aritméticos, es la razón por la cual son empleados en la resolución de problemas en todos los campos de la ciencia y de la ingeniería.

Sin embargo, si bien el proceso es simple, es también moroso: se requieren cientos, miles e incluso millones de operaciones aritméticas para encontrar un resultado, aspecto que dificultó su aplicación práctica hasta la invención de las computadoras.

Con la invención de las computadoras se hizo posible automatizar el proceso de cálculo, pues como se sabe, las computadoras son capaces de realizar, de forma confiable y eficiente, miles de operaciones aritméticas en fracciones de segundo.

Actualmente, con la proliferación de computadoras y dispositivos móviles programables, prácticamente todos los problemas matemático de aplicación práctica, se resuelven con la ayuda de los métodos numéricos. Si bien los dispositivos móviles, principalmente los celulares, no son vistos como computadoras, son en realidad micro-computadoras, que en algunos casos (sobre todo los de última generación), tienen capacidades de procesamiento que compiten fácilmente con los de una computadora portátil o de escritorio.

Por todo lo anteriormente expuesto, sería inadmisible pretender enseñar la materia sin recurrir a una computadora o un dispositivo programable.

1.1. LENGUAJE DE PROGRAMACIÓN A EMPLEAR EN LA MATERIA

Existe una amplia oferta de software, tanto libre como pago, que tienen implementado un amplio conjunto de métodos numéricos, por lo que es posible resolver con los mismos la mayoría de los problemas matemáticos: integración, diferenciación, resolución de ecuaciones, etc. Dicho software cuenta con su propio lenguaje de programación, lo que permite implementar en los mismos otros métodos numéricos, o, con mayor frecuencia, programar funciones y/o resolver problemas que implican dos o más métodos numéricos.

Entre dicho software se puede mencionar a Mathematica, Matlab, Maple, Matcad, Derive y Eureka, que son pagos, y a Octave y Scilab que son libres. En todos ellos es bastante sencillo resolver problemas matemáticos simples, sin embargo, para resolver problemas más complejos, que involucran dos o más métodos y/o funciones complejas, que son los problemas que se presentan en la vida real, se deben elaborar programas, lo que implica aprender el lenguaje de programación del software elegido.

La principal desventaja, de este tipo de software, es que el lenguaje de programación es específico del mismo, y aunque algunos de ellos son compatibles entre sí (como Octave y Matlab), no son compatibles con otros lenguajes de propósito más general, por lo que resulta difícil su aplicación en otros ambientes como Internet.

Por lo anteriormente mencionado y tomando en cuenta que todos los dispositivos móviles actuales cuentan con un navegador Internet con soporte Javascript, en la materia se ha elegido ese lenguaje para el desarrollo de la misma. Sin embargo, se empleará también (principalmente para efec-

tos de comparación) Octave, el cual puede ser instalado en una computadora (www.gnu.org/software/octave) o en un dispositivo móvil con sistema Android, bajando la aplicación de google play store (https://play.google.com/store/).

1.2. BREVE INTRODUCCIÓN A JAVASCRIPT

Aunque JavaScript es un lenguaje creado con el propósito de darle funcionalidad a las páginas WEB (crear páginas WEB dinámicas), es en realidad un lenguaje de propósito general, que puede y es empleado, en diferentes campos de la actividad y conocimiento humano.

Quizá su principal ventaja es su universalidad: para hacer correr un programa Javascript sólo se requiere un navegador Internet y los navegadores Internet están disponibles en todas las computadoras y dispositivos programables actuales.

Como lenguaje soporta tanto la programación estructurada como la orientada a objetos y es lo suficientemente potente y general como para permitir la creación de aplicaciones complejas en todos los campos del conocimiento humano.

Para escribir un programa en JavaScript lo único que se requiere es un editor de textos, como el block de notas de Windows (notepad), aunque se pueden recurrir a editores más especializados como notepad++ (http://notepad-plus-plus.org) para Windows o droidedit (https://play.google.com/store/) para sistemas Android. Existen también, entornos completos de desarrollo como eclipse (www.eclipse.org/) y NetBeans (http://netbeans.org/).

Como ya se dijo, para ejecutar un programa JavaScript lo único que se requiere es un navegador Internet. Por esta razón, los programas Javascript pueden ser ejecutados en cualquier sistema operativo siempre y cuando cuente con un navegador Internet. Esto significa que un programa escrito en un celular funcionará igualmente en Windows, Linux, Free BSD, OS System (Macintosh), Solaris, Androide, Symbian y en cualquier otra computadora y/o dispositivo programable que cuente con un navegador Internet.

Además la tecnología en el campo de la informática se está enfocando cada vez más en el desarrollo de aplicaciones WEB (donde JavaScript juega un rol primordial), por lo que se prevé que en un futuro cercano todas las aplicaciones, en todos los campos de la ciencia y del conocimiento, se encuentren en ese ambiente. Se puede vislumbrar un futuro con aplicaciones corriendo en los navegadores de los dispositivos móviles, resolviendo problemas relativamente complejos en el mismo dispositivo y solicitando la solución de problemas más complejos a un servidor Internet, el mismo que estará ubicado en alguna parte del mundo, o quizá incluso fuera de este.

1.3. FUNDAMENTOS BÁSICOS DEL LENGUAJE

Javascript es un lenguaje interpretado, es decir que para ejecutar un programa, Javascript interpreta (traduce) la instrucción a lenguaje de máquina (en memoria) ejecuta dicha instrucción, devuelve el resultado de la instrucción y pasa a la siguiente instrucción, donde se repite todo el proceso. Traducir cada instrucción, cada vez que se ejecuta, consume tiempo, por lo que los lenguajes interpretados, como Javascript, son más lentos que los lenguajes compilados, como C++. En estos últimos el programa es traducido a lenguaje de máquina en su integridad y es guardado

en un archivo que luego se ejecuta (sin necesidad de traducción pues ya se encuentra en lenguaje de máquina).

En contrapartida, los lenguajes interpretados son más flexibles que los compilados, lo que por una parte hace más fácil su programación y por otra permite crear, evaluar y usar elementos que se crean mientras el programa se ejecuta (algo que es difícil de conseguir en los lenguajes compilados).

Un aspecto que se debe tomar muy en cuenta al programar en **Javascript**, es que **diferencia entre mayúsculas y minúsculas**, por lo tanto "sqrt" es diferente de "Sqrt" o de "sQrt", "SQRT", etc.

En este acápite se hace un breve repaso de los fundamentos básicos del lenguaje. Estos fundamentos deben ser estudiados por comparación con los fundamentos del lenguaje de programación estudiado en la materia pre-requisito. Si dicho lenguaje es Javascript, entonces este acápite les sirve para refrescar los conocimientos adquiridos y en cualquier caso sirve como referencia futura (***tome en cuenta que, al ser las evaluaciones con consulta, no es necesario memorizar la información que se les proporciona, pero sí comprenderla***).

1.3.1. Variables

Como ocurre con la mayoría de los lenguajes interpretados, las variables en JavaScript pueden almacenar cualquier tipo de información. Las variables toman el tipo de dato en función al valor que almacenan.

Los nombres de las variables pueden estar formados por letras, números y los símbolos "$" y "_". Los nombres de las variables no pueden comenzar con números.

Para declarar una variable se emplea la palabra "**var**" seguida del nombre de la variable (o del nombre de las variables separados con compas, si se quiere declarar más de una variable).

Por ejemplo, las siguientes instrucciones declaran (crean) las variables a, b, c y d:

```
var a;
var b, c, d;
```

Estas variables, sin embargo, no tienen un valor asignado. Una vez declaradas se les puede asignar valores con el operador de asignación "**=**", por ejemplo:

```
a = 4.23;
b = "Javascript";
c = 125;
d = [1,2,3,4];
```

Donde a la variable "a" se le ha asignado un número real, a la variable "b" un texto, a la variable "c" un entero y a la variable "d" un vector. Observe también que después de cada instrucción se ha escrito un punto y coma (";"). En este ejemplo en particular el punto y coma no es imprescindible (porque cada instrucción se encuentra en una línea diferente), sin embargo, para evitar errores, ***se debe escribir un punto y coma al final de cada instrucción***.

Es posible declarar una variable y asignarle un valor inicial al mismo tiempo (variables inicializadas), por ejemplo las anteriores declaraciones y asignaciones se pueden hacer al mismo tiempo con:

```
var a = 4.23, b = "Javascript", c = 125, d = [1,2,3,4];
```

Si no se declara una variable con la palabra "**var**", Javascript crea una variable global, lo que puede dar lugar a errores difíciles de encontrar dentro de los programas, por eso **se recomienda que las variables en Javascript sean declaradas siempre con la palabra "var"**.

1.3.2. Operadores

En JavaScript se pueden emplear los siguientes operadores aritméticos:

+	:	Suma
-	:	Resta
*****	:	Multiplicación
/	:	División
%	:	Residuo de la división
++	:	Incremento
--	:	Decremento

Los cuatro primeros permiten realizar las operaciones básicas, el quinto "%" devuelve el residuo de la división (entera o real), el operador "++" incrementa en uno el valor de una variable numérica, mientras que "--" disminuye en uno el valor de una variable numérica.

Los dos últimos operadores ("++" y "--") tienen un comportamiento que varía en función a si son escritos como prefijo (antes) o sufijo (después) de la variable: si se encuentran como prefijo primero incrementan (o disminuyen) el valor de la variable y luego realiza las operaciones con el nuevo valor de la variable, mientras que como sufijo primero realiza las operaciones con el valor original de la variable y luego incrementa o disminuye su valor.

A más del operador de asignación ("**=**"), en JavaScript se cuentan con los operadores de asignación compuestos:

+=	:	Suma y asignación (x+=5; equivale a x=x+5;)
-=	:	Resta y asignación (x-=5; equivale a x=x-5;)
***=**	:	Multiplicación y asignación (x*=5; equivale a x=x*5;)
/=	:	División y asignación (x/=5; equivale a x=x/5;)
%=	:	Residuo y asignación (x%=5; equivale a x=x%5;)

1.3.3. Estructuras "if-else"

La estructura ***if-else*** tiene la siguiente sintaxis:

```
if (condición) {
  instrucciones 1;
} else {
  instrucciones 2;
}
```

Si la condición es verdadera se ejecutan las "instrucciones 1" y si es falsa "las instrucciones 2". Como en otros lenguajes el caso contrario

("else") es opcional. Además, si en las "instrucciones" sólo se tiene una instrucción, no es necesario escribir las llaves, aunque el escribirlas no constituye un error.

Otro aspecto que es importante hacer notar es que **JavaScript no distingue los espacios en blanco adicionales (incluido los saltos de línea y los tabuladores)**. Así la instrucción "if-else" puede ser escrita también de las siguientes formas:

```
if (condición) {instrucciones 1;} else { instrucciones 2;}
```

```
if     (  condición   )
{
  instrucciones 1;
}
else
{
  instrucciones 2;
}
```

```
if(condición){instrucciones 1;}else{instrucciones 2;}
```

```
if
(condición)
{    instrucciones 1;    }
else
{    instrucciones 2;     }
```

Y por supuesto muchas otras formas más. Esta característica se aprovecha para ordenar (no desordenar) las instrucciones y hacerlas más legibles, sin embargo, no se debe olvidar que **los nombres de las variables, funciones y otros identificadores no pueden tener espacios en blanco**.

1.3.4. Operadores relacionales y lógicos

Como en otros lenguajes, la condición se construye empleando operadores relacionales y lógicos. Los operadores lógicos disponibles en Javascript son:

>	:	Mayor que
<	:	Menor que
==	:	Igual a
===	:	Igual y del mismo tipo que
<=	:	Menor o igual a
>=	:	Mayor o igual a
!=	:	Diferente de
!==	:	Diferente o de otro tipo que

Y los operadores lógicos son:

!	:	**No** lógico
&&	:	**Y** lógico
||	:	**O** lógico

El operador "**!**" devuelve verdadero si el valor al que precede es falso y falso si es verdadero. El operador "**&&**" devuelve verdadero sólo si los dos valores que compara son verdaderos, en cualquier otro caso devuelve

falso. El operador "||" devuelve falso sólo si los dos valores que compara son falsos, en cualquier otro caso devuelve verdadero.

En Javascript cualquier valor diferente de 0 es interpretado como verdadero, mientras que 0 es interpretado como falso, igualmente el texto vacío ("") es interpretado como falso y cualquier otro texto como verdadero. Sin embargo las expresiones relacionales y/o lógicas devuelven el valor lógico "true" (verdadero) o "false" (falso), que si se requiere son convertidos automáticamente por Javascript en 1 o 0.

1.3.5. La estructura "switch"

Si existen dos o más condiciones consecutivas y en las mismas se compara un mismo valor con otros, resulta más eficiente emplear la estructura "switch":

```
switch (n) {
  case valor 1:
    instrucciones 1;
    break;
  case valor 2:
    instrucciones 2;
    break;
  case valor 3:
    instrucciones 3;
    break;
  …
  default:
    instrucciones por defecto;
}
```

Donde "n" es el valor a comparar y "valor 1", "valor 2", etc., son los valores con los que se compara. Si "n" es igual a uno de estos valores se ejecutan las instrucciones que se encuentran dentro de ese caso ("case") y el programa continúa con la instrucción que sigue a "switch" (después de las llaves). Si el caso no tiene un "break", entonces se ejecutan las instrucciones de ese caso, luego las instrucciones del caso siguiente y luego del caso subsiguiente, hasta que se encuentre un "break" o hasta que termina la estructura.

1.3.6. El operador "?"

Si las instrucciones que se ejecutan en base a una condición son simples, se puede emplear el operador "?", en lugar de "if-else". La sintaxis de este operador es la siguiente:

```
r = (condición) ? instrucción_1 : instrucción_2;
```

Donde se ejecuta la "instrucción_1" si la "condición" es verdadera y la "instrucción_2" si es falsa. El resultado devuelto por la "instrucción_1" o "instrucción_2", es asignado a la variable "r" (o puede ser empleado directamente como argumento de una función o dentro de una expresión).

En realidad es posible escribir más de una instrucción, si se separan con comas y encierran entre paréntesis, en cuyo caso el valor devuelto es el resultado de la última instrucción ejecutada, sin embargo, si se tiene más de una instrucción generalmente es preferible emplear la estructura "if-else" en lugar del operador "?".

1.3.7. La estructura "while"

La estructura "while" tiene la siguiente sintaxis:

```
while (condición) {
  instrucciones;
}
```

En esta estructura las "instrucciones" se ejecutan, repetidamente, mientras la "condición" es verdadera. Si se trata de una sola instrucción no son necesarias las llaves, pero el escribirlas tampoco constituye un error.

1.3.8. La estructura "do-while"

Esta estructura tiene la siguiente sintaxis:

```
do {
  instrucciones;
} while (condición);
```

Esta estructura tiene la misma lógica que la anterior, excepto que la condición está al final del ciclo, razón por la cual las "instrucciones" se ejecutan por lo menos una vez (mientras que con "while" es posible que no se ejecuten nunca).

1.3.9. La estructura "for"

La estructura "for" tiene la siguiente sintaxis:

```
for (inicialización; condición; incremento) {
  instrucciones;
}
```

En esta estructura las "instrucciones" se repiten mientras la "condición" es verdadera, sin embargo, a diferencia de "while", primero se ejecutan las instrucciones escritas en "inicialización" (una sola vez) y antes de repetir el ciclo se ejecutan las instrucciones escritas en "incremento" (cada vez que se repite el ciclo). Normalmente en "inicialización" se escriben las instrucciones que inicializan la o las variables (separadas con comas), mientras que en "incremento" se escriben las instrucciones que incrementan o disminuyen el valor de los contadores.

1.3.10. La estructura "for-in"

La sintaxis de esta estructura es:

```
for (atributo in objeto) {
  instrucciones;
}
```

Esta estructura realiza una iteración por cada elemento que existe en el "objeto", recuperando en la variable "atributo", una a una, las propiedades del objeto. Si el objeto es un vector o texto, los atributos son los índices del vector, es decir números enteros consecutivos, comenzando en 0.

1.3.11. Los comandos "break" y "continue"

Para salir del interior de un ciclo, sin que se cumpla la condición del mismo, se escribe el comando "break". Cuando Javascript encuentra este comando, termina inmediatamente la ejecución del ciclo y pasa a ejecutar la siguiente instrucción del programa (la que está después del ciclo).

Para continuar con el siguiente ciclo, sin terminar el ciclo actual, se escribe el comando "continue". Cuando Javascript encuentra el comando "continue" pasa inmediatamente al siguiente ciclo, dejando sin ejecutar las instrucciones restantes.

1.3.12. Ciclos infinitos

La aparición de un ciclo infinito en un programa indica la existencia un error: una condición errónea. No obstante, cuando debido a la lógica del problema, se debe salir del ciclo en algún punto intermedio del mismo (no al principio ni al final) se crea un ciclo infinito (escribiendo "true" en lugar de la condición) entonces se sale del ciclo, desde cualquier punto intermedio del mismo, empleando el comando "break".

La sintaxis de un ciclo que termina en algún punto intermedio del mismo es aproximadamente la siguiente:

```
while (true) {
  instrucciones;
  if (condición) break;
  instrucciones;
}
```

Que puede ser implementada también con las estructuras "do-while" y "for":

```
do {
  instrucciones;
  if (condición) break;
  instrucciones;
} while (true);

for (;;) {
  instrucciones;
  if (condición) break;
  instrucciones;
}
```

Observe que en el caso de la estructura "for", para crear un ciclo infinito, simplemente se dejan en blanco los sectores de inicialización, condición e incremento.

1.4. ESCRITURA DE CÓDIGO JAVASCRIPT

Una vez repasados los fundamentos del lenguaje, se pueden elaborar algunos programas básicos en Javascript.

Como ya se dijo, Javascript es un lenguaje que corre en los navegadores Internet, por lo tanto, un programa Javascript, debe ser escrito y/o importado en una página HTML. Ello implica aprender también algo de HTML (el lenguaje de las páginas WEB).

Las instrucciones en HTML se escriben dentro de etiquetas. Las etiquetas son palabras reservadas que se encierran entre los símbolos "< >". La mayoría de las etiquetas tienen una etiqueta de apertura y otra de cierre.

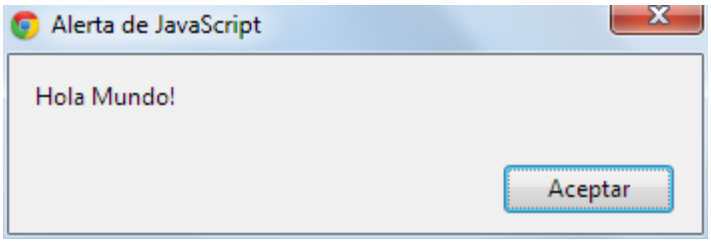
La etiqueta de cierre se diferencia de la de apertura porque está precedida por el símbolo "/", por ejemplo todas las etiquetas de un página HTML se encierran dentro de la etiqueta "html":

```
<html>
. . .
</html>
```

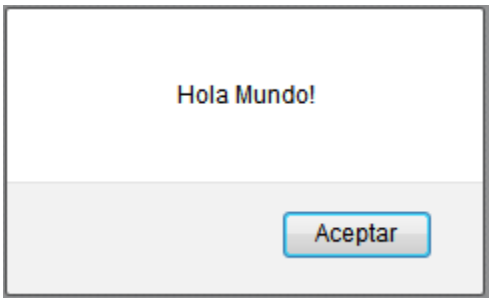
La etiqueta dentro la cual se escribe (o importa) código Javascript es "script". En la mayoría de los navegadores es suficiente escribir esta etiqueta para que sea reconocido como código (instrucciones) Javascript. Así por ejemplo si se crea el archivo "hola.html", en Notepad, Notepad++, DroidEdit o cualquier otro editor de textos y se escribe dentro del mismo lo siguiente:

```
<script>
  alert("Hola Mundo!");
</script>
```

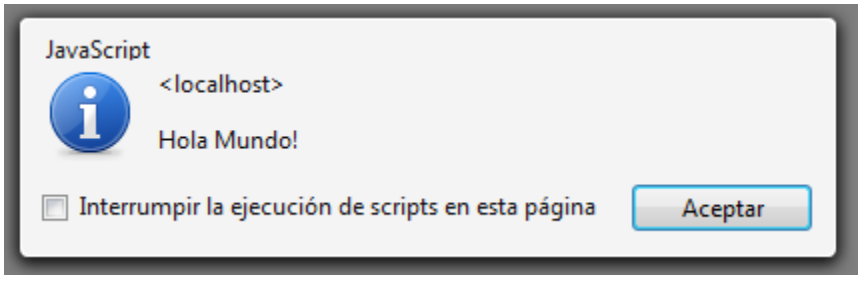
Al abrir el archivo, en un navegador, aparece una ventana de alerta con el mensaje "Hola Mundo!" (la instrucción "alert" de Javascript, sirve justamente para ese fin). La apariencia de esta ventana varía de un navegador a otro, pero no su funcionalidad, así en Google Chrome es:



En Firefox:

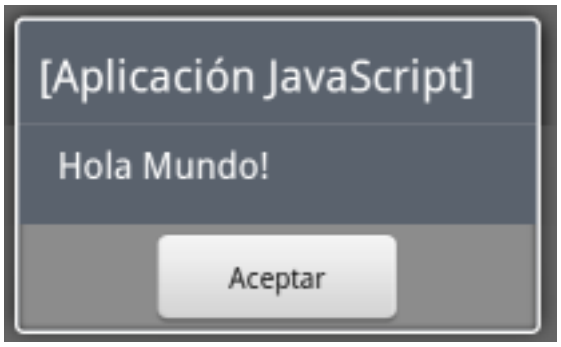


Y en Opera:



Por supuesto, la apariencia varía también en los dispositivos móviles, así en Dophin (en los dispositivos Android) es:

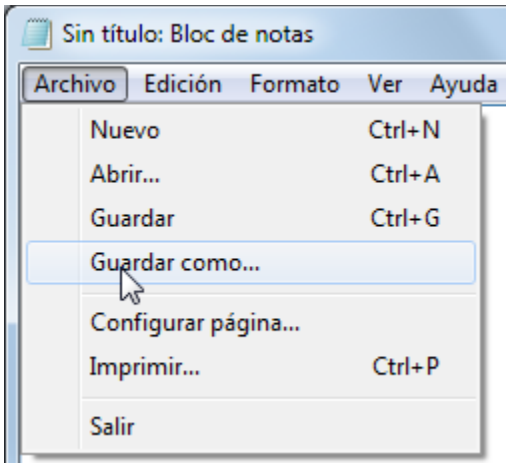
En Firefox:



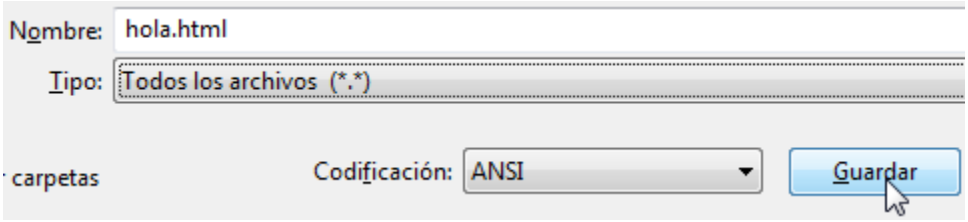
Y en Opera:

UC Browser tiene algún defecto (un bug) con las ventanas emergentes pues no las muestra ni genera ningún mensaje de error.

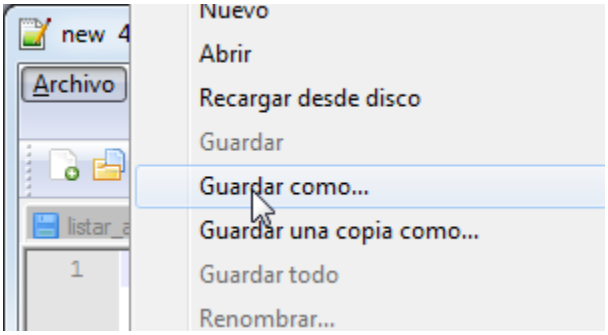
Ahora se verá con más detalle cómo crear un archivo HTML. Una vez abierto el editor, simplemente se elige la opción archivo->nuevo y se guarda el archivo con la extensión ".html". Así en Notepad (en el block de notas), se elige la opción "Guardar como…":



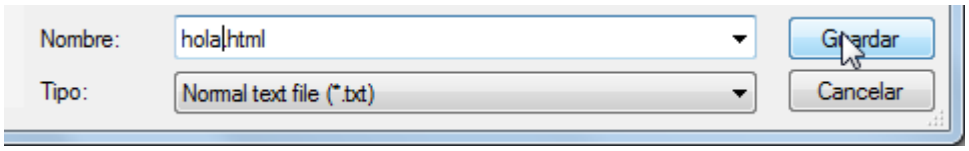
En la ventana que aparece se elige la carpeta en la cual se quiere guardar el archivo y se guarda el archivo con el nombre "hola.html":



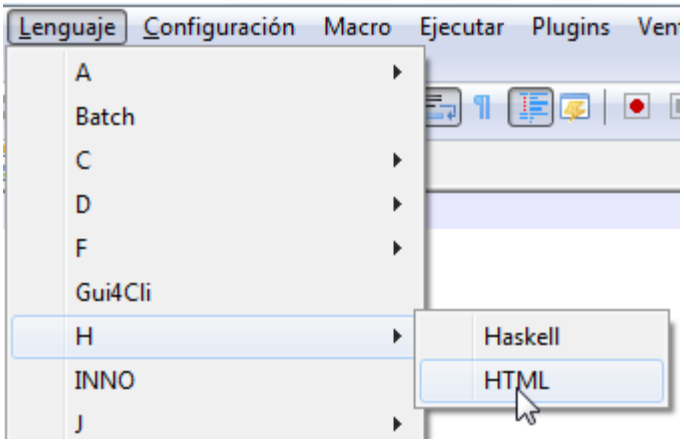
Igualmente, en Notepad++, después de crear una nueva página ("Ctrl+N" o menú Archivo->Nuevo), se elige la opción "Guardar como…"



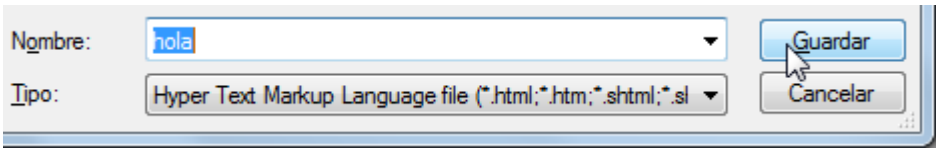
Y se guarda el archivo con la extensión ".html":



Alternativamente, en Notepad++, se puede elegir primero el lenguaje:



Y luego guardar el archivo (Archivo->guardar) con el nombre "hola" (la extensión "html") se añade automáticamente:

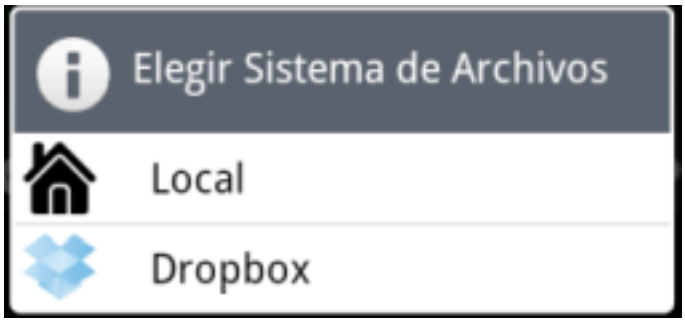


En los dispositivos móviles con sistema Android, si se está trabajando con DroidEdit, se abre el editor, haciendo tap (clic) sobre el icono de la aplicación:

Por supuesto se pueden emplear otros editores como WebMaster's HTML, DreamEdit, Syntax Highlighted, etc. Una vez en el navegador se crea un nuevo archivo haciendo tap (clic) en el icono respectivo, que en el caso de DroidEdit es:

Entonces se escribe algún carácter (para que se habilite el icono "guardar") y se guarda el archivo haciendo tap en el icono guardar, que en el caso de DroidEdit es:

Entonces en la ventana que aparece se hace tap en "Local" (es decir que se elige guardar el archivo en el dispositivo móvil):



Y en la nueva ventana que aparece se elige la carpeta en la que se quiere guardar el archivo y se crea el archivo con el nombre "hola.html":

Como se ha visto, es posible escribir código Javascript simplemente con la etiqueta "script", sin embargo, para crear programas realmente útiles, es necesario emplear algunas otras etiquetas más. La estructura HTML básica que se empleará para escribir programas en Javascript, es:

```
<html>
<head>
  <title>Título de la página</title>
  <script type="text/javascript">
    código Javascript
  </script>
</head>
<body>
  Etiquetas HTML
  <script type="text/javascript">
    código Javascript
  </script>
  Etiquetas HTML
</body>
</html>
```

Como ya se explicó, la etiqueta "html" le indica al navegador que su contenido es HTML.

La etiqueta "head" identifica el encabezado de la página. El código Javascript escrito en esta parte, así como otros elementos que se incluyen en esta parte, se cargan en primer lugar.

La etiqueta "title" sirve para darle un título a la página, este título es el que aparece en la pestaña del navegador, ayudando a identificar la página (por supuesto no es imprescindible).

La etiqueta "body" identifica el cuerpo del documento. Es en esta parte donde se escribe la mayoría de las instrucciones HTML.

Como se puede observar, las instrucciones Javascript pueden ser escritas tanto en el encabezado como en el cuerpo del documento. Es posible también escribir varios bloques de código en diferentes partes del documento (dentro de etiquetas "script") y en ese caso son ejecutados secuencialmente, en el orden en que fueron escritos.

Sin embargo, es aconsejable escribir las instrucciones en un solo bloque: en el encabezado del documento (head). De esa manera se garantiza que las funciones, estén disponibles para su uso desde un principio, además se facilita su mantenimiento y se evita entremezclar código Javascript con instrucciones HTML.

En realidad, lo más aconsejable es escribir el código Javascript en un archivo independiente (con la extensión ".js") e importar el mismo en la página HTML con la siguiente etiqueta:

```
<script type="text/javascript" src="nombre_del_archivo_Javascript.js"></script>
```

Donde se reemplaza "nombre_del_archivo_Javascript" por el nombre real del archivo que contiene el código Javascript.

El atributo "type", que se ha escrito en todas las etiquetas "script", no es realmente necesario. La mayoría de los navegadores actuales asumen que, si no existe el atributo "type", se trata de código Javascript, pero, por razones de claridad y universalidad, es conveniente escribirlo.

Con estas consideraciones se reescribe el programa "Hola Mundo!", teniendo en Notepad++ la siguiente apariencia:

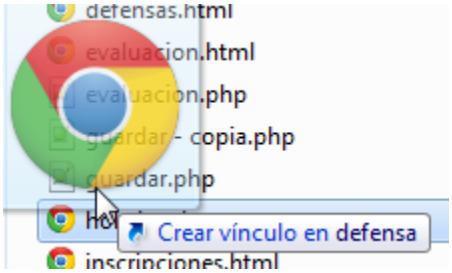
```
1  <html>
2  <head>
3    <title>Hola Mundo</title>
4    <script type="text/javascript">
5      alert("Hola Mundo!");
6    </script>
7  </head>
8  <body>
9  </body>
10 </html>
```

Y en DroidEdit:

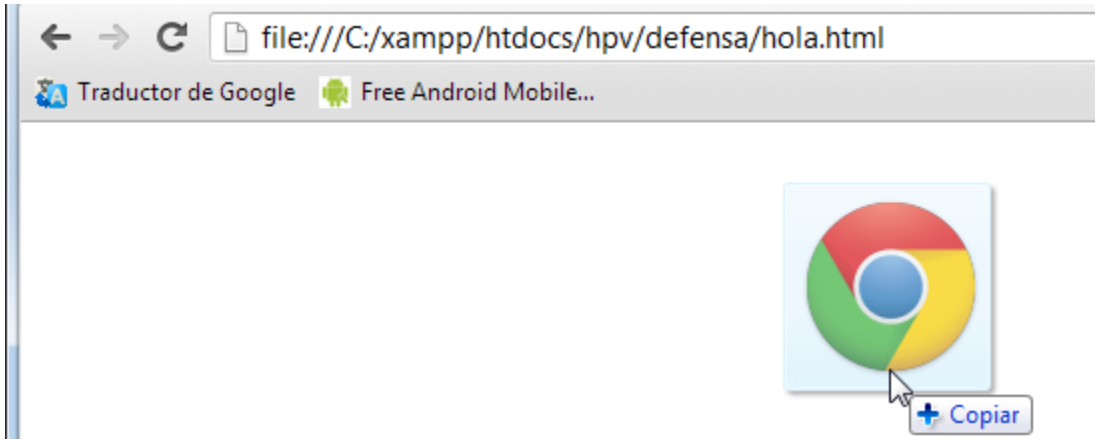
```
1  <html>
2  <head>
3    <title>Hola Mundo</title>
4    <script type="text/javascript">
5      alert("Hola Mundo!");
6    </script>
7  </head>
8  <body>
9  </body>
10 </html>
```

Por supuesto este programa hace lo mismo que el anterior (con excepción del título de la página). ***No debe olvidar "guardar" el archivo una vez escrito el código***.

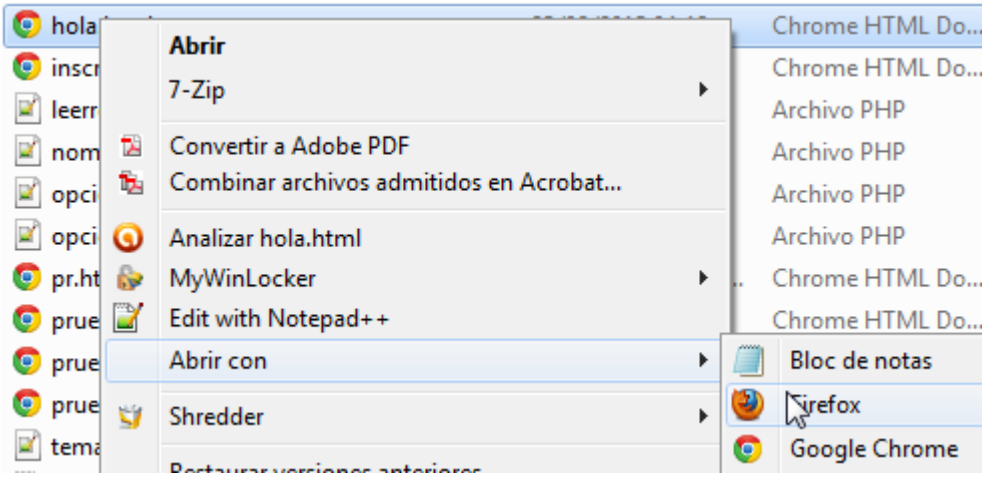
Una vez creado el archivo HTML debe ser abierto en un navegador. Y para ello existen varias alternativas, la más sencilla (en Windows) consiste en arrastrar el icono del archivo:



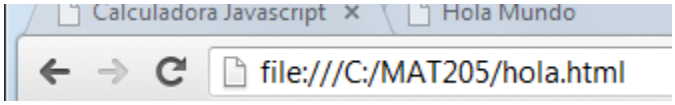
Y soltarlo sobre la ventana del navegador:



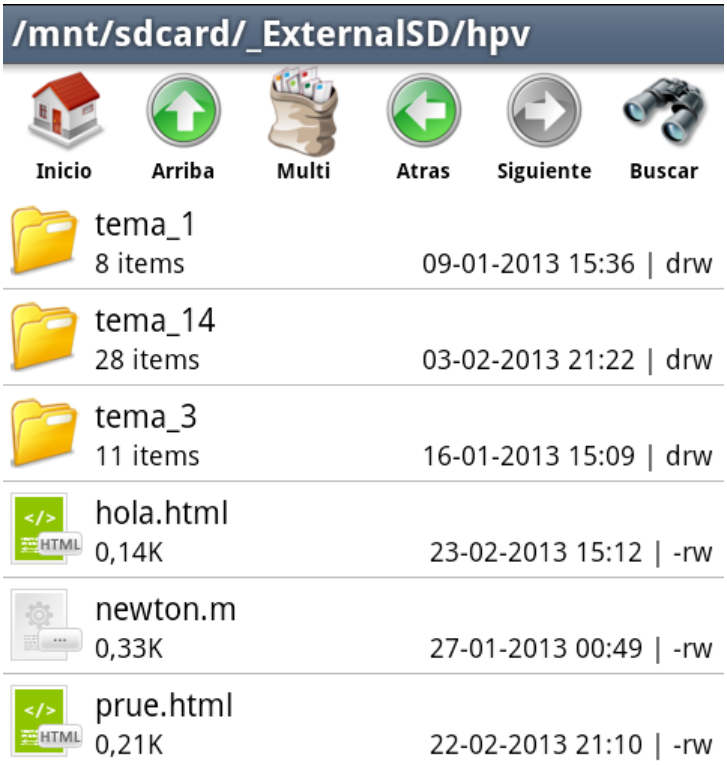
También se puede hacer clic sobre el archivo, con el botón derecho del mouse y en el menú emergente se elige "abrir con..", seleccionando luego el navegador con el cual se quiere abrir la página:



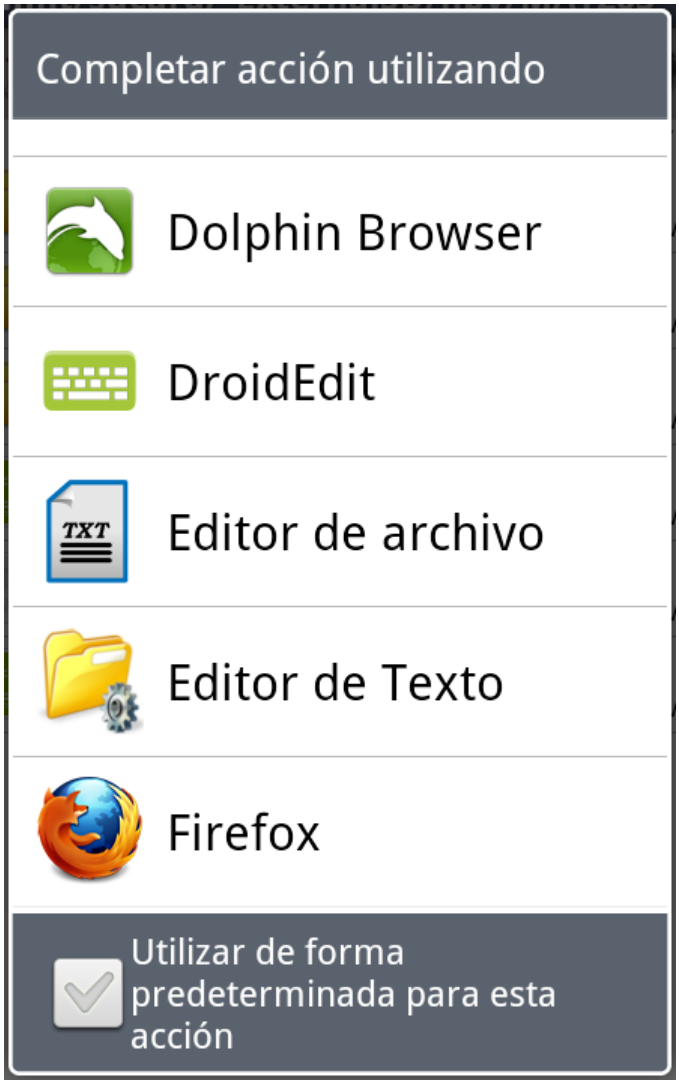
Es posible también escribir directamente, en el campo de direcciones del navegador, el camino donde se encuentra el archivo, precediendo dicho camino con la palabra file:///, así si el archivo se encuentra en C:\MAT205\hola.html, se escribe:



En los dispositivos móviles con sistema Android, se abre la página ubicando el mismo con un administrador de archivos (como File Manager):



Entonces se hace tap en el archivo (en "hola.html") y en el menú que aparece se hace tap en el navegador con el cual se quiere abrir la página:



Es posible también escribir en el navegador el camino donde se encuentra el archivo (de manera similar a como se procede en Windows).

A más del editor de textos, ***en los dispositivos Android se recomienda instalar el teclado "Multiling"*** (de play store). Este teclado está disponible en varios idiomas y tiene una configuración que facilita considerablemente la escritura de código.

1.5. EJERCICIOS

1. Cree la página "saludo.html" que muestre un mensaje de alerta con su nombre completo.
2. Cree la página "info.html" que muestre un primer mensaje de alerta (escrito en el encabezado de la página) con el número de su carnet de identidad y un segundo mensaje (escrito en el cuerpo del documento) con el número de su carnet universitario.

1.6. FUNCIONES MATEMÁTICAS EN JAVASCRIPT

Las funciones matemáticas en Javascript han sido definidas dentro del objeto "Math". En Javascript, para acceder tanto las funciones como a los datos de un objeto, se escribe el nombre del objeto, un punto y el nombre

de la función, es decir:

```
Nombre_del_objeto.Nombre_de_la_función_o_dato
```

Por ejemplo, para calcular la raíz cuadrada de 4.52 se escribe:

```
Math.sqrt(4.52)
```

Las funciones matemáticas existentes en este objeto son:

sin(x):	Seno de en ángulo en radianes "x".
cos(x):	Coseno del ángulo en radianes "x".
tan(x):	Tangente del ángulo en radianes "x".
asin(x):	Arco seno del valor x, valor devuelto en radianes.
acos(x):	Arco coseno del valor x, valor devuelto en radianes.
atan(x):	Arco tangente del valor x, valor devuelto en radianes.
atan2(x,y):	Arco tangente del cociente de "x" y "y".
log(x):	Logaritmo natural de "x" ($\log_e(x)$
exp(x):	Exponente de "x" (e^x).
sqrt(x):	Raíz cuadrada de "x".
pow(x,y):	Resultado de elevar "x" a "y" (x^y).
abs(x):	Valor absoluto de "x" (x sin signo).
round(x):	Valor de "x" redondeado al entero más cercano (el entero mayor si la parte fraccionaria es mayor o igual a 0.5).
ceil(x):	Valor de "x" redondeado al entero mayor inmediato.
floor(x):	Valor de "x" redondeado al entero menor inmediato.
random():	Número real aleatorio comprendido entre 0 y 1.

Donde también se ha definido las siguientes las constantes matemáticas:

E:	Número de Euler.
LN2:	Logaritmo natural de 2.
LN10:	Logaritmo natural de 10.
LOG2E:	Logaritmo del número de Euler en base 2.
LOG10E:	Logaritmo de Euler en base 10.
PI:	Número pi.
SQRT1_2:	Raíz cuadrada de 1/2.
SQRT2:	Raíz cuadrada de 2.

A propósito, en la programación orientada a objetos las funciones se denominan "métodos" y los datos "propiedades" o "atributos", por lo tanto, las anteriores funciones son más propiamente "métodos" del objeto "Math" y las constantes los "atributos" de dicho objeto.

Cuando se trabaja con expresiones matemáticas, resulta tedioso tener que escribir, una y otra vez, "Math." delante del nombre de cada una de las funciones y/o constantes matemáticas. Afortunadamente, en Javascript es muy simple asignar otro nombre (un alias) a un método o atributo. Por ejemplo, para calcular la raíz cuadrada con "sqrt" en lugar de "Math.sqrt", se escribe:

```
sqrt = Math.sqrt;
```

Después de esta asignación, la raíz cuadrada de 4.52 se calcula simplemente con:

```
sqrt(4.52)
```

Sin embargo este alias es local: sólo puede ser empleado en la página donde ha sido creado, por lo que si se requiere en otra, debe volver a ser creado. El crear alias, en cada página que así lo requiera, no es nada práctico, por lo que debe ser evitado mediante la creación de una librería (un archivo Javascript independiente) donde se crean, una sola vez,

todos los alias para todas las funciones matemáticas. Esto es justamente lo que se ha hecho en la librería "mat205.js".

En la librería "mat205.js" (que se distribuye con el material del tema) no sólo se han creado alias para todas las expresiones matemáticas existente, sino que además se han añadido algunas funciones matemáticas que no están presentes en el objeto "Math".

Las funciones matemáticas añadidas en "mat205.js" son:

csc(x)	Cosecante de x.
sec(x)	Secante de x.
cot(x)	Cotangente de x.
sinh(x)	Seno hiperbólico de x.
cosh(x)	Coseno hiperbólico de x.
tanh(x)	Tangente hiperbólica de x.
csch(x)	Cosecante hiperbólica de x.
sech(x)	Secante hiperbólica de x.
coth(x)	Cotangente hiperbólica de x.
asinh(x)	Arco seno hiperbólico (seno hiperbólico inverso) de x.
acosh(x)	Arco coseno hiperbólico (coseno hiperbólico inverso) de x.
atanh(x)	Arco tangente hiperbólico de x.
sqr(x)	Cuadrado de x.
sign(x)	Signo de x: 1 positivo, -1 negativo, 0 cero.
cbrt(x)	Raíz cúbica de x.
odd(x)	Impar. Verdadero si x es impar, falso en caso contrario.
even(x)	Par. Verdadero si x es par, falso en caso contrario.
xroot(x,n)	Raíz enésima (n) de x.
quot(x,y)	Cociente de la división entre "x" y "y".
frac(x)	Parte fraccionaria de x.
isInteger(x)	Es entero. Veradero si x es un entero.
toDeg(x)	Convierte x de radianes a grados.
toRad(x)	Convierte x de grados a radianes.
log10(x)	Logaritmo en base 10 de x.
log2(x)	Logaritmo en base 2 de x.
logb(b,x)	Logaritmo en base "b" de x.
fact(n)	Factorial de n.

Esta librería cuenta también con muchas otras funciones y se le añadirán otras más a medida que se avance en la materia, por lo tanto, será actualizada frecuentemente.

Para emplear esta librería, simplemente se la importa en la cabecera de la página HTML con la etiqueta "script" (como ya se explicó previamente):

```
<script type="text/javascript" src="mat205.js"></script>
```

Por ejemplo, para calcular el valor de la siguiente expresión matemática:

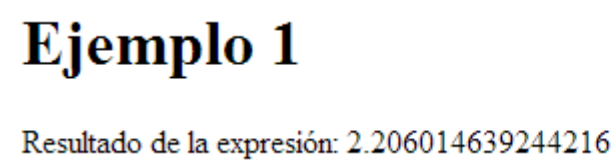
$$\ln(\csc(7) - \sec(9.2) + \tan(0.4))^{9.3}$$

Se puede crear la siguiente página HTML (ejemplo1.html):

```
1  <html>
2  <head>
3    <title>Ejemplo 1</title>
4    <script type="text/javascript" src="mat205.js"></script>
5  </head>
6  <body>
7    <h1>Ejemplo 1</h1>
```

```
  <p>Resultado de la expresión:
    <script type="text/javascript">
      var r = pow(log(csc(7)-sec(9.2)+tan(0.4)),9.3);
      document.write(r);
    </script>
  </p>
</body>
</html>
  <p>Resultado de la expresión:</p>
  <script type="text/javascript">
```

Que al ser abierta en el navegador muestra lo siguiente:



Si el navegador no muestra el resultado, lo más probable es que se haya cometido algún error al escribir el código Javascript, por lo debe ser revisado. Por defecto los navegadores Internet no muestran ningún mensaje cuando se produce algún error en la ejecución del código Javascript. ***Si no se logra el resultado buscado, se debe asumir que se ha cometido algún error y revisar el código***. El cometer errores de escritura (sintaxis) es lo más frecuente cuando se programa y es más frecuente en los primeros programas, por eso es necesario acostumbrarse a revisar el código, encontrar los errores y corregirlos, tratando de evitar volver a cometerlos.

En esta página se han empleado dos nuevas etiquetas HTML: "h1" y "p".

La etiqueta "h1" muestra, el texto que se escribe en su interior, como un encabezado de primer nivel (un título). En HTML se tienen etiquetas tipo encabezado hasta un sexto nivel: "h2", "h3", "h4", "h5" y "h6".

La etiqueta "p", por otro lado, muestra el texto que se escribe en su interior como un párrafo, es decir que añade un salto de línea (y cierto espaciamiento) antes y después del texto.

Observe que, como era de esperar, la expresión matemática ha sido evaluada sin necesidad de escribir "Math.". Es importante aclarar que para que el código funciones correctamente ***el archivo "mat205.js" (la librería) debe estar en el mismo directorio que la página HTML***.

En esta página se ha empleado también una nueva instrucción Javascript: "document.write". Esta instrucción inserta en la página, el texto que se escribe, como código HTML. El objeto "document" representa a la página HTML y a más de "write" cuenta con otras propiedades y métodos, algunos de los cuales serán estudiados posteriormente.

En el ejemplo, la instrucción "document.write" transforma, automáticamente, el resultado almacenado en a variable "r" en texto e inserta dicho texto en la página (dentro de la etiqueta "p").

Puesto que "document.write" permite insertar instrucciones HTML en la página, es posible crear todo el contenido de la página directamente en Javascript. Por ejemplo para calcular el valor de la siguiente expresión matemática, para valores de "y" iguales a 3.1, 4.2 y 7.3:

$$\sqrt[3]{\frac{y+4!+y^{3.2}}{e^{y}+\cos(y)}}$$

Se puede crear la siguiente página HTML (ejemplo2.html):

```
<html>
<head>
   <title>Ejemplo 2</title>
   <script type="text/javascript" src="mat205.js"></script>
   <script type="text/javascript">
      document.write("<h1>Ejemplo 2</h1>");
      var y = 3.1;
      var r = cbrt((y+fact(4)+pow(y,3.2))/(exp(y)+cos(y)));
      document.write("<p>Resultado de la expresión para y = "+
        y+" => "+r+"</p>");
      y = 4.2;
      r = cbrt((y+fact(4)+pow(y,3.2))/(exp(y)+cos(y)));
      document.write("<p>Resultado de la expresión para y = "+
        y+" => "+r+"</p>");
      y = 7.3;
      r = cbrt((y+fact(4)+pow(y,3.2))/(exp(y)+cos(y)));
      document.write("<p>Resultado de la expresión para y = "+
        y+" => "+r+"</p>");
   </script>
</head>
<body>
</body>
</html>
```

Que al ser abierta en un navegador muestra lo siguiente:

Ejemplo 2

Resultado de la expresión para y = 3.1 => 1.4487162948454166

Resultado de la expresión para y = 4.2 => 1.2423120431812926

Resultado de la expresión para y = 7.3 => 0.7441584513794072

Como se puede observar en este programa, para concatenar (unir) texto se emplea el operador de suma ("+"), además, Javascript convierte automáticamente los valores numéricos en texto.

Como ya se explicó, Javascript ignora los espacios en blanco adicionales (incluyendo saltos de línea y tabulaciones). En este programa se ha aprovechado ese hecho para dividir, en dos líneas, las instrucciones "document.write" de manera que quepan en la pantalla, sin embargo, el programa funciona igual si dichas instrucciones son escritas en una sola línea.

Observe también que las variables "y" y "r" han sido declaradas una sola vez (con "var"), pues una vez declaradas se les puede asignar nuevos

valores, sin necesidad de volver a declararlas, las veces que se requiera.

Como es lógico suponer, al crear este programa las instrucciones han sido escritas una sola vez: para el primer valor. Para los dos valores restantes, simplemente se han copiado y pegado estas instrucciones cambiando luego el valor de "y" y borrando la palabra "var".

Aunque en los dos ejemplos anteriores se ha creado una página HTML para cada expresión, como se supondrá, es perfectamente posible evaluar tantas expresiones como se quiera en un sola página. Por ejemplo, para encontrar el valor de la siguiente expresión:

$$\frac{\log\left(x+\sqrt{y+z}\right)}{\sqrt[5]{x+y^z}}$$

Para x=1.1, y=1.4, z=1.9; x=2.3, y=5.2, z=8.3; x=1.9, y=1.2, z=3.1. Y el valor de la siguiente expresión:

$$\log\left(\frac{\sin(x)+\cos(y)}{3.4\tan(x+y)}\right)$$

Para x=130°, y=200°; x=200°, y=120°; x=160°, y=125°, se puede crear la siguiente página HTML (ejemplo3.html):

```
<html>
<head>
  <title>Ejemplo 3</title>
  <script type="text/javascript" src="mat205.js"></script>
  <script type="text/javascript">
    document.write("<h1>Ejemplo 3</h1>");
    document.write("<h2>Resultados de la primera expresión:</h2>");
    var x=1.1, y=1.4, z=1.9;
    var r = log10(x+sqrt(y+z))/xroot(x+pow(y,z),5);
    document.write("<p>Para x="+x+", y="+y+", z="+z+" => "+r+"</p>")
    x=2.3; y=5.2; z=8.3;
    r = log10(x+sqrt(y+z))/xroot(x+pow(y,z),5);
    document.write("<p>Para x="+x+", y="+y+", z="+z+" => "+r+"</p>")
    x=1.9, y=1.2, z=3.1;
    r = log10(x+sqrt(y+z))/xroot(x+pow(y,z),5);
    document.write("<p>Para x="+x+", y="+y+", z="+z+" => "+r+"</p>")
    document.write("<h2>Resultados de la segunda expresión:</h2>");
    var x0=130, y0=200;
    x=toRad(x0), y=toRad(y0);
    r = log10((sin(x)+cos(y))/(3.4*tan(x+y)));
    document.write("<p>Para x="+x0+"°, y="+y0+"° => "+r+"</p>");
    x0=200, y0=120;
    x=toRad(x0), y=toRad(y0);
    r = log10((sin(x)+cos(y))/(3.4*tan(x+y)));
    document.write("<p>Para x="+x0+"°, y="+y0+"° => "+r+"</p>");
    x0=160, y0=125;
    x=toRad(x0), y=toRad(y0);
    r = log10((sin(x)+cos(y))/(3.4*tan(x+y)));
    document.write("<p>Para x="+x0+"°, y="+y0+"° => "+r+"</p>");
  </script>
```

```
31  </head>
32  <body>
33  </body>
34  </html>
```

Que al ser abierta en un navegador muestra:

Ejemplo 3

Resultados de la primera expresión:

Para x=1.1, y=1.4, z=1.9 => 0.373295627961028

Para x=2.3, y=5.2, z=8.3 => 0.0502867873527808

Para x=1.9, y=1.2, z=3.1 => 0.4622452740293394

Resultados de la segunda expresión:

Para x=130°, y=200° => -1.0532480596708227

Para x=200°, y=120° => -0.5299699661565509

Para x=160°, y=125° => -1.7387698762266683

1.7. EJERCICIOS

Para presentar los ejemplos y ejercicios del tema, debe crear una carpeta con el nombre "tema1" y guardar en la misma todos los archivos HTML creados.

3. Cree una página HTML para calcular el valor de la siguiente expresión:

$$\sqrt{\frac{\log(6.75)}{6.2 + 1.3^{2.7}}}\cos(5.2)$$

4. Cree una página HTML para calcular el valor de la siguiente expresión, para valores de "x" iguales a 2.3, 4,5 y 7.2.

$$\sin^{-1}\left(\frac{\sqrt{x + 2x^{0.3} + 1.2}}{2x + 0.5}\right)$$

5. Cree una página HTML para calcular el valor de la siguiente expresión, para valores los siguientes valores de "x" y "y": x=9.2, y=5.6; x=2.3, y=8.4; x=7.1, y=6.3.

$$\left(\sin(x) - \frac{\sec(y)}{\sinh(x)}\right)^{0.34}$$

6. Cree una página HTML para calcular el valor de la siguiente expresión, para los siguientes valores de "x", "y" y "z": x=2.2, y=3.2, z=4.5; x=3.1, y=5.2, z=1.3; x=8.2, y=6.1, z=4.6.

$$\log\left(\frac{x^{3.2}+y^{1.6}+e^{z}}{\sqrt[5]{x+y^{z}}}\right)$$

7. Cree una página HTML para calcular el valor de las siguientes expresiones:

$$\ln(z+1)+\log(z+2.2)+\log_2(z)$$

Para valores de "z" iguales a: 1.2, 2.3 y 5.8.

$$\frac{\sinh^{-1}(x)+\cosh^{-1}(y)+\tanh^{-1}(z)}{x+y+z^{4.9}+6!}$$

Para los siguientes valores de "x", "y" y "z": x=12.1, y=10.2, z=0.5; x=6.7, y=8.2, z=0.7; x=8.2, y=9.5, z=0.2.

$$\frac{\sqrt[5]{\sin(x)+\cos(y)+\tan(z)}}{\sqrt[3]{e^{\sin(x+y)}+e^{\cos(z)}}}$$

Para los siguientes valores de "x", "y" y "z": x=21°, y=87°, z=35°; x=67°, y=82°, z=70° y x=182°, y=195°, z=200°.

2. FUNCIONES

En el anterior tema se vio que al calcular el valor de una expresión para diferentes valores, era necesario copiar la expresión y cambiar el valor de las variables tantas veces como se evaluaba la expresión.

Si bien el proceso no es demasiado moroso para dos o tres evaluaciones, se vuelve moroso cuando la expresión se evalúa más veces y en diferentes lugares, pero además, si se ha cometido un error en la expresión original, es necesario corregirla y volver a copiarla en todos esos lugares, o alternativamente corregir el error en todas las expresiones copiadas (lo que es igualmente moroso).

Además, es siempre posible modificar, inadvertidamente, una o varias de las expresiones dando lugar así a resultados inesperados, o corregir una expresión y olvidar actualizarla en los otros lugares donde fue copiada.

Por todas estas razones, los lenguajes de programación actuales permiten crear funciones. En una función se puede escribir la expresión matemática (o cualquier conjunto de instrucciones) y para evaluarla simplemente se escribe el nombre de la función y se le pasan los datos.

2.1. FUNCIONES EN JAVASCRIPT

Las funciones son subprogramas que pueden recibir datos, realizar operaciones y/o acciones con esos datos y devolver resultados, sin embargo, no todas las funciones reciben datos, por ejemplo una función que devuelve el número PI, no recibe ningún dato, simplemente devuelve el número PI. Igualmente, no todas las funciones devuelven un resultado, por ejemplo, una función que actualiza la pantalla, no devuelve ningún resultado, simplemente realiza una acción (actualizar la pantalla).

En métodos numéricos, no obstante, al ser en su mayoría funciones de tipo matemático, casi siempre recibirán por lo menos un dato y devolverán por lo menos un resultado.

Para crear una función en Javascript existen tres alternativas. La forma más frecuente (y la que con más frecuencia se empleará en la materia) es la forma "estándar":

```
function nombre_de_la_funcion(lista_de_parámetros) {
  instrucciones;
}
```

Donde "nombre_de_la_función" es el nombre que se da a la función y cuya denominación sigue las mismas reglas de las variables. Se debe evitar asignar nombres reservado por Javascript (como "if", "while", etc.) o nombres empleados en las librerías, como "sin", "cos", "quot", etc. En este último caso, no se produce ningún error, simplemente se crea la nueva función y desaparece la función original (la de la librería).

La "lista_de_parámetros" son los nombres de las variables en los que se guardan, temporalmente, los datos que se mandan a la función. Dichos nombres deben estar separados por comas y siguen las reglas de las variables.

Para devolver un resultado desde el interior de una función se emplea el comando "return", de acuerdo a la sintaxis:

```
return valor_o_expresión;
```

Cuando Javascript encuentra este comando termina la ejecución de la función y devuelve el resultado de evaluar el "valor_o_expresión". El

"valor_o_expresión" puede ser un valor literal (por ejemplo un número o texto), una variable (cuyo valor es devuelto) o una expresión (que es evaluada y cuyo resultado es devuelto).

Si en una función no se escribe el comando "return", la función termina, al ejecutarse su última instrucción, sin devolver ningún resultado.

La otra forma que se pueden crear funciones en Javascript es la "literal":

```
var variable=function nombre_de_la_funcion(lista_de_parámetros) {
  instrucciones;
}
```

Como se puede ver, la única diferencia con relación a la forma estándar, es que la función es asignada a una variable (se le da un alias), entonces la función puede ser llamada indistintamente con el "nombre_de_la_función" o el "nombre_de_la_variable" (el alias).

En la práctica, casi siempre se omite el "nombre_de_la_función", creándose así una función anónima que, sin embargo, puede ser llamada gracias al alias que se le asigna (el nombre de la variable):

```
var variable=function (lista_de_parámetros) {
  instrucciones;
}
```

Finalmente otra forma de crear funciones (que es empleada en raras ocasiones) es la siguiente (note que **F**unction está en mayúsculas):

```
var variable = new Function("par1","par2",…,"parn", "instrucciones");
```

Donde "par1", "par2",…,"parn" son los parámetros de la función e "instrucciones" son las instrucciones Javascript, es decir el código de la función. Esta forma permite crear funciones en tiempo de ejecución (funciones dinámicas) las cuales, a diferencia de las formas estándar y literal, son siempre globales (sin importar donde hayan sido creadas).

En Javascript las funciones pueden ser anidadas, es decir que es posible crear una función dentro de otra (excepto, por supuesto con la última forma, que siempre crea funciones globales).

A manera de demostración, se volverá a resolver el ejemplo 3 del anterior tema empleando funciones, es decir se encontrará el valor de la siguiente expresión para x=1.1, y=1.4, z=1.9; x=2.3, y=5.2, z=8.3; x=1.9, y=1.2, z=3.1:

$$\frac{\log\left(x+\sqrt{y+z}\right)}{\sqrt[5]{x+y^{z}}}$$

Y el valor de la siguiente expresión para x=130°, y=200°; x=200°, y=120°; x=160°, y=125°:

$$\log\left(\frac{\sin(x)+\cos(y)}{3.4\tan(x+y)}\right)$$

La página HTML donde se resuelve el problema es:

```
1  <html>
2  <head>
3    <title>Ejemplo 1</title>
4    <script type="text/javascript" src="mat205.js"></script>
```

```
<script type="text/javascript">
  function f1(x,y,z){
    return log10(x+sqrt(y+z))/xroot(x+pow(y,z),5);
  }
  function f2(x,y){
    return log10((sin(x)+cos(y))/(3.4*tan(x+y)));
  }
  document.write("<h1>Ejemplo 1</h1>");
  document.write("<h2>Resultados de la primera expresión:</h2>");
  document.write("<p>Para x=1.1, y=1.4, z=1.9 => "+f1(1.1,1.4,1.9)
    +"</p>");
  document.write("<p>Para x=2.3, y=5.2, z=8.3 => "+f1(2.3,5.2,8.3)
    +"</p>");
  document.write("<p>Para x=1.9, y=1.2, z=3.1 => "+f1(1.9,1.2,3.1)
    +"</p>");
  document.write("<h2>Resultados de la segunda expresión:</h2>");
  document.write("<p>Para x=130°, y=200° => "+f2(toRad(130),
    toRad(200))+"</p>");
  document.write("<p>Para x=200°, y=120° => "+f2(toRad(200),
    toRad(120))+"</p>");
  document.write("<p>Para x=160°, y=125° => "+f2(toRad(160),
    toRad(125))+"</p>");
</script>
</head>
<body>
</body>
</html>
```

Que al ser abierto en el navegador devuelve, como era de esperar, los mismos resultados que en el tema anterior:

Ejemplo 1

Resultados de la primera expresión:

Para x=1.1, y=1.4, z=1.9 => 0.373295627961028

Para x=2.3, y=5.2, z=8.3 => 0.0502867873527808

Para x=1.9, y=1.2, z=3.1 => 0.4622452740293394

Resultados de la segunda expresión:

Para x=130°, y=200° => -1.0532480596708227

Para x=200°, y=120° => -0.5299699661565509

Para x=160°, y=125° => -1.7387698762266683

Observe, sin embargo, que incluso en un problema tan simple como este, el código resulta más compacto y claro que en el tema anterior.

2.2. EJERCICIOS

1. Cree una página HTML que, mediante una función, calcule el resultado de la siguiente expresión, para valores de "x" iguales a 2.3, 4,5 y 7.2.

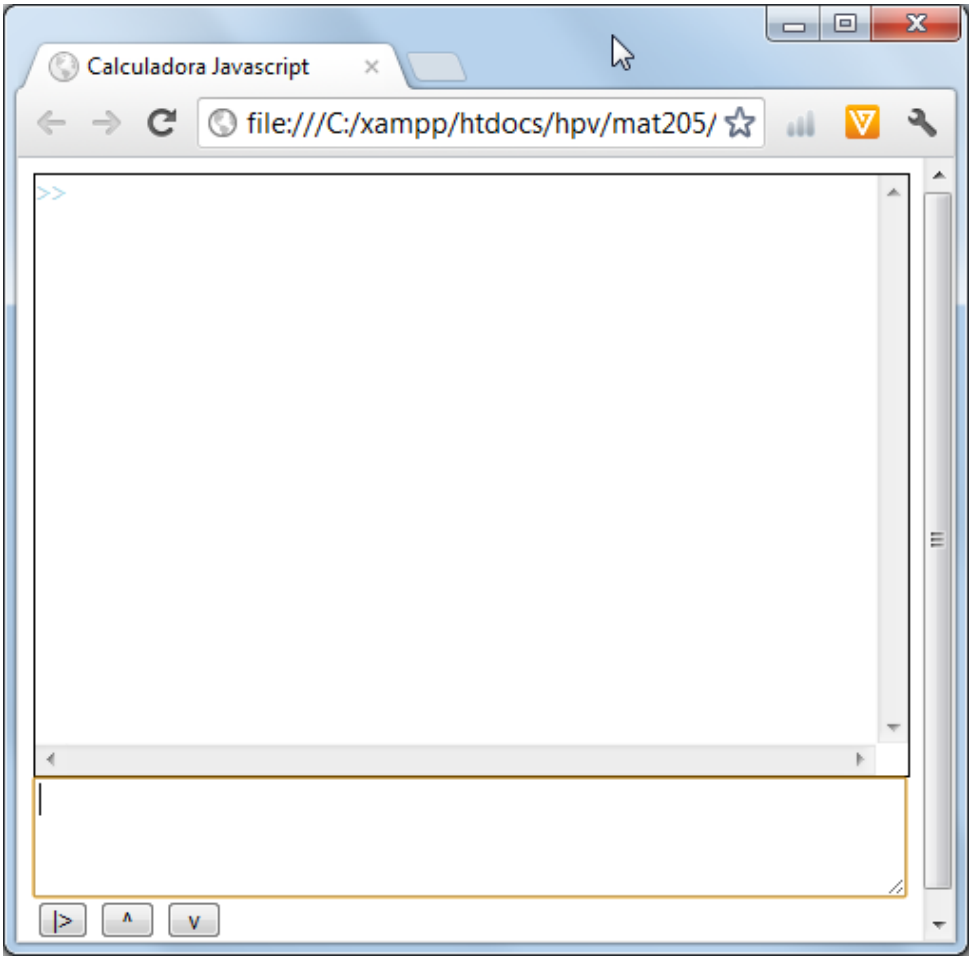
$$\sin^{-1}\left(\frac{\sqrt{x+2x^{0.3}+1.2}}{2x+0.5}\right)$$

2. Cree una página HTML, que mediante una función, calcule el resultado de la siguiente expresión, para los siguientes valores de "x", "y" y "z": x=2.2, y=3.2, z=4.5; x=3.1, y=5.2, z=1.3; x=8.2, y=6.1, z=4.6.

$$\log\left(\frac{x^{3.2}+y^{1.6}+e^{z}}{\sqrt[5]{x+y^{z}}}\right)$$

2.3. LA CALCULADORA JAVASCRIPT

Con el propósito de facilitar algunos cálculos rápidos, así como probar de manera inmediata operadores, instrucciones y otras características del lenguaje Javascript, se ha creado una página: "calculadora.html" (suministrada con el material del tema):



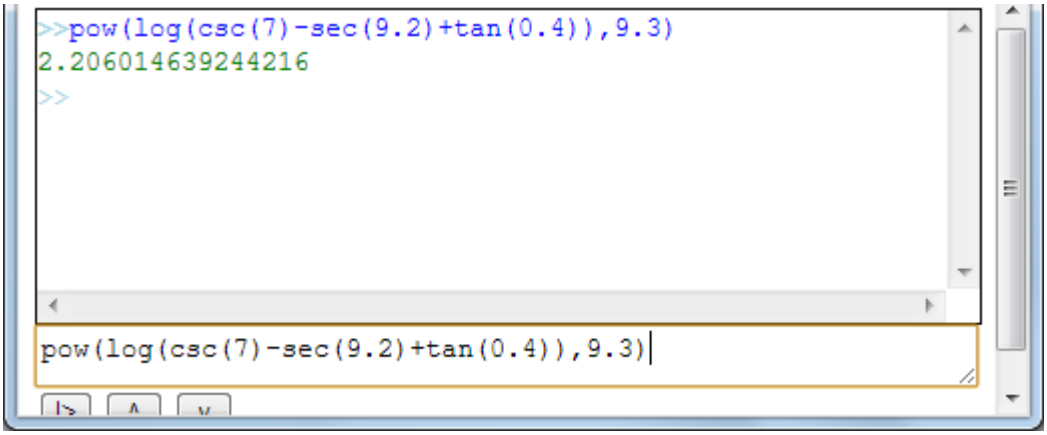
Esta página hace uso de la librería "mat205.js" y de la librería "diagram.js" (Javascript Diagram Builder: 3.3: www.lutanho.net/diagram/), modificada para adaptarla a los requerimientos de la materia (ambas librerías se proporcionan conjuntamente el material del tema).

Como toda página, para utilizarla simplemente se abre en un navegador (verificando que tanto "mat205.js" como "diagram.js" se encuentren en el mismo directorio que "calculadora.html").

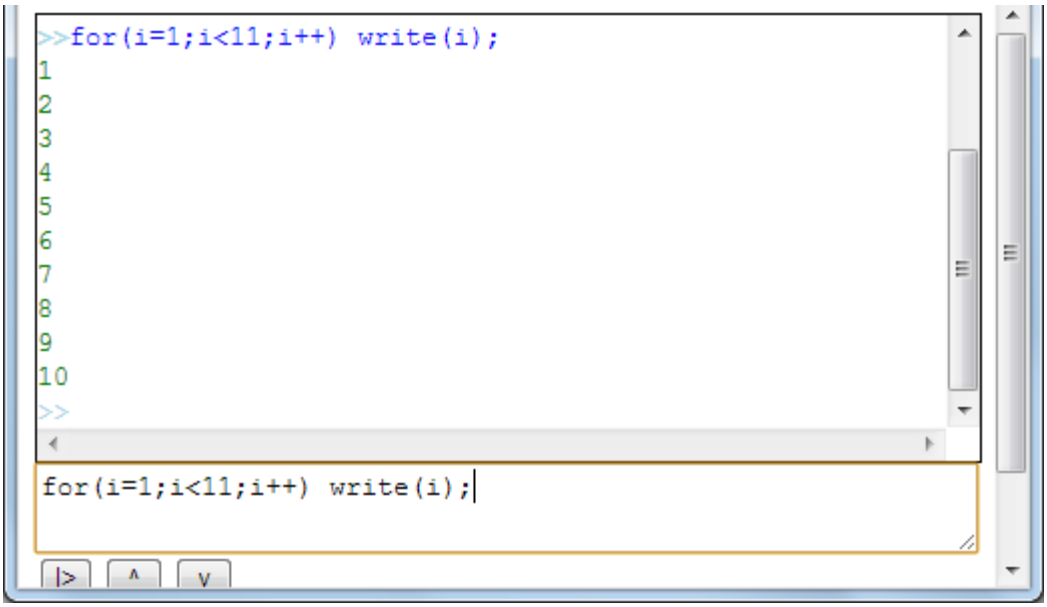
Para ejecutar una instrucción, simplemente se la escribe en el recuadro inferior y se pulsa la tecla "Enter" (o se hace click en el botón [|>]). Por ejemplo, para calcular el valor de la siguiente expresión:

$$\ln(\csc(7) - \sec(9.2) + \tan(0.4))^{9.3}$$

Se escribe la expresión en el recuadro inferior y al pulsar "Enter" se obtiene:



En la calculadora se pueden ejecutar prácticamente todas las instrucciones Javascript, así como las de la librería "mat205.js". Por ejemplo, se puede crear un ciclo "for" con contador que vaya del 1 al 10 y que en cada repetición del ciclo muestre el valor del contador:



La función "write" es una función propia de la calculadora (no de Javascript) y como se puede ver, sirve para mostrar en el recuadro de resultados la expresión que se escribe en ella.

La calculadora almacena las instrucciones a medida que se ejecutan y pueden ser recuperadas (para ser modificadas o vueltas a ejecutar) con las teclas de cursores, o con los botones [^] y [v]. En los celulares con sistema Android, el teclado **Multiling** cuenta con dichas teclas, mismas que funcionan correctamente en **UCBrowser** y **Firefox**.

En la calculadora se pueden crear funciones empleando la forma literal (no la forma estándar). Por ejemplo, en las siguientes instrucciones se crea una función para calcular el cubo de un número y luego se la emplea para calcular el cubo de 3 y de 7.89:

```
>>cubo = function(x){return x*x*x;}
function()
>>cubo(3)
27
>>cubo(7.89)
491.1690689999999
```

Sin embargo, ***las funciones creadas en la calculadora sólo existen mientras la página está abierta en el navegador*** y desaparecen tan pronto como se cierra. Por esa razón, en la calculadora sólo se deben crear funciones que no vayan a ser reutilizadas y/o que sean muy fáciles de programar.

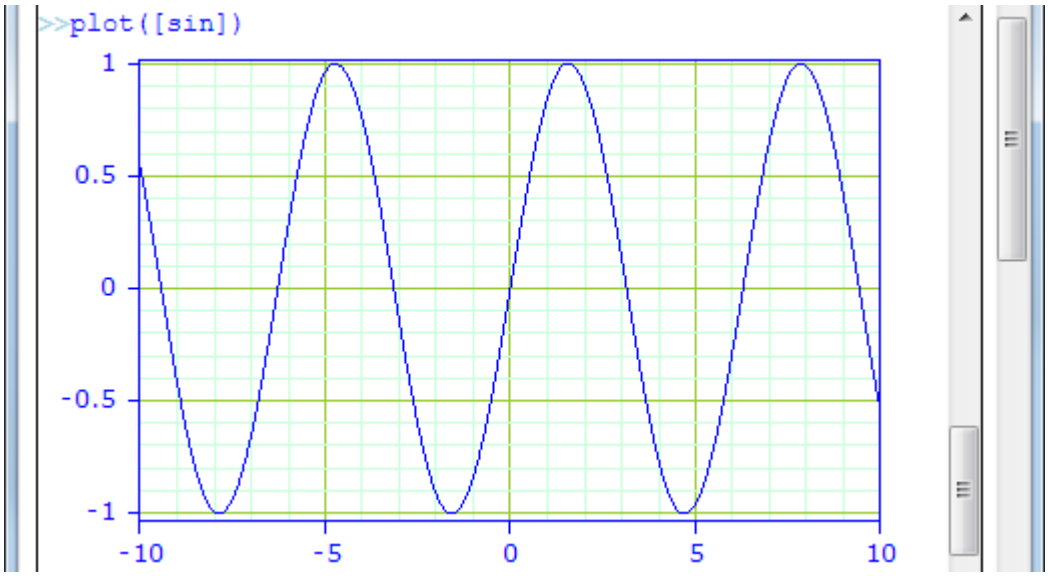
En las computadoras es posible insertar saltos de línea al escribir una función (u otra instrucción) con las teclas "Shift+Enter", no obstante, si la función es compleja, no es recomendable resolverla en la calculadora, sino más bien en una página HTML.

Por otra parte, a la librería "diagram.js" se le ha añadido la función "plot" (la misma que puede ser empleada también desde la calculadora):

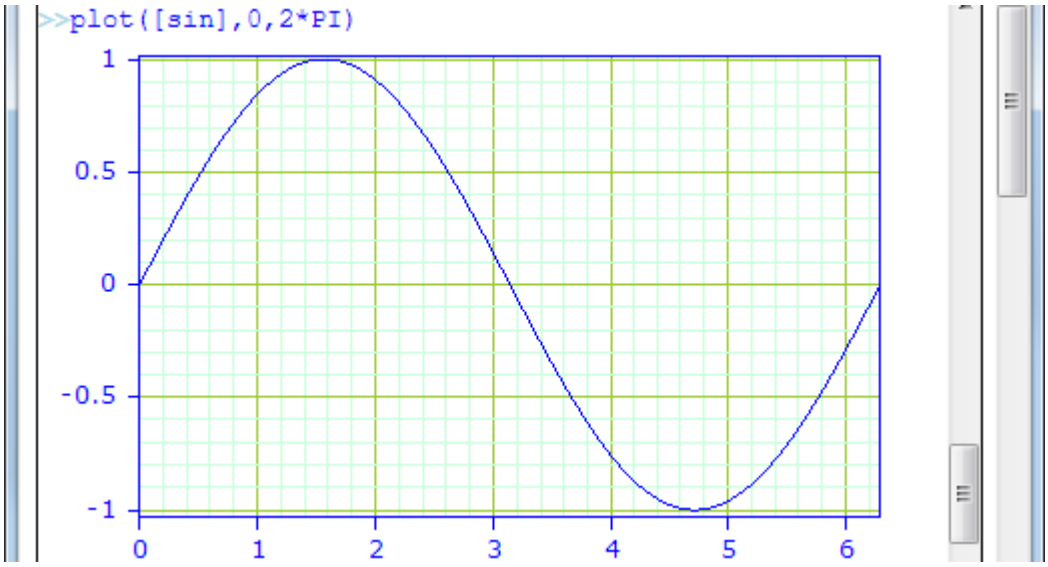
```
plot([lista_de_funciones],xi,xf,w,h,yi,yf);
```

Donde "lista_de_funciones" son los nombres de las funciones a graficar (separadas con comas), "xi" y "xf" son los límites inicial y final de la variable independiente (valores por defecto -10 y 10); "w" y "h" son el ancho y alto de la gráfica en pixeles (valores por defecto 420 y 240); "yi" y "yf" son los valores inicial y final de la variable dependiente (los valores por defecto se calculan automáticamente).

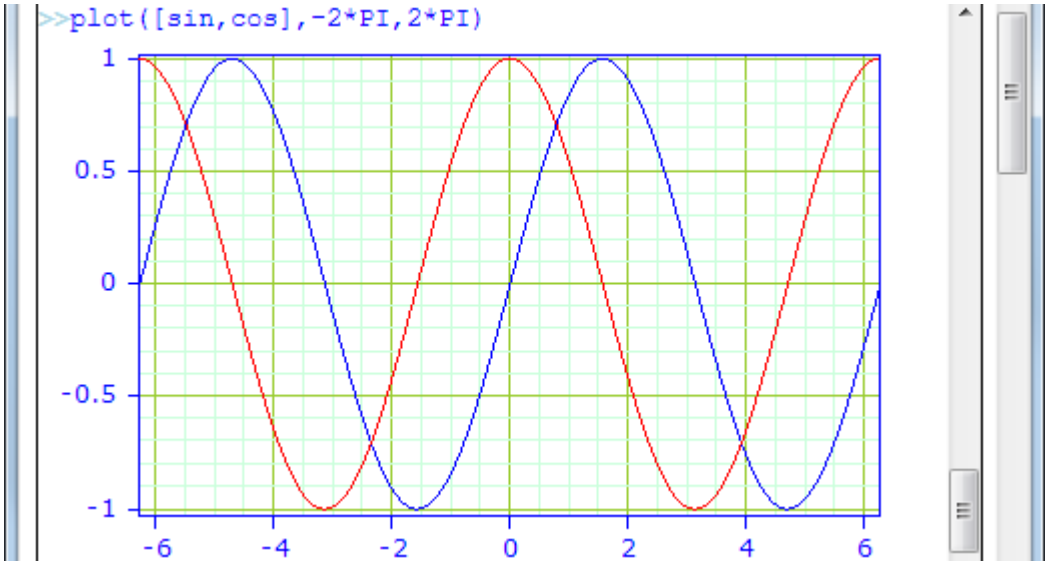
Por lo tanto sólo es imprescindible el nombre, o nombres, de las funciones, así para graficar la función "sin" se puede escribir:



Que es la gráfica de la función seno entre -10 y 10. En la práctica casi siempre se dan los límites de la variable independiente, así, con la siguiente instrucción se grafica la función seno entre 0 y 2π:



Y con la siguiente se grafican las funciones seno y coseno entre -2π y 2π.



Como se dijo, la utilidad principal de la calculadora es la de permitir realizar algunos cálculos rápidos, pero sobre todo probar, interactivamente, las estructuras, características y funciones del lenguaje y/o de las librerías.

Para probar las funciones de otra u otras librerías (aparte de "mat205.js" y "diagrama.js") simplemente se edita la página "calculadora.html" y se le añade las etiquetas "script" para importar las librerías correspondientes.

2.4. LAS CONSOLAS DE LOS NAVEGADORES

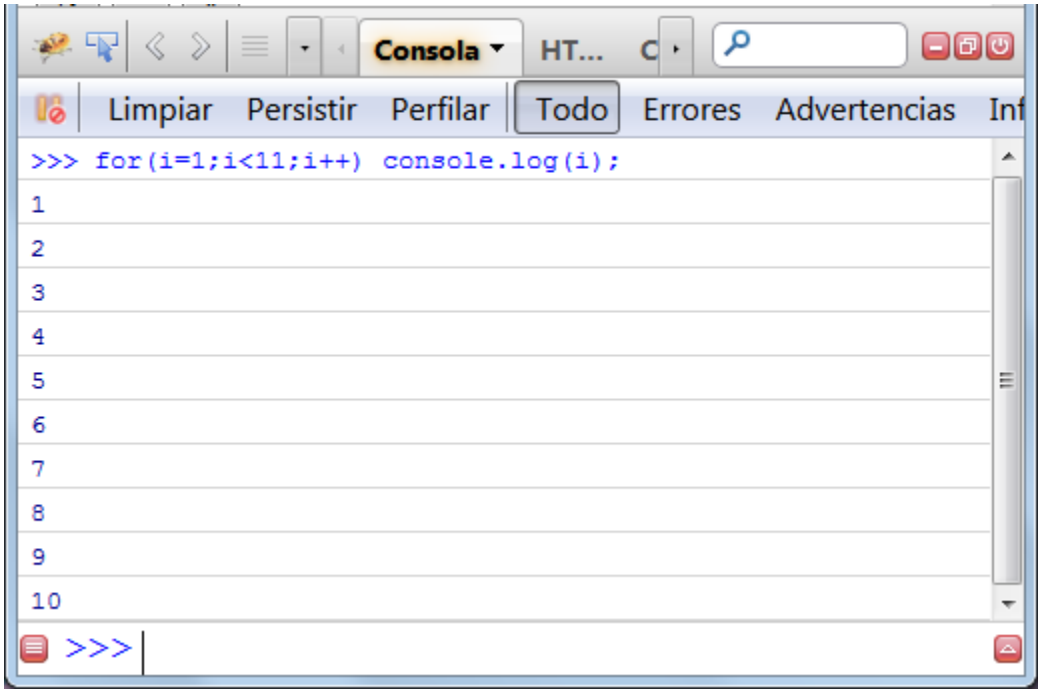
Actualmente, debido a la creciente importancia de Javascript en el desarrollo de aplicaciones, los navegadores modernos cuentan con consolas

para ayudar a desarrollar aplicaciones. Dichas consolas (entre otras cosas) permiten ejecutar instrucciones Javascript, de manera similar a como lo hace la calculadora.

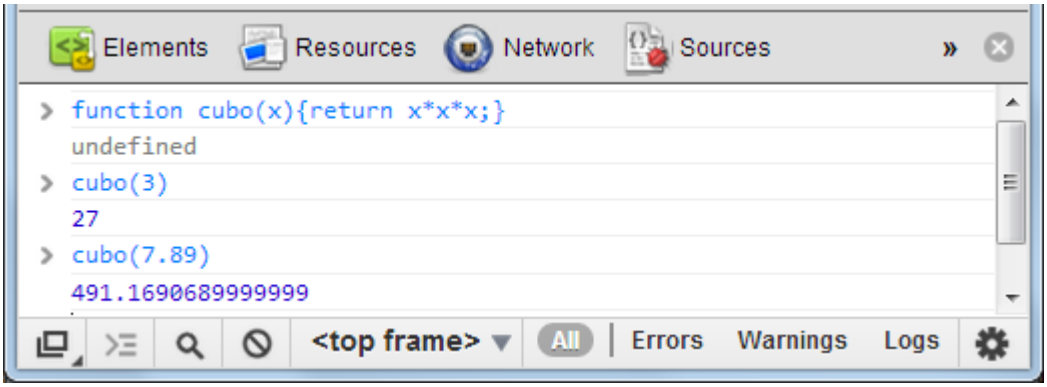
Para emplear la consola, simplemente se abre la página donde se encuentra el código Javascript y/o donde se han importado (con "script") las librerías Javascript. Entonces se accede a la consola pulsando la tecla F12 (si, en la ventana que aparece, no está por defecto la consola, se la abre haciendo clic en el menú que aparece).

Al igual que sucede con la calculadora, en las consolas se pueden probar prácticamente todas las características del lenguaje y/o librerías. Es posible también imprimir valores pero empleando la instrucción "console.log" en lugar de "write".

Por ejemplo la instrucción que imprime los números del 1 al 10, escrito en la consola de Mozilla Firefox, es:



También se pueden crear funciones, pero en las consolas es posible crearlas en la forma estándar, así, la función "cubo", creada como ejemplo en la calculadora, en Google Chrome se crea y utiliza de la siguiente forma:



Como en la calculadora, las funciones creadas en las consolas sólo existen mientras la página está abierta y dejan de existir tan pronto se cierra.

La mayoría de los navegadores, proporcionan, además de la consola, otras herramientas, tales como autocompletado, depuración, análisis de la estructura HTML, etc. Lamentablemente, no todos los navegadores proporcionan las mismas herramientas, es más, no todos los navegadores poseen estas herramientas.

Las herramientas más avanzadas las proporcionan los navegadores Google Chrome y Mozilla Firefox. En este último navegador, para acceder a la consola y las herramientas de depuración, se debe instalar el complemento "Firebug" (disponible en: http://getfirebug.com/), siendo conveniente instalar también "Acebug" (disponible en: https://addons.mozilla.org/en-US/firefox/addon/acebug/).

Si se dispone de conexión a Internet la instalación de estos complementos es automática, de no ser así, se deben bajar los complementos, guardarlos en un directorio e instalarlos de la siguiente forma: se abre el menú principal del navegador (botón del extremo superior izquierdo del navegador):

En el menú que aparece se elige la opción complementos. Dentro de complementos se hace clic en el botón:

Se elige la opción "Instalar complemento desde archivo...", se busca el archivo en el lugar donde fue guardado y se abre. De esa manera el complemento queda instalado.

2.5. EJERCICIOS

Los siguientes ejercicios deben ser resueltos en la calculadora Javascript (o en una de las consolas) y presentados en la misma, por lo que debe copiar las instrucciones que empleó para resolverlos en cualquier editor de texto y volverlos a copiar a la calculadora, uno por uno, al momento de la presentación.

3. Empleando la estructura "for" muestre los números del 20 al 1 (en orden descendente).

4. Grafique las funciones seno y coseno entre 0 y 4π.

2.6. LA FUNCIÓN MAP

Cuando se trabaja con series de más de 2 datos puede resultar de utilidad la función "map" (implementada en la librería "mat205.js"):

```
map(función, vector_1, vector_2, vector_3,…)
```

Donde vector_1, vector_2, etc., son los datos de los parámetros 1, 2, etc., de la "función". Map evalúa la "función" para cada uno de los elementos de los vectores devolviendo los resultados igualmente en un vector.

Por ejemplo, para calcular los valores del seno para: 0.2, 0.8, 1.2, 1.4, 1.8, 2.1, 2.5, 3.4, 3.6, 4.2, en lugar de evaluar 10 veces la función "sin", cambiando su valor en cada ejecución, se pueden calcular todos los resultados a la vez (como se puede probar en la calculadora):

```
>>map(sin,[0.2,0.8,1.2,1.4,1.8,2.1,2.5,3.4,3.6,4.2])
[0.19866933079506122, 0.7173560908995228,
0.9320390859672263, 0.9854497299884601,
0.9738476308781951, 0.8632093666488737,
0.5984721441039565, -0.2555411020268312,
-0.44252044329485246, -0.8715757724135882]
```

Empleando esta función, el ejemplo 1 puede ser resuelto con el siguiente código:

```
<head>
  <title>Ejemplo 1</title>
  <script type="text/javascript" src="mat205.js"></script>
  <script type="text/javascript">
    function f1(x,y,z){
      return log10(x+sqrt(y+z))/xroot(x+pow(y,z),5);
    }
    function f2(x,y){
      return log10((sin(x)+cos(y))/(3.4*tan(x+y)));
    }
    document.write("<h1>Ejemplo 2</h1>");
    document.write("<h2>Resultados de la primera expresión</h2>"+
      "<p>Para (1.1,1.4,1.3), (2.3,5.2,8.3), (1.9,1.2,3.1):</p>");
    document.write("<p>"+map(f1,[1.1,2.3,1.9],[1.4,5.2,1.2],
      [1.3,8.3,3.1])+"</p>");
    document.write("<h2>Resultados de la segunda expresión</h2>"+
      "<p>Para (130°,200°), (200°,120°), (160°,125°)</p>");
    document.write("<p>"+map(f2,map(toRad,[130,200,160]),
      map(toRad,[200,120,125]))+"</p>");
  </script>
</head>
<body>
</body>
</html>
```

Que al ser abierto en un navegador muestra los siguientes resultados:

Ejemplo 2

Resultados de la primera expresión

Para (1.1,1.4,1.3), (2.3,5.2,8.3), (1.9,1.2,3.1):

0.3606764637020599,0.0502867873527808,0.4622452740293394

Resultados de la segunda expresión

Para (130°,200°), (200°,120°), (160°,125°)

-1.0532480596708227,-0.5299699661565509,-1.7387698762266683

Y, como era de esperar, se obtienen los mismos resultados que en el ejemplo 1.

2.7. EJERCICIOS

5. Elabore una página HTML, que mediante una función y haciendo uso de "map", calcule el valor de la siguiente expresión para valores de "x", "y" y "z" iguales a: (2.2,3.2,4.5), (3.1,5.2,1.3) y (8.2,6.1,4.6).

$$f(x,y,z) = \log\left(\frac{x^{3.2} + y^{1.6} + e^{z}}{\sqrt[5]{x + y^{z}}}\right)$$

6. Elabore una página HTML, que mediante una función y haciendo uso de "map", calcule el valor de la siguiente expresión para valores de "x", "y" y "z" iguales a: (12.1,10.2,0.5), (6.7,8.2,0.7) y (8.2,9.5,0.2).

$$f(x,y,z) = \frac{\sinh^{-1}(x) + \cosh^{-1}(y) + \tanh^{-1}(z)}{x + y + z^{4.9} + 6!}$$

7. Elabore una página HTML, que mediante una función y haciendo uso de "map", calcule el valor de la siguiente expresión, para valores de "x", "y" y "z" iguales a: (21°,87°,35°), (67°,82°,70°) y (182°,195°,200°).

$$f(x,y,z) = \frac{\sqrt[5]{\sin(x) + \cos(y) + \tan(z)}}{\sqrt[3]{e^{\sin(x+y)} + e^{\cos(z)}}}$$

3. ECUACIONES ALGEBRAICAS I

En este tema comienza el estudio de los métodos que permiten resolver ecuaciones algebraicas con una incógnita, es decir, encontrar el valor de la variable independiente que satisface la igualdad representada por la expresión matemática.

En general existen dos formas en las que se expresa dicha igualdad:

$$x=g(x) \tag{3.1}$$

Y

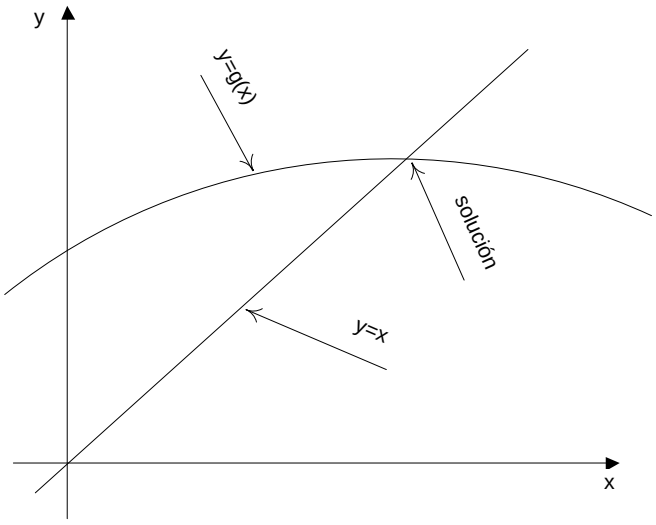
$$f(x)=0 \tag{3.2}$$

En consecuencia, para resolver una ecuación se puede despejar una de las incógnitas e igualarla al resto de la ecuación, o agrupar todos los miembros en un lado de la ecuación e igualarla a cero.

En este tema se estudia el primer método para resolver ecuaciones que se encuentran (o colocan) en la primera forma.

3.1. SOLUCIÓN GRÁFICA

Gráficamente la solución de una ecuación que está en la forma (3.1), se encuentra en la intersección de la curva "y=g(x)" con la recta "y=x", es decir:



Así por ejemplo para encontrar gráficamente la solución aproximada de la siguiente ecuación:

$$f(x)=\left(52+3\sqrt{x}-8x^{0.8}\right)^{0.36}-x=0 \tag{3.3}$$

Se la coloca primero en la forma x=g(x):

$$x=\left(52+3\sqrt{x}-8x^{0.8}\right)^{0.36}=g(x) \tag{3.4}$$

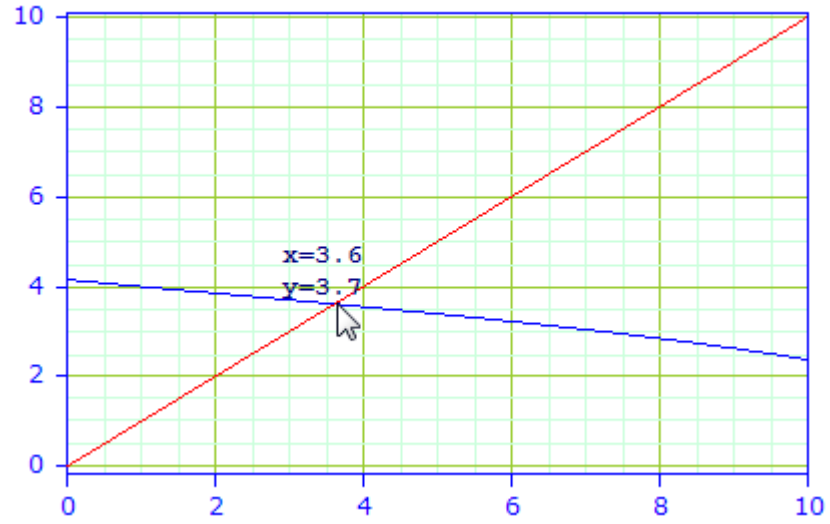
Para ver la solución gráfica en la calculadora, se la programa conjuntamente la recta y=x:

```
>>f1=function(x){return pow(52+3*sqrt(x)-8*pow(x,0.8),0.36);}
function()
```

```
>>f0=function(x){return x;}
function()
```

Y se grafican entre dos límites dados (en este ejemplo entre 0 y 10):

```
>>plot([f1,f0],0,10)
```



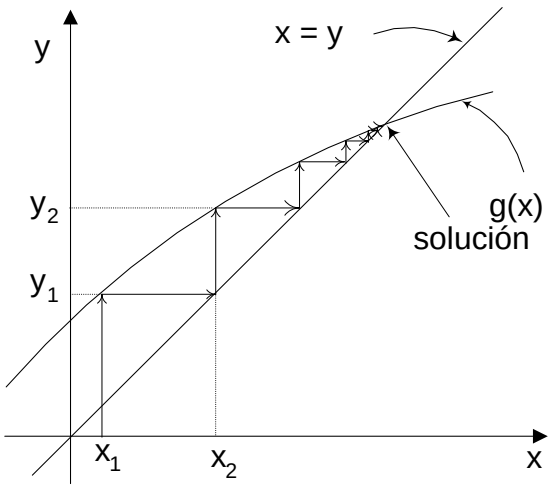
Por tanto la solución gráfica (el valor de "x") es aproximadamente igual a 3.6, es decir que al sustituir este valor en la ecuación el resultado debe ser aproximadamente el mismo valor (3.6):

```
>>f1(3.6)
3.611149591389499
```

Y como se puede ver así ocurre (al sustituir 3.6 en la ecuación se obtiene 3.61). Pero como sólo se trata de una solución aproximada, no se obtiene exactamente el mismo valor, siendo esta es una característica de las soluciones gráficas: son sólo aproximadas. Por esta razón, dichas soluciones, sólo se emplean como valores iniciales de los métodos numéricos.

3.2. MÉTODO DE SUSTITUCIÓN DIRECTA

Ahora se pasa a estudiar el método más sencillo que existe para resolver ecuaciones de la forma (3.1), el método de sustitución directa, que gráficamente es:



Es decir, con el valor asumido "x_1" se calcula el valor de la función "y_1" (gráficamente en la intersección con la curva y numéricamente sustituyendo "x1" en la función), luego "y_1" se convierte en el nuevo valor de prueba "x2" (gráficamente intersectando con la recta "x=y"). Con "x_2" se repite el proceso obteniendo "y_2" y con "y_2" "x_3", continuando así hasta que los valores de "xn" y "y_n" son aproximadamente iguales, lo que ocurre cuando la curva (x=g(x)) y la recta (y=x) intersectan.

Por lo tanto el proceso a seguir es: a) Se asume un valor inicial "x_1"; b) Con "x_1" se calcula el valor de la función "y_1=g(x_1)"; c) Se compara "y_1" con "x_1" y si son aproximadamente iguales el proceso concluye, sino, "x_1" toma el valor de "y_1" y el proceso se repite desde el paso (b).

Por ejemplo, para encontrar la solución numérica de la ecuación (3.3), se asume un valor inicial, como ya se tiene una solución aproximada (la solución gráfica), lo lógico sería tomar esa solución como valor inicial. En este ejemplo se toma un valor diferente (1.1), sólo para ilustrar la forma en que el método se acerca (converge) hacia la solución:

```
>>x1=1.1
1.1
```

Con este valor se calcula el valor de la función:

```
>>y1=f1(x1)
3.9840553877884246
```

Como el valor asumido (1.1) y el calculado (3.9840553877884246) no son iguales, el proceso se repite, haciendo que "x1" tome el valor de "y1" y calculando con el mismo el valor de la función:

```
>>x1=y1;y1=f1(x1)
3.552012132021903
```

Una vez más el valor asumido (3.9840553877884246) y el calculado (3.552012132021903) no son iguales, por lo que el proceso se repite (simplemente recuperando la instrucción anterior y volviendo a ejecutarla):

```
>>x1=y1;y1=f1(x1)
3.618479599247315
```

Como se puede ver los valores asumido y calculado se van acercando, pero aún son diferentes, por lo que se debe repetir el proceso hasta que se logre la igualdad:

```
>>x1=y1;y1=f1(x1)
3.6083235654919994
>>x1=y1;y1=f1(x1)
3.609876925738287
>>x1=y1;y1=f1(x1)
3.6096399376571533
>>x1=y1;y1=f1(x1)
3.6096757048676587
>>x1=y1;y1=f1(x1)
3.609670149216255
>>x1=y1;y1=f1(x1)
3.6096709988371494
>>x1=y1;y1=f1(x1)
3.6096708689053827
>>x1=y1;y1=f1(x1)
3.609670888775732
```

```
>>x1=y1;y1=f1(x1)
3.6096708857369775
>>x1=y1;y1=f1(x1)
3.6096708862016915
>>x1=y1;y1=f1(x1)
3.609670886130623
>>x1=y1;y1=f1(x1)
3.6096708861414912
>>x1=y1;y1=f1(x1)
3.60967088613983
>>x1=y1;y1=f1(x1)
3.6096708861400835
>>x1=y1;y1=f1(x1)
3.609670886140045
>>x1=y1;y1=f1(x1)
3.6096708861400506
>>x1=y1;y1=f1(x1)
3.60967088614005
>>x1=y1;y1=f1(x1)
3.6096708861400497
>>x1=y1;y1=f1(x1)
3.6096708861400497
```

En esta última repetición los valores asumido y calculado se igualan, por lo que el proceso concluye, siendo la solución x=3.6096708861400497.

Cuando, como en este ejemplo, el método se acerca hacia la solución en cada repetición, se dice que es **convergente**. Por el contrario cuando se aleja de la solución se dice que es **divergente**. Si fluctúa, acercándose y alejándose de la solución, se dice que es **oscilatorio**.

Un método es estable si converge en la mayoría de los casos, e inestable si no lo hace. El método de sustitución directa es un método inestable porque diverge y/u oscila en la mayoría de los casos.

Si bien el método es muy sencillo y es posible aplicarlo manualmente (simplemente recuperando la orden y volviendo a ejecutarla) son necesarias muchas repeticiones, por lo que es conveniente automatizar el proceso a través de un programa.

3.2.1. Algoritmo y código

Si bien el algoritmo básico es suficiente en algunos casos, en general es necesario considera que: a) Debido a errores de redondeo, no siempre es posible lograr que los valores asumido y calculado sean iguales, por lo que el proceso debe repetirse sólo hasta se logre la igualdad en un determinado número de dígitos (precisión); b) En los métodos iterativos la convergencia no está garantizada (y mucho menos en métodos inestables como el presente), por lo que se debe establecer un límite de repeticiones para evitar que el proceso se repita indefinidamente.

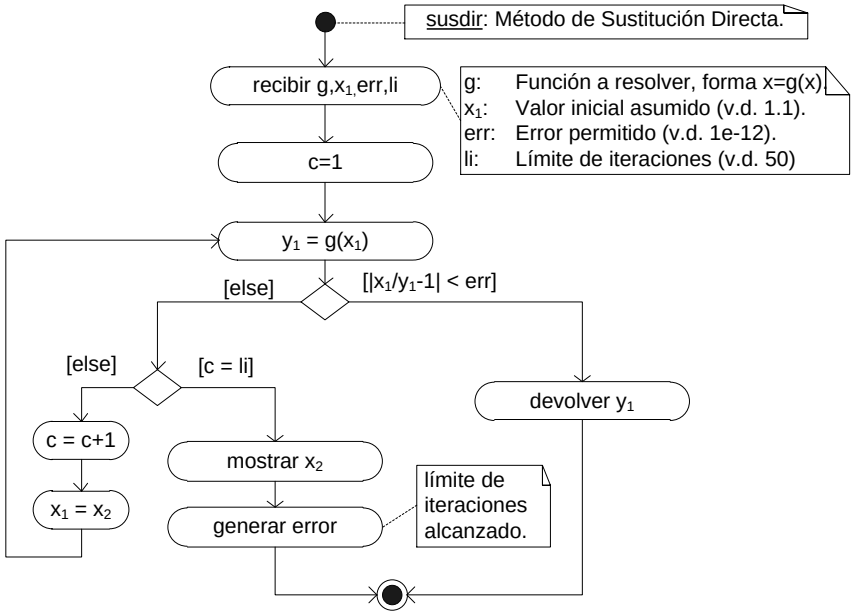
Para determinar si dos números "x_1" y "x_2" son iguales en un determinado número de dígitos se evalúa la expresión:

$$\left|\frac{x_1}{x_2} - 1\right| < 1 \times 10^{-n} \qquad (3.5)$$

Que es verdadera cuando los valores de "x_1" y "x_2" son iguales en por lo menos "n" dígitos.

Para establecer un límite de repeticiones, se debe incluir un contador que al llegar al límite establecido termine el proceso generando un error.

El algoritmo que toma en cuenta las anteriores consideraciones es:



En este algoritmo "v.d." son los valores por defecto, es decir los valores que toman los parámetros cuando no se le manda el valor respectivo. En Javascript se sabe que un parámetro no ha recibido un valor cuando es igual a "null", entonces para averiguar si un parámetro no recibió un valor y de ser así asignarle un valor por defecto, simplemente se compara con "null". Por ejemplo en la siguiente instrucción se averigua si el parámetro "x" recibió un valor y de no ser así se le asigna un valor por defecto igual a 3.23.

```
if (x==null) x=3.23;
```

Por otra parte el ciclo repetitivo del algoritmo corresponde a un ciclo infinito (no tiene condición ni al principio ni al final), por lo tanto se sale del mismo al cumplirse una de las dos condiciones, en la primera devolviendo el resultado con "return" y en la segunda con "throw". Con estas consideraciones el código para el método de Sustitución Directa es:

```
function susdir(g,x1,err,li){
  if (x1==null) x1=1.1;
  if (err==null) err=1e-12;
  if (li==null) li=50;
  var y1,c=1;
  while (true){
    y1=g(x1);
    if (abs(x1/y1-1)<err) return y1;
    if (c++ == li) throw new Error("susdir no converge, x="+y1);
    x1=y1;
  }
}
```

Esta función ha sido añadida a la librería "mat205.js", por lo que puede ser empleada en la calculadora y en cualquier página que incluya la

incluya. El único argumento (dato) realmente imprescindible es el nombre de la función que se quiere resolver, sin embargo, casi siempre se manda también el valor inicial asumido.

Si no se logra convergencia se puede cambiar el valor asumido, incrementar el error, incrementar el límite de iteraciones o despejar otra variable de la expresión matemática.

Si se quiere cambiar uno de los valores por defecto y mantener los otros, se debe mandar la palabra "null" para aquellos valores por defecto que se quiere conservar. Por ejemplo, para resolver la función "g" con un límite de 250 iteraciones, sin cambiar los otros valores por defecto, se escribe:

```
susdir(g,null,null,250)
```

Para resolver la ecuación (3.3), en la calculadora, empleando los valores por defecto, se escribe:

```
>>g=function(x){return pow(52+3*sqrt(x)-8*pow(x,0.8),0.36)}
function()
>>susdir(g)
3.60967088613983
```

Que tiene un precisión de 12 dígitos. Para ver sólo los 12 dígitos que son confiables en este resultado, se puede emplear la función "precision":

```
>>precision(ans,12)
3.60967088614
```

Donde "ans" es la variable que guarda el último resultado calculado (el penúltimo resultado se guarda en res[1], el antepenúltimo en res[2] y así sucesivamente), se aclara, sin embargo, que estas variables están disponibles únicamente en la calculadora. En las consolas se tiene que copiar los resultados o emplear variables auxiliares para guardarlos.

Si se emplea un valor asumido igual a 1.1 y un error igual a 1e-16, se obtiene el mismo resultado que en el proceso manual:

```
>>susdir(g,null,1e-16)
3.6096708861400497
```

Observe que en este caso se ha mandado "null" como segundo argumento. Esto debido a que el valor asumido es igual al valor por defecto.

Con un valor inicial igual a 12.1, un error igual a 1e-14 y un límite de 12 iteraciones se obtiene:

```
>>susdir(g,12.1,1e-14,12)
err: susdir no converge, x=3.6096708882576096
```

En este caso se genera un error porque el método no logra convergencia en el límite de iteraciones dado, incrementando dicho límite a 20, se obtiene la solución correcta (en otros casos es necesario cambiar también el valor asumido):

```
>>susdir(g,12.1,1e-14,20)
3.6096708861400457
```

Que en los 14 dígitos es que es preciso el resultado es:

```
>>precision(ans,14)
3.60967088614
```

También es posible redondear el resultado con "round", así, el anterior

resultado redondeado a 9 dígitos después del punto es:

```
>>round(res[1],9)
3.609670886
```

3.3. EJEMPLOS

1. **Encuentre las soluciones aproximadas de la siguiente función graficándola entre x=0 y x=10. Luego empleando estas soluciones como valores asumidos, encuentre soluciones más precisas, con 6 dígitos de precisión, aplicando manualmente el método de sustitución directa.**

$$f(x) = \cos(x) + (1 + x^2)^{-1} = 0 \tag{3.6}$$

Primero ser lleva la ecuación a la forma "x=g(x)" y la manera más sencilla de hacerlo es sumar "x" a ambos lados de la ecuación:

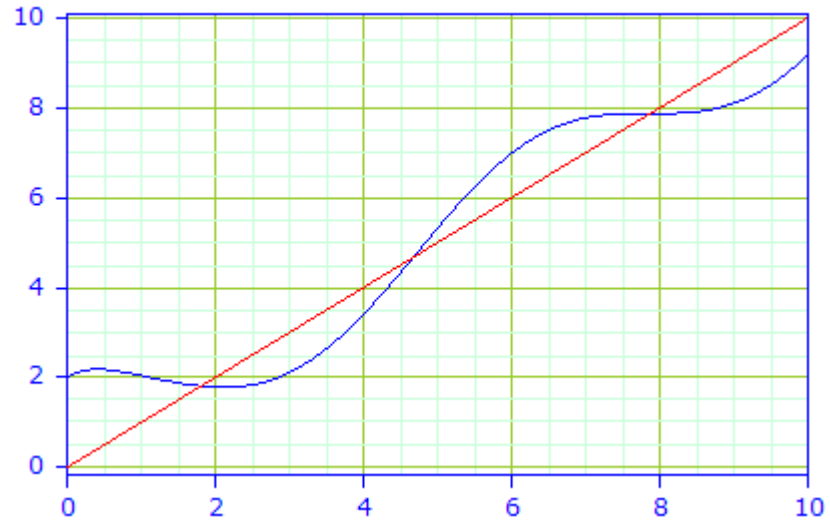
$$0 + x = x = g(x) = \cos(x) + (1 + x^2)^{-1} + x \tag{3.7}$$

Se programa la función y la recta "y=x":

```
>>g1=function(x){return cos(x)+1/(1+sqr(x))+x;}
function()
>>y1=function(x){return x;}
function()
```

Y se elabora la gráfica:

```
>>plot([g1,y1],0,10)
```



Y como se puede ver existen tres soluciones (tres intersecciones) que ocurren aproximadamente en 1.7, 4.6 y 7.9. Tomando estos valores, como valores iniciales, se encuentran soluciones más precisas (con 6 dígitos de precisión) aplicando manualmente el método de sustitución directa.

Para la primera raíz se obtiene:

```
>>x1=1.7; y1=g1(x1); precision(y1,6)
1.82822
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80392
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80797
```

```
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80727
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80739
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80737
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80738
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80738
```

Por lo tanto la primera solución, con 6 dígitos de precisión, es 1.80738. Para la segunda solución, se obtiene:

```
>>x1=4.6; y1=g1(x1); precision(y1,6)
4.53297
>>x1=y1; y1=g1(x1); precision(y1,6)
4.40093
>>x1=y1; y1=g1(x1); precision(y1,6)
4.14357
>>x1=y1; y1=g1(x1); precision(y1,6)
3.65998
>>x1=y1; y1=g1(x1); precision(y1,6)
2.86083
>>x1=y1; y1=g1(x1); precision(y1,6)
2.00886
>>x1=y1; y1=g1(x1); precision(y1,6)
1.78326
>>x1=y1; y1=g1(x1); precision(y1,6)
1.81162
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80665
>>x1=y1; y1=g1(x1); precision(y1,6)
1.8075
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80735
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80738
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80737
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80738
>>x1=y1; y1=g1(x1); precision(y1,6)
1.80738
```

En este caso el método no converge hacia la segunda solución, sino que vuelve a converger a la primera, algo que no es raro en un método inestable como el presente. Para la tercera solución se obtiene:

```
>>x1=7.9; y1=g1(x1); precision(y1,6)
7.86977
>>x1=y1; y1=g1(x1); precision(y1,6)
7.86987
>>x1=y1; y1=g1(x1); precision(y1,6)
7.86987
```

En este caso el método converge a la solución en tan solo 3 iteraciones, por lo tanto la tercera solución, con 6 dígitos de precisión es: 7.86987.

2. Encuentre las soluciones gráficas de la siguiente función (entre x=1.5 y x=2). Luego con las soluciones aproximadas, encuentre soluciones más precisas, con 9 dígitos de precisión, con "susdir".

$$\begin{aligned} 3x^{2.1} - 5y &= 7.0 \\ y^{1.2} + 4z &= 14.3 \\ x + y^2 + z^2 &= 14.0 \end{aligned} \tag{3.8}$$

Sistemas de ecuaciones como este pueden ser resueltos si es posible colocarlos como una función de una variable (en este caso "x"):

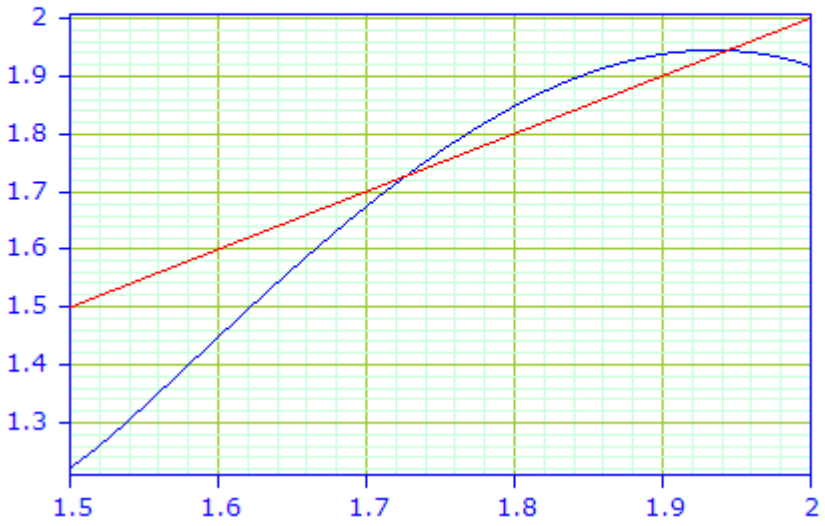
$$f(x) = 14.0 - y^2 - z^2 - x = 0$$
$$y = \frac{3x^{2.1} - 7.0}{5}$$
$$z = \frac{14.3 - y^{1.2}}{4}$$

Ahora esta función, a pesar de su apariencia, es simplemente una función de una variable, la misma que puede ser llevada a la forma "x=g(x)" (sumando o restando "x" a ambos lados de la ecuación):

$$\begin{aligned} x = g(x) &= 14.0 - y^2 - z^2 \\ y &= \frac{3x^{2.1} - 7.0}{5} \\ z &= \frac{14.3 - y^{1.2}}{4} \end{aligned} \tag{3.9}$$

Entonces se programa la función, la recta "y=x" y se grafican:

```
>>g2=function(x){var y=(3*pow(x,2.1)-7)/5; z=(14.3-pow(y,1.2))/4;
return 14-sqr(y)-sqr(z);}
function()
>>y2=function(x){return x;}
function()
>>plot([g2,y2],1.5,2)
```



Como se puede ver, ahora existen dos soluciones (dos raíces), siendo sus valores aproximadas (obtenidos haciendo clic en las intersecciones): 1.7 y 1.9.

Empleando 1.7, como valor asumido, se obtiene:

```
>>precision(susdir(g2,1.7,1e-9),9)
err: susdir no converge, x=NaN
```

Por lo tanto, para esta solución el método de sustitución directa no logra convergencia. El valor de "x" es "NaN" (no es un número), valor que se genera como consecuencia de una operación inválida (como una división entre cero). Como se dijo este método es inestable, por lo tanto no es raro que con frecuencia no logre convergencia.

Empleando 1.9, como valor asumido, se obtiene:

```
>>precision(susdir(g2,1.9,1e-9),9)
1.94379708
```

En este caso el método logra convergencia, siendo esta la segunda solución (segunda raíz) con 9 dígitos de precisión.

3.4. EJERCICIOS

Como en el anterior tema para presentar los ejercicios en la calculadora, copie las instrucciones en un editor de texto y luego, al momento de presentar el trabajo, vuelva a copiarlos (y ejecutarlos) en la calculadora.

1. Encuentre la solución aproximada de la siguiente función, graficándola entre x=-5 y x=1. Luego, empleando la solución aproximadas como valor asumido, encuentre una solución más precisa, con 7 dígitos de precisión, aplicando manualmente el método de sustitución directa.

$$f(x) = x^3 + 2x^2 + 3x + 4 = 0$$

2. Encuentre las soluciones aproximadas de la siguiente función, graficándola entre x=-1 y x=2. Luego, empleando las soluciones aproximadas como valores asumidos, encuentre soluciones más precisas, con 5 dígitos de precisión, aplicando manualmente el método de sustitución directa.

$$f(x) = 2 * x^2 + 1 - \exp(x) = 0$$

3. Encuentre las soluciones aproximadas de la siguiente función, graficándola entre z=-10 y z=10. Luego, empleando las soluciones aproximadas como valores asumidos, encuentre soluciones más precisas, con 6 dígitos de precisión, aplicando manualmente el método de sustitución directa.

$$f(z) = e^{z/2} + z^2 - 60 = 0$$

4. Encuentre las soluciones aproximadas del siguiente sistema de ecuaciones, graficándola entre x=1 y x=10. Luego, empleando las soluciones aproximadas como valores asumidos, encuentre soluciones más precisas, con 10 dígitos de precisión, con la función "susdir" (recuerde colocar el sistema como una función de una variable, la variable "x").

$$2x^2 + 5xy - 4x = 115$$
$$e^{\frac{x+y}{5}} + x^2y^2 - 70y = 15$$

5. Encuentre las soluciones aproximadas del siguiente sistema de ecuaciones, graficándola entre x=12 y x=13. Luego, emplee las soluciones aproximadas como valores asumidos, para encontrar soluciones más

precisas, con 14 dígitos de precisión, con la función "susdir" (recuerde colocar el sistema de ecuaciones como una función de una variable, la variable "x").

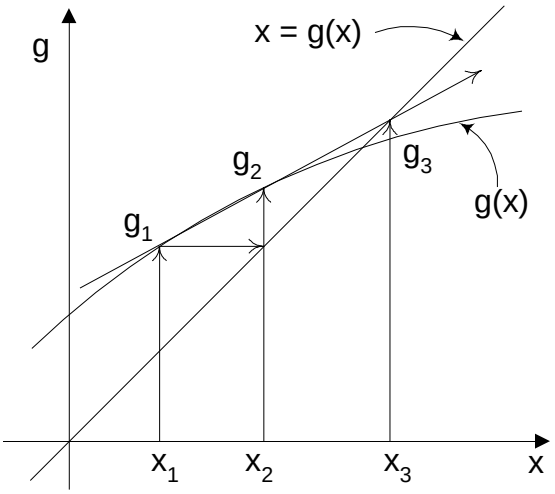
$$15y + 20x^2 - x^3 = 1500$$
$$9z - 1.5x^2 + e^{x/2} = 300$$
$$y^2 - z^3 - 5x = 166.81$$

4. ECUACIONES ALGEBRAICAS II

Una vez estudiado los principios en los que se basa la solución de ecuaciones algebraicas con una incógnita (con el método más sencillo disponible), en este tema se estudian otros métodos, más estables que el método de sustitución directa, aunque no tan sencillos.

4.1. MÉTODO DE WEGSTEIN

El método de Wegstein acelera y mejora la estabilidad del método de Sustitución Directa, extrapolando y/o interpolando linealmente hacia la solución, tal como se muestra en la figura:



Los dos puntos con que inicia el proceso ((g1,x1) y (g2,x2)), se obtienen aplicando el método de sustitución directa, es decir haciendo que: g1=g(x1), x2=g1, g2=g(x2).

Numéricamente el valor extrapolado "x_3" se calcula tomando en cuenta la ecuación de la línea recta:

$$g = a + bx$$

La pendiente de la recta que pasa a través de los dos puntos es:

$$g_1 = a + bx_1$$
$$g_2 = a + bx_2$$
$$b = \frac{g_1 - g_2}{x_1 - x_2}$$

Y la ordenada en el origen:

$$g_1 = a + bx_1 = a + \frac{g_1 - g_2}{x_1 - x_2} x_1$$
$$a = g_1 - \frac{g_1 x_1 - g_2 x_1}{x_1 - x_2}$$

$$a = \frac{g_2 x_1 - g_1 x_2}{x_1 - x_2}$$

Por lo tanto, la ecuación de la línea recta que pasa a través de los puntos (g1,x1), (g2,x2) es:

$$g = \frac{g_2 x_1 - g_1 x_2}{x_1 - x_2} + \frac{g_1 - g_2}{x_1 - x_2} x$$

En la intersección con la recta "x=y", "*x3*" y "*g3*" son iguales:

$$g_3 = x_3 = \frac{g_2 x_1 - g_1 x_2}{x_1 - x_2} + \frac{g_1 - g_2}{x_1 - x_2} x_3$$

$$x_3 \left(1 - \frac{g_1 - g_2}{x_1 - x_2}\right) = \frac{g_2 x_1 - g_1 x_2}{x_1 - x_2}$$

$$x_3 \left(\frac{x_1 - x_2 - g_1 + g_2}{x_1 - x_2}\right) = \frac{g_2 x_1 - g_1 x_2}{x_1 - x_2}$$

$$x_3 = \frac{g_2 x_1 - g_1 x_2}{g_2 + x_1 - g_1 - x_2}$$

Esta ecuación es conocida como la ecuación de Wegstein.

Al igual que con el método de sustitución directa, el método de Wegstein se repite hasta que "x3" y "g3" son iguales, sólo que ahora, antes de repetir el proceso se realizan los siguientes intercambios de variables: x1=x2, x2=x3, g1=g2 y g2=g3.

Si bien el método de Wegstein acelera el proceso de convergencia y mejora la estabilidad, tiene el inconveniente de generar errores de redondeo (debido a las restas del denominador de la ecuación) por lo que su precisión es limitada.

Para comprender mejor el método se volverá a resolver la siguiente ecuación:

$$x = \left(52 + 3\sqrt{x} - 8x^{0.8}\right)^{0.36} = g(x)$$

Para ello (y como es usual) se programa la función:

```
g=function(x){
  return pow(52+3*sqrt(x)-8*pow(x,0.8),0.36);
}
```

Entonces se asume un valor inicial (en este ejemplo 1.1) y con el mismo se calculan los dos primeros puntos:

```
>>x1=1.1; g1=g(x1); x2=g1; g2=g(x2)
3.5520121320219027
```

Ahora, con estos puntos, se puede calcular el nuevo valor de "x" (x3) aplicando la ecuación del método:

```
>>x3=(g2*x1-g1*x2)/(g2+x1-g1-x2)
3.6083015838499186
```

Y se calcula el valor de la función para este nuevo valor de "x":

```
>>g3=g(x3)
3.609880287208415
```

Estos valores son iguales en los 3 primeros dígitos, por lo que inclusive en esta primera iteración se logra ya un resultado preciso en 3 dígitos (lo que constituye una mejora considerable con relación al método de sustitución directa).

Continuando con el proceso, para encontrar un resultado con mayor precisión, se realiza un cambio de variables y se calcula un nuevo valor de "x":

```
>>x1=x2; x2=x3; g1=g2; g2=g3; x3=(g2*x1-g1*x2)/(g2+x1-g1-x2)
3.6096696044950036
```

Y con el mismo se calcula el valor respectivo de la función:

```
>>g3=g(x3)
3.6096710821408835
```

Ahora "x3" y "g3" son iguales en 6 dígitos (redondeados), por lo tanto el resultado ya es preciso en 6 dígitos. Repitiendo el proceso una vez más se obtiene:

```
>>x1=x2; x2=x3; g1=g2; g2=g3; x3=(g2*x1-g1*x2)/(g2+x1-g1-x2)
3.6096708861436144
>>g3=g(x3)
3.609670886139505
```

Ahora "x3" y "g3" son iguales en los primeros 12 dígitos (redondeados), por lo que el resultado es preciso en 12 dígitos, sin embargo, si se repite el proceso unas cuantas veces más (buscando mayor precisión) se obtiene:

```
>>x1=x2; x2=x3; g1=g2; g2=g3; x3=(g2*x1-g1*x2)/(g2+x1-g1-x2)
3.609670885332564
>>g3=g(x3)
3.609670886263538
>>x1=x2; x2=x3; g1=g2; g2=g3; x3=(g2*x1-g1*x2)/(g2+x1-g1-x2)
3.60967144609595
>>g3=g(x3)
3.609670800506491
>>x1=x2; x2=x3; g1=g2; g2=g3; x3=(g2*x1-g1*x2)/(g2+x1-g1-x2)
3.6096708873982237
>>g3=g(x3)
3.6096708859476383
>>x1=x2; x2=x3; g1=g2; g2=g3; x3=(g2*x1-g1*x2)/(g2+x1-g1-x2)
3.609670885320682
>>g3=g(x3)
3.6096708862653553
>>x1=x2; x2=x3; g1=g2; g2=g3; x3=(g2*x1-g1*x2)/(g2+x1-g1-x2)
3.6096701969948333
>>g3=g(x3)
3.609670991530414
>>x1=x2; x2=x3; g1=g2; g2=g3; x3=(g2*x1-g1*x2)/(g2+x1-g1-x2)
3.6096708850568073
>>g3=g(x3)
3.609670886305709
```

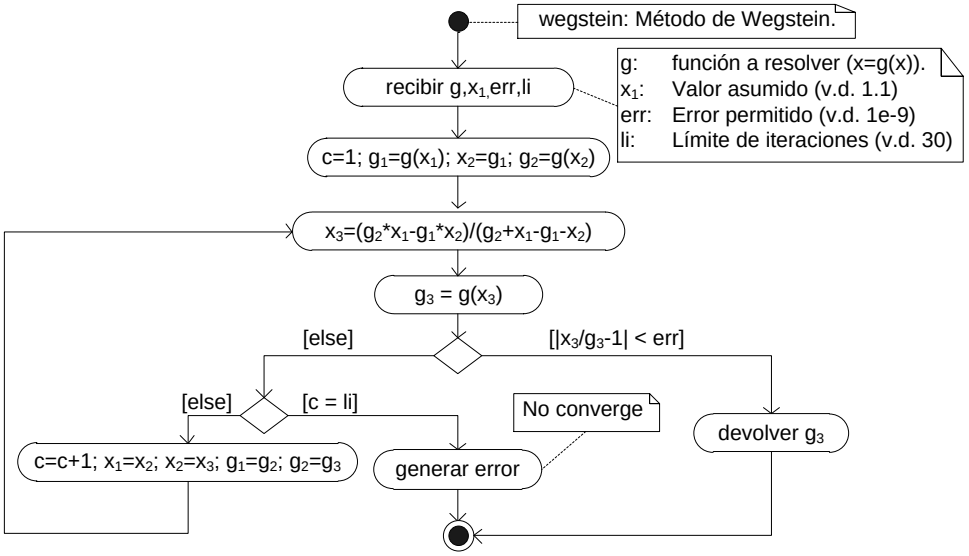
Se comprueba que la precisión no incrementa, sino que por el contrario disminuye a 6 dígitos y llega como máximo a 9 dígitos de precisión. Este

comportamiento oscilatorio se repite indefinidamente y los valores asumido y calculado nunca llegan igualarse. Como se dijo, esto se debe a las restas del denominador de la ecuación de Wegstein, las cuales (cuando los valores asumido y calculado son cercanos) dan lugar a valores cercanos a cero con la consiguiente pérdida de dígitos de precisión, por esta razón este método sólo permite encontrar resultados con una precisión limitada, normalmente unos 9 dígitos de precisión.

4.1.1. Algoritmo y código

El algoritmo básico puede ser deducido fácilmente del procedimiento seguido en el anterior acápite. Sin embargo, y como ya se señaló en el anterior tema, es necesario fijar una precisión (un error) y un límite de iteraciones.

El algoritmo es el siguiente:



Siendo el código respectivo (añadido a la librería "mat205.js"):

```
function wegstein(g,x1,err,li){
  if (x1==null) x1=1.1;
  if (err==null) err=1e-9;
  if (li==null) li=30;
  var c=1,x2,x3,g1,g2,g3;
  g1=g(x1);x2=g1;g2=g(x2);
  while (true) {
    x3=(g2*x1-g1*x2)/(g2+x1-g1-x2);
    g3=g(x3);
    if (abs(x3/g3-1)<err) return g3;
    if (c++ == li) throw new Error("wegstein no converge, x="+g3);
    x1=x2;x2=x3;g1=g2;g2=g3;
  }
}
```

Como se puede ver, por defecto el método encuentra el resultado con 9 dígitos de precisión, un límite de 30 iteraciones y un valor inicial igual a 1.1. Empleando esta función con la ecuación resuelta previamente, se obtiene:

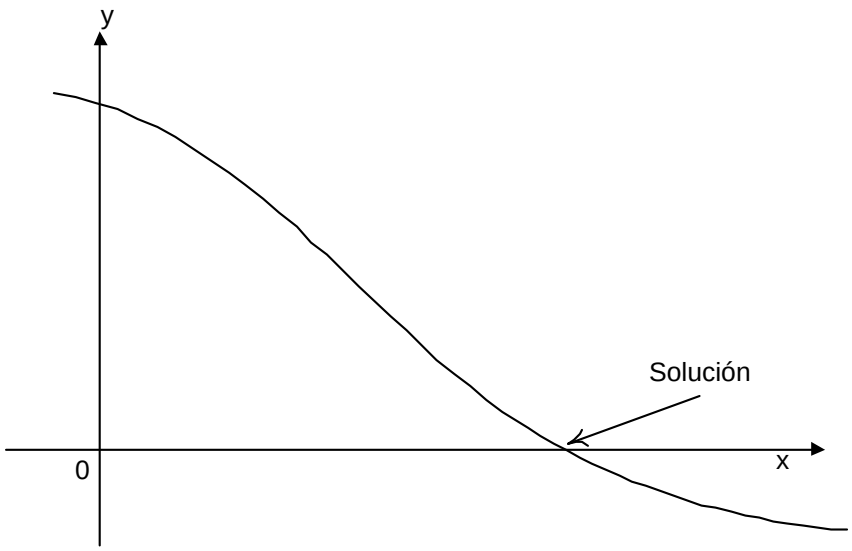
```
>>wegstein(g)
3.60967088613935
```

Resultado que es confiable sólo en los primeros 9 dígitos, es decir:

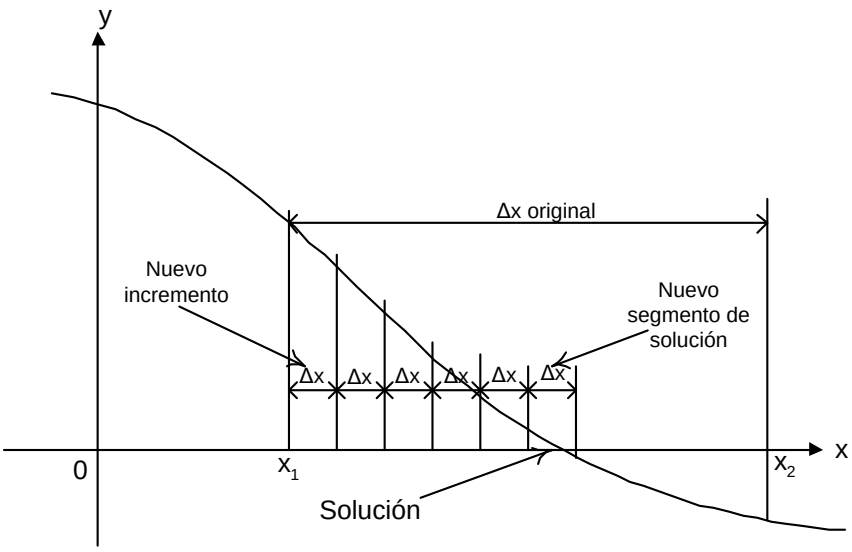
```
>>precision(wegstein(g),9)
3.60967089
```

4.2. MÉTODO INCREMENTAL

El método incremental es, conceptualmente, el método más sencillo que existe para resolver ecuaciones que se encuentran en la forma f(x)=0. En esta forma, la solución gráfica se encuentra en la intersección de la función con la recta "y=0":



Como a ambos lados de la solución la función cambia de signo, el método incremental, aprovecha ese hecho para encontrar el lugar donde ocurre dicho cambio. Gráficamente sigue el procedimiento que se muestra en la figura:



Comenzando con el valor asumido "x_1" y el incremento inicial "Δx", se calculan valores sucesivos de "x" ($x=x+\Delta x$) y los respectivos valores de

la función (f(x)), hasta que ocurre un cambio de signo. Cuando ello ocurre se sabe que se ha pasado a través de una solución y que por lo tanto la misma se encuentra entre los dos últimos valores. Son esos valores, el segmento de solución, lo que se busca en este método.

Una vez encontrado el segmento de solución, se puede dividir el mismo entre 10 ($\Delta x/10$) y encontrar un segmento de solución más pequeño y este procedimiento puede repetirse hasta que el segmento sea lo suficientemente pequeño como para constituir una solución, sin embargo y como ya se dijo, ello requiere muchas iteraciones, por lo que generalmente sólo se encuentra el primer segmento de solución y luego se encuentra la solución recurriendo a otros métodos más eficientes.

Por ejemplo, para encontrar el segmento de solución de la ecuación:

$$f(x) = \left(52 + 3\sqrt{x} - 8x^{0.8}\right)^{0.36} - x = 0$$

Se crea la función, pero ahora igualándola a 0 (no a "x").

```
f=function(x){
  return pow(52+3*sqrt(x)-8*pow(x,0.8),0.36)-x;
}
```

Se asume un valor inicial (en este ejemplo 1.1), un incremento inicial (en este ejemplo 0.1) y se calcula el valor de la función:

```
>>x=1.1; dx=0.1; f(x)
2.8840553877884245
```

Entonces se incrementa el valor de "x" y se calcula el valor de la función, repitiendo el proceso hasta que ocurre un cambio de signo:

```
>>x+=dx; f(x)
2.7691274777568307
>>x+=dx; f(x)
2.654242296560439
>>x+=dx; f(x)
2.53939166098368
>>x+=dx; f(x)
2.424568232957627
>>x+=dx; f(x)
2.3097653851731876
>>x+=dx; f(x)
2.1949770907864843
>>x+=dx; f(x)
2.0801978326955783
>>x+=dx; f(x)
1.9654225285726117
>>x+=dx; f(x)
1.8506464685571395
>>x+=dx; f(x)
1.7358652631442468
>>x+=dx; f(x)
1.6210747993137486
>>x+=dx; f(x)
1.5062712033539727
>>x+=dx; f(x)
```

```
1.3914508091529303
>>x+=dx; f(x)
1.276610130979083
>>x+=dx; f(x)
1.1617458399686025
>>x+=dx; f(x)
1.0468547436884825
>>x+=dx; f(x)
0.9319337682646118
>>x+=dx; f(x)
0.8169799426585072
>>x+=dx; f(x)
0.7019903847514222
>>x+=dx; f(x)
0.5869622889544344
>>x+=dx; f(x)
0.4718929151110802
>>x+=dx; f(x)
0.3567795784978256
>>x+=dx; f(x)
0.24161964075898457
>>x+=dx; f(x)
0.12641050163820466
>>x+=dx; f(x)
0.011149591389496116
>>x+=dx; f(x)
-0.10416563623210928
```

Entonces se sabe que la solución se encuentra entre los dos últimos valores de "x", es decir "x-dx" y "x":

```
>>write(x-dx); x
3.6000000000000023
3.7000000000000024
```

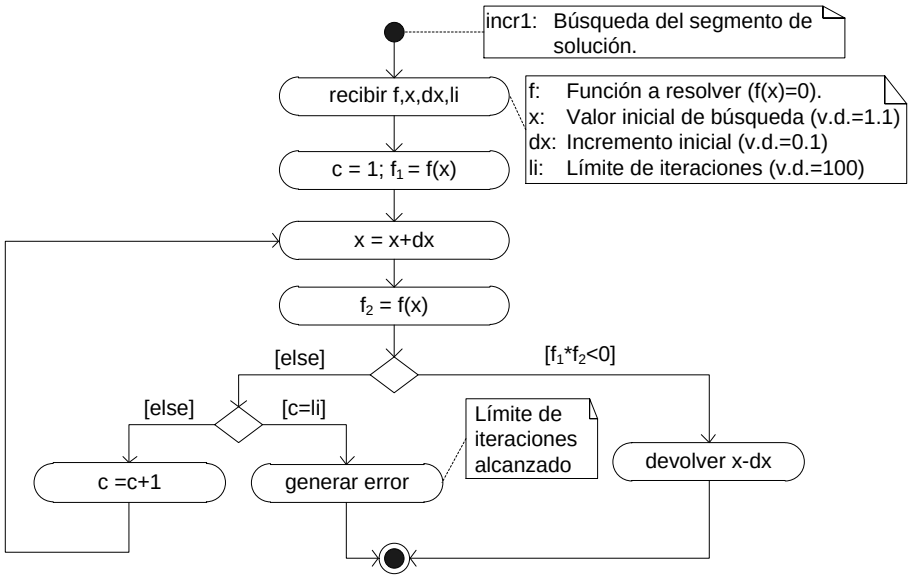
Por lo tanto la solución se encuentra entre 3.6 y 3.7 (los dígitos después de los ceros se deben a errores de redondeo).

4.2.1. Algoritmo y código

Como en los otros métodos, se debe fijar un límite de iteraciones, porque puede suceder que no exista una raíz en el lugar donde se realiza la búsqueda, o que el incremento sea muy pequeño y no se llegue al cambio de signo o que por el contrario sea muy grande y pase a través de dos soluciones (con lo que no se produce un cambio de signo).

El algoritmo, que busca el segmento de solución, se se presenta en la siguiente página y el código respectivo, añadido a la unidad "mat205.js", es el siguiente:

```
function incr1(f,x,dx,li){
  if (x==null) x=1.1;
  if (dx==null) dx=0.1;
  if (li==null) li=100;
  var c=1,f1=f(x),f2;
```




```
  while (true) {
    x+=dx;
    f2=f(x);
    if (f1*f2<0) return x-dx;
    if (c++ == li) throw new Error("incr1 no converge, x="+x);
  }
}
```

Como se puede ver, esta función devuelve el valor inicial del segmento de solución. El segundo valor se obtiene simplemente sumando "dx" al mismo.

Empleando este programa con la ecuación resuelta manualmente (con los valores por defecto) se obtiene:

```
>>incr1(f)
3.6000000000000023
```

Como el incremento por defecto es 0.1, se sabe que la solución se encuentra entre 3.6 y 3.6+0.1=3.7.

Como ya se dijo, una vez encontrado el segmento de solución, generalmente se encuentra una solución más precisa recurriendo a otros métodos, sin embargo, y como también se explicó, es posible encontrar una solución aproximada volviendo a aplicar el método con incrementos cada vez más pequeños (iguales a la décima parte del incremento anterior).

Así, partiendo del segmento encontrado con el incremento original, se puede encontrar un nuevo segmento:

```
>>dx=0.1/10; x=incr1(f,3.6,dx)
3.6
```

Que aparentemente es igual al anterior, pero que en realidad tiene un dígito más de precisión (por lo tanto ahora se sabe que la solución se encuentra entre 3.60 y 3.61).

Empleando este nuevo valor (almacenado en la variable "x") y la décima parte de este incremento, se encuentra un resultado que tiene un dígito más de precisión:

```
>>dx/=10; x=incr1(f,x,dx)
3.608999999999999
```

Por lo tanto ahora se sabe que la solución se encuentra entre 3.609 y 3.610). Prosiguiendo de esta manera se incrementa la precisión del segmento en un dígito en cada repetición:

```
>>dx/=10; x=incr1(f,x,dx)
3.6096000000000004
>>dx/=10; x=incr1(f,x,dx)
3.609670000000001
>>dx/=10; x=incr1(f,x,dx)
3.609670000000001
>>dx/=10; x=incr1(f,x,dx)
3.6096707999999995
>>dx/=10; x=incr1(f,x,dx)
3.609670879999999
>>dx/=10; x=incr1(f,x,dx)
3.6096708859999995
>>dx/=10; x=incr1(f,x,dx)
3.6096708860999995
>>dx/=10; x=incr1(f,x,dx)
3.6096708861399995
>>dx/=10; x=incr1(f,x,dx)
3.6096708861399995
>>dx/=10; x=incr1(f,x,dx)
3.6096708861399995
>>dx/=10; x=incr1(f,x,dx)
3.6096708861400404
>>dx/=10; x=incr1(f,x,dx)
3.6096708861400493
```

En esta última repetición el valor del incremento "dx" es:

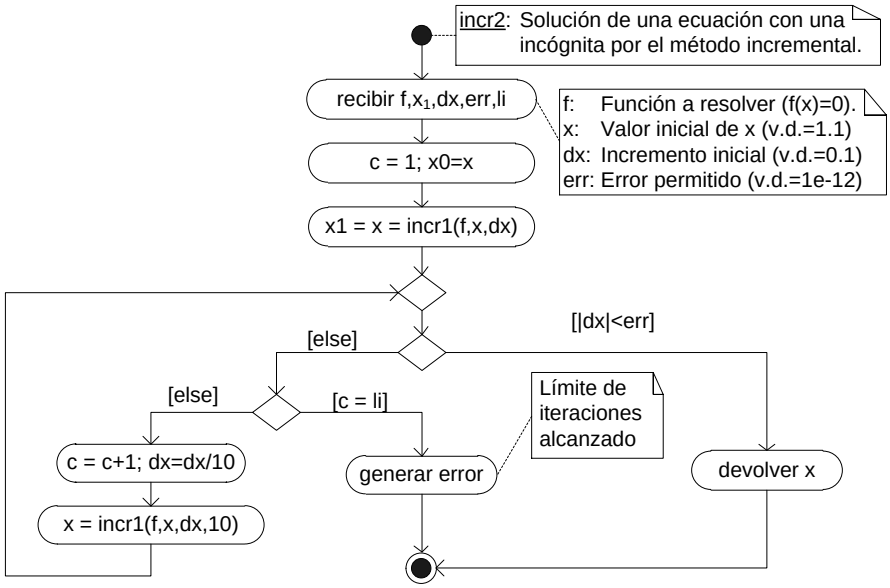
```
>>dx
1e-15
```

Por lo que en este punto se puede afirmar que la solución se encuentra entre 3.609670886140049 y 3.609670886140050, por lo tanto, se puede asegurar que la solución (con 15 dígitos de precisión) es: 3.60967088614005.

No es posible encontrar una solución con mayor precisión porque la precisión de los números reales en Javascript es de 16 dígitos, que es la precisión que ya tiene el segmento de solución (aunque por supuesto, la solución tiene un dígito de precisión menos).

El algoritmo que automatiza este proceso se presenta en la siguiente página y el código respectivo, añadido a la librería "mat205.js", es:

```
function incr2(f,x,dx,err){
  if (x==null) x=1.1;
  if (dx==null) dx=0.1;
  if (err==null) err=1e-12;
  var c=1;
  x=incr1(f,x,dx);
  while (true) {
    if (abs(dx)<err) return x;
```




```
        if (c++ == 16) throw new Error("incr2 no converge, x="+x);
        dx/=10;
        x=incr1(f,x,dx,10);
    }
}
```

Al emplear este programa se debe recordar que el error permitido corresponde a los dígitos que son precisos después del punto, así si el error es igual a 1e-7, el resultado es preciso hasta el sexto dígito después del punto.

Resolviendo la ecuación del ejemplo manual con este programa (empleando los valores por defecto) se obtiene:

```
>>incr2(f)
3.6096708861400018
```

Y como se ha empleado el error por defecto (1e-12), este resultado es confiable hasta el décimo primer dígito después del punto, es decir:

```
>>round(incr2(f),11)
3.60967088614
```

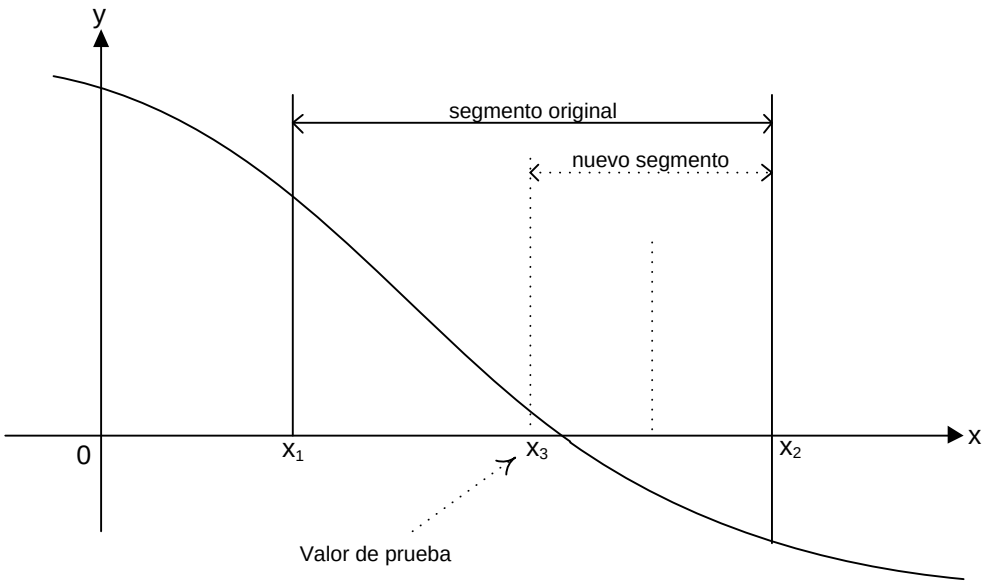
Donde la función "round" es la función que redondea un número, al número de dígitos especificado después del punto.

4.3. MÉTODO DE LA BISECCIÓN (BOLTZANO)

El método de la Bisección, también conocido como método de Boltzano, busca la solución a partir de un segmento de solución conocido. Para ello simplemente comprueba si la solución se encuentra a la mitad del segmento y de ser así el proceso concluye, caso contrario continúa la búsqueda en la mitad del segmento donde ocurre el cambio de signo, prosiguiendo de esa manera hasta que el valor de la función es cero o cercano a cero.

Gráficamente el método sigue el proceso que se muestran en la figura de la siguiente página. Matemáticamente, el lugar de la solución se determina multiplicando los valores de la función en "x_1" ($f(x_1)$) y "x_3" ($f(x_3)$),

si es negativa, la solución se encuentra en la mitad izquierda (entre x_1 y x_3), caso contrario se encuentra en la mitad derecha (entre x_3 y x_2).



La ecuación para el cálculo del nuevo valor de "x" ("x_3") se deduce a partir de la gráfica:

$$x_3 = \frac{x_2 - x_1}{2} + x_1$$

$$x_3 = \frac{x_2 - x_1 + 2x_1}{2}$$

$$x_3 = \frac{x_1 + x_2}{2}$$

Para comprender mejor el proceso, se resolverá (una vez más) la ecuación:

$$f(x) = \left(52 + 3\sqrt{x} - 8x^{0.8}\right)^{0.36} - x = 0$$

Primero se crea una función para la expresión (recuerde que en estos métodos las funciones se igualan a cero):

```
f=function(x){
  return pow(52+3*sqrt(x)-8*pow(x,0.8),0.36)-x;
}
```

El segmento de solución puede ser encontrado con "incr1" (o también gráficamente):

```
>>x1=incr1(f)
3.6000000000000023
```

Por lo tanto la solución se encuentra entre 3.6 y 3.7 (recuerde que el incremento por defecto en "incr1" es 0.1). El valor de la función en este punto es:

```
>>y1=f(x1)
0.011149591389496116
```

Con el segmento de solución (los valores de "x_1" y "x_2") se calcula el nuevo valor de prueba "x_3":

```
>>x2=x1+0.1; x3=(x1+x2)/2
3.650000000000002
```

Siendo el valor de la función en este nuevo punto:

```
>>y3=f(x3)
-0.046501074346197324
```

Como el valor de la función de error no es aún cero y ha cambiado de signo (con relación a "y_1"), se sabe que la solución se encuentra al lado izquierdo del segmento, por lo tanto "x_2" toma el valor de "x_3". Como en este método el nuevo valor de prueba "x_3" es asignado a "x_2" o a "x_1" dependiendo de que exista o no un cambio de signo, se puede emplear la estructura "if-else" para realizar dicha asignación y calcular un nuevo punto:

```
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.625000000000002  -0.017674024220964757
```

Entonces el proceso se repite hasta que el valor de la función ("y3") es cero (o cercano a cero) y/o hasta que el nuevo valor de prueba ("x3") es igual al anterior (en un determinado número de dígitos):

```
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6125000000000025  -0.0032617895746791525
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6062500000000024  0.003944007308300357
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6093750000000027  0.0003411355057147958
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6109375000000026  -0.0014603203699219414
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.610156250000003  -0.0005595907665680855
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6097656250000028  -0.0001092272141183237
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6095703125000025  0.00011595424986632352
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609667968750003  0.000003363543891854448
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609716796875003  -0.00005293182860910406
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609692382812503  -0.00002478414073259216 4
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096801757812527  -0.000010710298013805186
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096740722656278  -0.000003673376959500 9846
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096710205078155  -1.5491650851018335e-7
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609669494628909  0.000001604313697889 3813
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096702575683626  7.24698595799822e-7
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
```

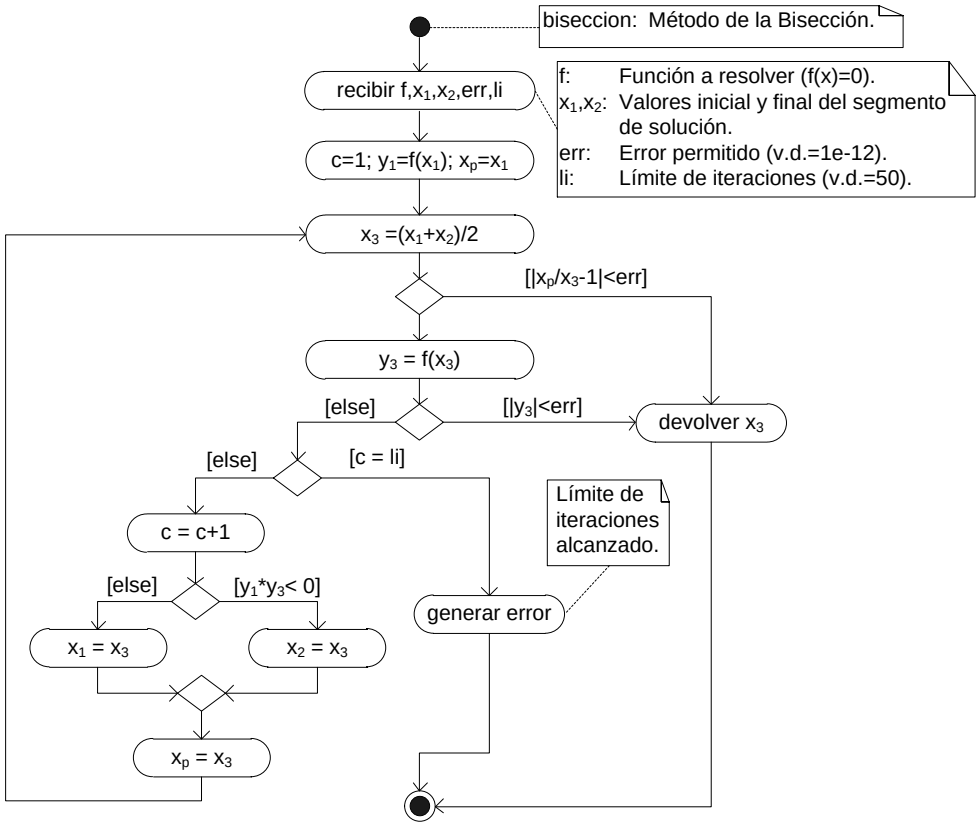
```
3.609670639038089  2.8489104453299774e-7
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670829772952  6.49872680114072e-8
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670925140384  -4.496462047143268e-8
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708774566677  1.0011324214076467e-8
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670901298526  -1.7476648128678107e-8
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670889377597  -3.732662179345425e-9
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708834171323  3.139331017365521e-9
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708863973648  -2.9666580303455703e-10
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708849072483  1.4213332732992967e-9
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708856523065  5.623337351323698e-10
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708860248357  1.3283374400430148e-10
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708862111004  -8.19162515597327e-11
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670886117968  2.545919031149424e-11
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670886164534  -2.822853062411923e-11
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670886141251  -1.3846701563124952e-12
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670886129609  1.2037482122195797e-11
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708861354303  5.325961893731801e-12
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670886138341  1.970423824104728e-12
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708861397957  2.9309887850104133e-13
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670886140523  -5.45785638905727e-13
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708861401594  -1.2612133559741778e-13
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708861399778  8.304468224196171e-14
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708861400684  -2.1316282072803006e-14
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670886140023  3.108624468950438e-14
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708861400457  4.884981308350689e-15
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670886140057  -7.993605777301127e-15
```

```
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708861400515  -1.7763568394002505e-15
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.609670886140049  1.3322676295501878e-15
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.60967088614005  -4.440892098500626e-16
>>if(y1*y3<0) x2=x3; else x1=x3; x3=(x1+x2)/2; y3=f(x3); write(x3,y3)
3.6096708861400497  0
```

Como se puede ver, para llegar a la solución exacta (para que la función se haga cero), se requieren 45 iteraciones, que son aún muchas iteraciones pero que no son ni la mitad de las iteraciones que se requieren con el método incremental.

4.3.1. Algoritmo y código

El algoritmo que automatiza el proceso es el siguiente:



Se hacen las mismas consideraciones que en los métodos anteriores. El proceso concluye cuando: a) La función es casi cero (que sería la condición natural del método) o b) Los valores de "x_3" de dos iteraciones consecutivas son casi iguales (que es cuando se alcanza la precisión deseada).

El código respectivo, añadido a la librería "mat205.js", es:

```
function biseccion(f,x1,x2,err,li) {
  if (err==null) {err=1e-12;}
  if (li==null) {li=50;}
  var c=1,y1=f(x1),xp=x1,x3,y3;
```

```
    while (true) {
      x3=(x1+x2)/2;
      if (abs(xp/x3-1)<err) {return x3;}
      y3=f(x3);
      if (abs(y3)<err) {return x3;}
      if (c++ == li) throw new Error("biseccion no converge, x="+x3);
      if (y1*y3<0) x2=x3; else x1=x3;
      xp=x3;
    }
  }
```

Resolviendo la ecuación del ejemplo manual con este programa (empleando los valores por defecto), se obtiene:

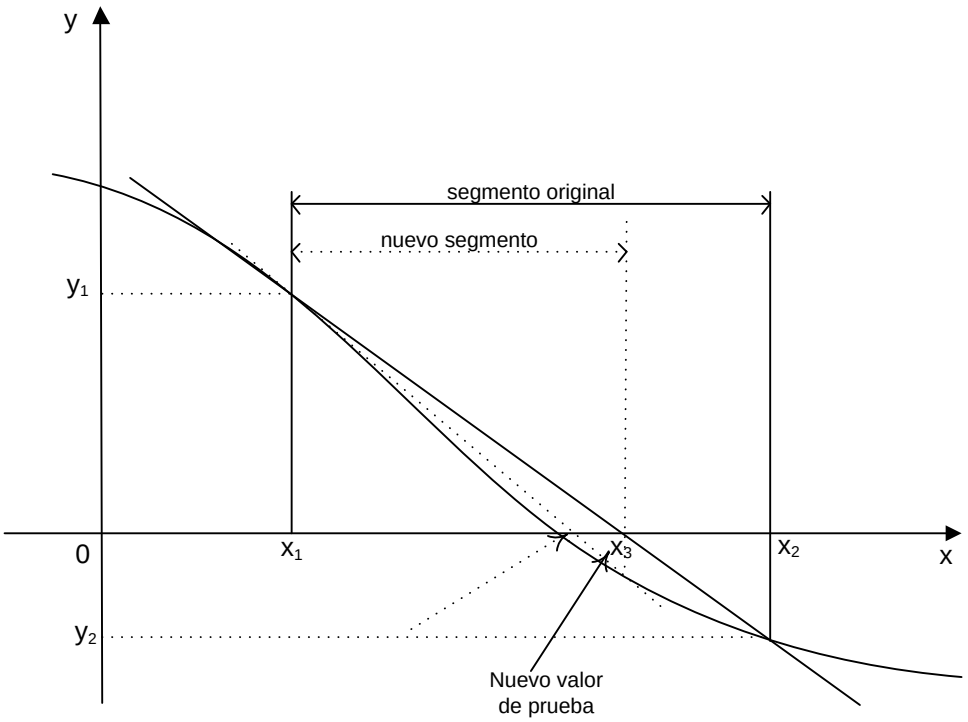
```
>>biseccion(f,3.6,3.7)
3.609670886138338
```

Y con un error de 16 dígitos (máxima precisión), se obtiene el mismo resultado que en el ejemplo manual:

```
>>biseccion(f,3.6,3.7,1e-16)
3.6096708861400497
```

4.4. MÉTODO DE REGULA FALSI (INTERPOLACIÓN LINEAL)

Al igual que el método de la Bisección, este método comienza con un segmento de solución conocido, sin embargo, en este caso el nuevo valor de prueba (x_3) se calcula por interpolación lineal, tal como se muestra en la figura:



En esencia difiere del método de la Bisección por la forma en que se encuentra el nuevo valor de prueba. La ecuación para calcular dicho valor

se deduce de la intersección entre la línea que pasa por los puntos (x_1,y_1), (x_2,y_2) y la recta y=0.

Reemplazando dichos puntos en la ecuación de la línea recta, de obtiene:

$$y_1 = a + bx_1$$

$$y_2 = a + bx_2$$

De donde resulta:

$$y_1 - y_2 = b(x_1 - x_2)$$

$$b = \frac{y_1 - y_2}{x_1 - x_2}$$

Y con "b" se puede calcular el valor de "a":

$$y_1 = a + \frac{y_1 - y_2}{x_1 - x_2} x_1$$

$$a = \frac{y_1 x_1 - y_1 x_2 - y_1 x_1 + y_2 x_1}{x_1 - x_2}$$

$$a = \frac{y_2 x_1 - y_1 x_2}{x_1 - x_2}$$

Reemplazando "a" y "b" en la ecuación general de la línea recta (y=a+bx), se obtiene:

$$y = \frac{y_2 x_1 - y_1 x_2}{x_1 - x_2} + \frac{y_1 - y_2}{x_1 - x_2} x$$

Pero en la intersección "y" es cero y "x" es "x_3", por lo tanto:

$$x_3 = \frac{y_2 x_1 - y_1 x_2}{y_2 - y_1}$$

Que es la ecuación del método de Regula Falsi.

Para comprender mejor el proceso se resolverá (una vez más) la ecuación, con 12 dígitos de precisión:

$$f(x) = \left(52 + 3\sqrt{x} - 8x^{0.8}\right)^{0.36} - x = 0$$

Como de costumbre, se comienza programando la función:

```
f=function(x){
  return pow(52+3*sqrt(x)-8*pow(x,0.8),0.36)-x;
}
```

Al igual que en el método de bisección, se encuentra el segmento de solución con "incr1", pero esta vez empleando un incremento igual a 1:

```
>>x1=incr1(f,1.1,1); x2=x1+1; write(x1,x2)
3.0999999999999996  4.1
```

En consecuencia, la solución se encuentra entre 3.1 y 4.1. Los valores de la función para estos puntos son (recuerde que en las consolas los valores se imprimen con "console.log"):

```
>>y1=f(x1); y2=f(x2); write(y1,y2)
0.586962288954437  -0.5660201764378305
```

Entonces, el nuevo valor de prueba ("x_3") y el respectivo valor de la función, son:

```
>>x3=(y2*x1-y1*x2)/(y2-y1); y3=f(x3);
write(precision(x3,12),precision(y3,12))
3.60908171336  0.000679273504388
```

Como se puede ver, con este método la función se aproxima a cero (en 3 dígitos) aún en una iteración.

Al igual que en el método de la bisección se repite el procedimiento empleando la estructura "if-else" para averiguar si la solución se encuentra a la izquierda o a la derecha. El proceso se repite hasta que el valor de la función tiene 12 ceros después del punto (es menor a 1e-12) y/o hasta que los dos últimos valores de "x" ("x_3") son iguales en 12 dígitos (que es la precisión con la que se está trabajando en el ejemplo):

```
>>if(y1*y3<0){x2=x3;y2=y3;} else {x1=x3;y1=y3;} x3=(y2*x1-y1*x2)/(y2-y1);
y3=f(x3); write(precision(x3,12),precision(y3,12))
3.60967015188  8.46552651979e-7
>>if(y1*y3<0){x2=x3;y2=y3;} else {x1=x3;y1=y3;} x3=(y2*x1-y1*x2)/(y2-y1);
y3=f(x3); write(precision(x3,12),precision(y3,12))
3.60967088522  1.05511244186e-9
>>if(y1*y3<0){x2=x3;y2=y3;} else {x1=x3;y1=y3;} x3=(y2*x1-y1*x2)/(y2-y1);
y3=f(x3); write(precision(x3,12),precision(y3,12))
3.60967088614  1.31583632879e-12
>>if(y1*y3<0){x2=x3;y2=y3;} else {x1=x3;y1=y3;} x3=(y2*x1-y1*x2)/(y2-y1);
y3=f(x3); write(precision(x3,12),precision(y3,12))
3.60967088614  1.33226762955e-15
```

Entonces, la solución, con 12 dígitos de precisión, es: 3.60967088614. Solución a la que se llega en un total de 5 iteraciones. Como ocurre en este ejemplo, el método de Regula Falsi requiere, casi siempre, menos iteraciones que el método de la Bisección.

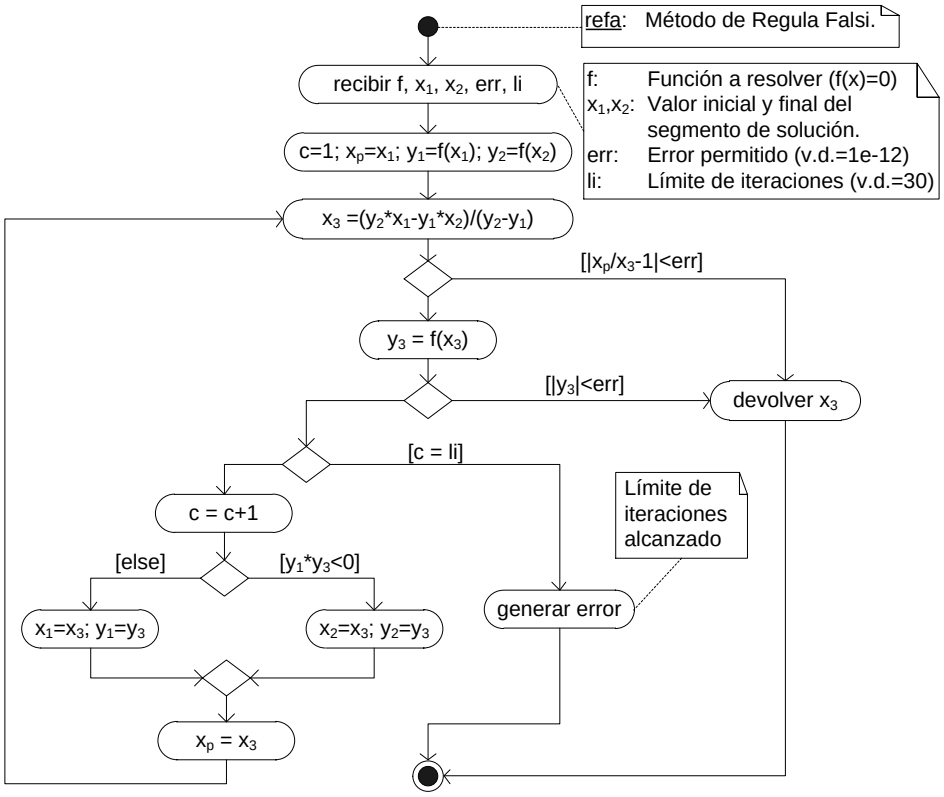
Por otra parte el método de Regula Falsi suele acercarse a la solución por un sólo lado (en el ejemplo se acerca sólo por el segmento derecho pues los valores de "y_3" son siempre positivos), lo que puede llevar a un número excesivo de iteraciones cuando la curva tiene pendientes muy pronunciadas cerca de la solución.

4.4.1. Algoritmo y código

El algoritmo del método de Regula-Falsi es esencialmente el mismo que el de la Bisección, excepto que ahora se emplea otra ecuación para el cálculo del nuevo valor de prueba ("x_3") y que el intercambio de variables implica tanto a "x" como a "y", tal como se puede ver en el diagrama de la siguiente página.

El código respectivo, añadido a la librería "mat205.js", es:

```
function refa(f,x1,x2,err,li) {
  if (err==null) {err=1e-12;}
  if (li==null) {li=30;}
  var c=1,xp=x1,y1=f(x1),y2=f(x2),x3,y3;
  while (true) {
```




```
        x3=(y2*x1-y1*x2)/(y2-y1);
        if (abs(xp/x3-1)<err) {return x3;}
        y3=f(x3);
        if (abs(y3)<err) {return x3;}
        if (c++ == li) throw new Error("biseccion no converge, x="+x3);
        if (y1*y3<0) {x2=x3;y2=y3;} else {x1=x3;y1=y3;}
        xp=x3;
    }
}
```

Encontrando la solución de la ecuación del ejemplo manual, con este programa (empleando "incr1" para determinar el segmento de solución), se obtiene:

```
>>x1=incr1(f,1.1,1); x2=x1+1; precision(refa(f,x1,x2),12)
3.60967088614
```

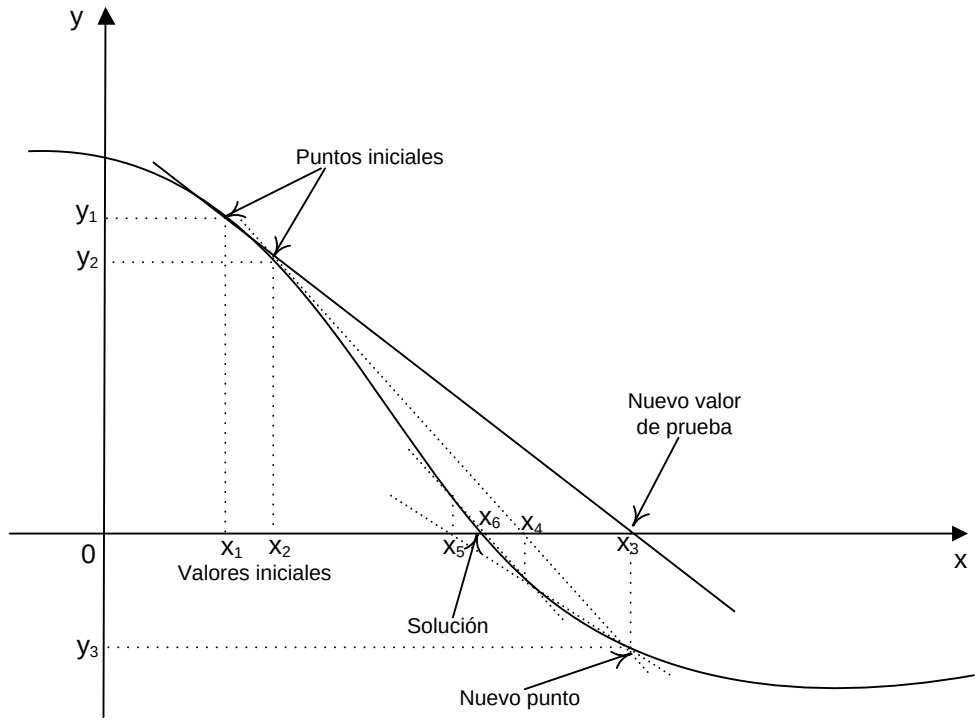
Que es el mismo resultado encontrado en el ejemplo manual (recuerde que el error por defecto del método es 1e-12).

4.5. MÉTODO DE LA SECANTE

El método de la Secante requiere dos valores iniciales (x_1 y x_2), sin embargo, a diferencia de los métodos de la Bisección y de Regula Falsi, dichos valores no necesitan estar a ambos lados de la solución, es decir que en el método de la secante no se requiere conocer el segmento de solución, sin embargo, si los valores iniciales constituyen el segmento de solución, la convergencia hacia la solución es más rápida y segura.

Como se puede ver en la figura de la siguiente página, con los valores iniciales se calcula un nuevo valor de prueba x_3, interpolando o extrapo-

lando a través de los puntos asumidos hasta la recta "y=0":



Luego se repite el proceso, empleando los dos últimos puntos, hasta que el valor de la función ($f(x_3)$) es cercano a cero o hasta que los valores de prueba de dos iteraciones consecutivas son aproximadamente iguales.

La ecuación para el cálculo del valor de prueba "x_3" es la misma que para el método de Regula Falsi:

$$x_3 = \frac{y_2 x_1 - y_1 x_2}{y_2 - y_1}$$

Para comprender mejor el proceso se encontrará (una vez más) la solución de la siguiente ecuación, con 9 dígitos de precisión:

$$f(x) = \left(52 + 3\sqrt{x} - 8x^{0.8}\right)^{0.36} - x = 0$$

Primero, como de costumbre, se programa la función:

```
>>f=function(x){ return pow(52+3*sqrt(x)-8*pow(x,0.8),0.36)-x; }
```

Luego se asumen valores iniciales, en este caso dos valores consecutivos: x1=1.1, x2=1.2, y con los mismos se calculan los valores respectivos de la función y el nuevo valor de prueba:

```
>>x1=1.1; x2=1.2; y1=f(x1); y2=f(x2); x3=(y2*x1-y1*x2)/(y2-y1);
precision(x3,9)
3.60944735
```

Siendo el valor de la función para el nuevo valor de prueba:

```
>>y3=f(x3); precision(y3,9)
0.000257726112
```

Como se puede ver, aún con el primer valor de prueba la función se acerca a cero en 3 dígitos.

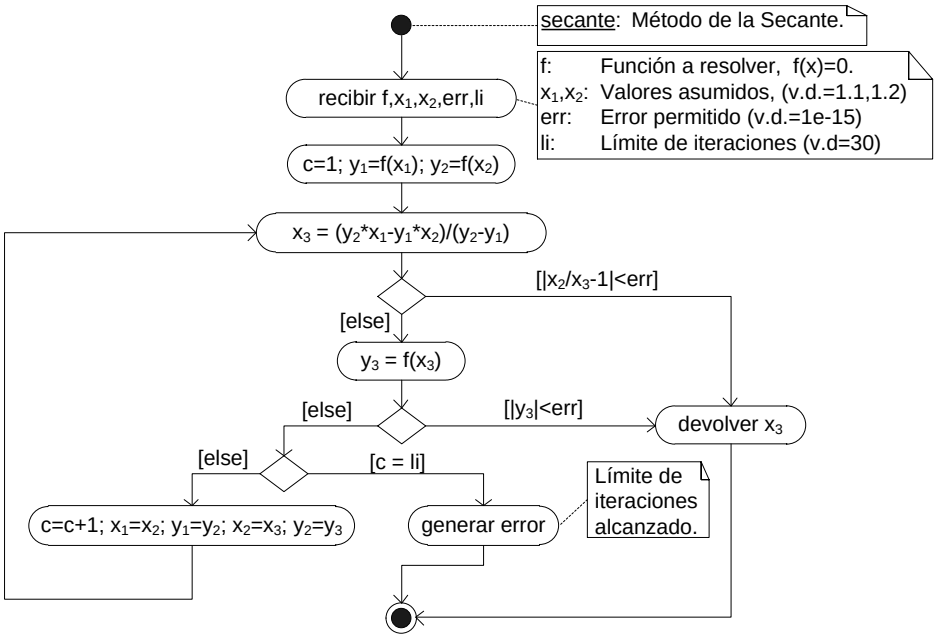
Ahora se repite el proceso, realizando el cambio de variables para emplear siempre los dos últimos puntos, hasta que los valores de prueba (x_3) de dos iteraciones sucesivas son iguales (en los 9 dígitos) o hasta que el valor de la función es cero (o menor a 10^{-9}):

```
>>x1=x2;x2=x3;y1=y2;y2=y3;x3=(y2*x1-y1*x2)/(y2-y1);y3=f(x3);
write(precision(x3,9),precision(y3,9))
3.60967162  -8.42444867e-7
>>x1=x2;x2=x3;y1=y2;y2=y3;x3=(y2*x1-y1*x2)/(y2-y1);y3=f(x3);
write(precision(x3,9),precision(y3,9))
3.60967089  4.45421477e-13
>>x1=x2;x2=x3;y1=y2;y2=y3;x3=(y2*x1-y1*x2)/(y2-y1);y3=f(x3);
write(precision(x3,9),precision(y3,9))
3.60967089  -4.4408921e-16
```

Como se puede ver en este caso se requieren apenas 3 iteraciones para llegar a la solución y en realidad sólo 2, porque el valor de la función es menor a 1e-9 en la penúltima iteración (sólo se ha repetido una vez más para garantizar que la precisión también se cumpla).

4.5.1. Algoritmo y código

El algoritmo que automatiza el proceso es:



Siendo el código respectivo, añadido a la librería "mat205.js":

```
function secante(f,x1,x2,err,li){
  if (x1==null && x2==null) {x1=1.1; x2=1.2;}
  if (x1!=null && x2==null) x2=x1*1.1;
  if (err==null) err=1e-15;
  if (li==null) li=30;
  var c=1,y1=f(x1),y2=f(x2),x3,y3;
  while (true) {
    x3=(y2*x1-y1*x2)/(y2-y1);
```

```
    if (abs(x2/x3-1)<err) return x3;
    y3=f(x3);
    if (abs(y3)<err) return x3;
    if (c++ == li) throw new Error("secante no converge, x="+x3);
    x1=x2;y1=y2;x2=x3;y2=y3;
  }
}
```

Volviendo a resolver la ecuación del ejemplo manual, con este programa (con la precisión de 9 dígitos), se obtiene:

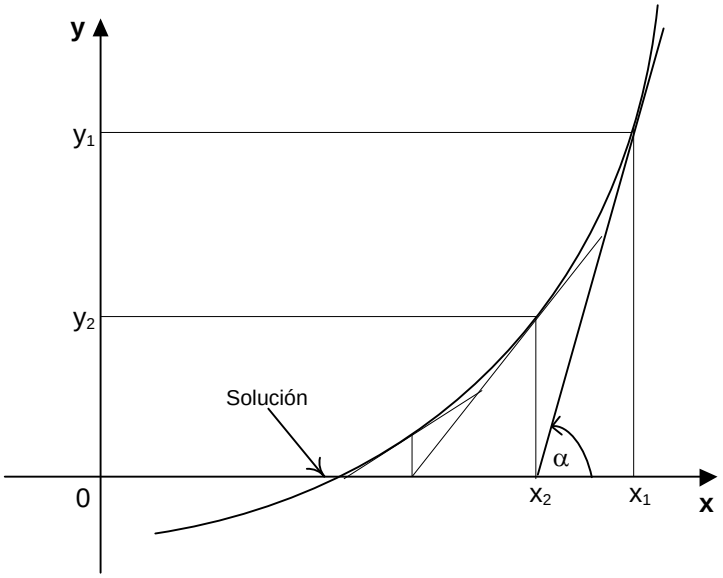
```
>>precision(secante(f,1.1,1.2,1e-9),9)
3.60967089
```

Que es el mismo resultado calculado en el ejemplo manual. Con la precisión por defecto (de 15 dígitos), se obtiene:

```
>>precision(secante(f),15)
3.60967088614005
```

4.6. MÉTODO DE NEWTON-RAPHSON

Este método sigue el proceso que se muestra en la figura:



Es decir se asume un valor inicial *(x_1)* y con el mismo se calcula la derivada de la función (la tangente a la curva en x_1). Con la tangente, en la intersección con la recta *y=0*, se encuentra el nuevo valor de prueba *(x_2)*. Este proceso se repite (calculando la tangente en el nuevo punto) hasta que el valor de la función es cercano a cero o hasta que los valores de "x" de dos iteraciones sucesivas son aproximadamente iguales.

La ecuación para el cálculo del nuevo valor de prueba *(x_2)* puede ser deducida del triángulo que forman la tangente con los segmentos (como se puede ver en la figura) donde se cumple que:

$$\tan(\alpha) = f'(x_1) = \frac{f(x_1) - 0}{x_1 - x_2}$$

De donde resulta:

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

$$x_2 = x_1 - \frac{y_1}{f'(x_1)}$$

Que es la ecuación del método de Newton-Raphson. Como se puede ver, esta ecuación es muy sencilla, sin embargo, implica el cálculo de la derivada primera de la función y es ahí donde radica la mayor dificultad del método.

Si bien en ocasiones el cálculo de la derivada analítica puede ser simple, en otros es complejo e incluso imposible (en funciones compuestas por ejemplo). Afortunadamente, es posible obtener un resultado numérico aproximado, aplicando las fórmulas de diferenciación (que se deducen a partir del concepto de la derivada).

En este método se empleará la fórmula de diferencia central de segundo orden (cuya deducción se estudiará en temas posteriores):

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

Donde "h" es el incremento de "x" (Δx). En teoría, puesto que la derivada se encuentra en el límite cuando Δx tiende a cero, mientras más pequeño sea el incremento más exacto será el resultado obtenido. En la práctica, no obstante, no es así debido a los errores de redondeo. Los errores de redondeo se hacen más grandes mientras más pequeño sea "h". Por eso se debe emplear un valor intermedio (ni muy grande, ni muy pequeño), siendo un valor adecuado para la mayoría de los casos $x_1 * 10^{-6}$.

Con este incremento se puede crear una función para el cálculo de la derivada primera (misma que ha sido añadida a la librería "mat205.js"):

```
function df1(f,x){
  var h=x*1e-6;
  return (f(x+h)-f(x-h))/(2*h);
}
```

Por ejemplo, para calcular las derivadas de la función:

$$f(x) = \left(52 + 3\sqrt{x} - 8x^{0.8}\right)^{0.36} - x = 0$$

En x=2.1, 3.2 y 6.5, se escribe:

```
>>f=function(x){ return pow(52+3*sqrt(x)-8*pow(x,0.8),0.36)-x; }
>>df1(f,2.1)
-1.147851702047797
>>df1(f,3.2)
-1.150909119257415
>>df1(f,6.5)
-1.1801206938162505
```

Y para encontrar la solución con el método de Newton-Raphson, se asume un valor inicial (por ejemplo 1.1) y con el mismo se calcula el valor de la función y el nuevo valor de prueba:

```
>>x1=1.1;y1=f(x1);x2=x1-y1/df1(f,x1);write(y1,x2)
2.8840553877884245  3.608916103269472
```

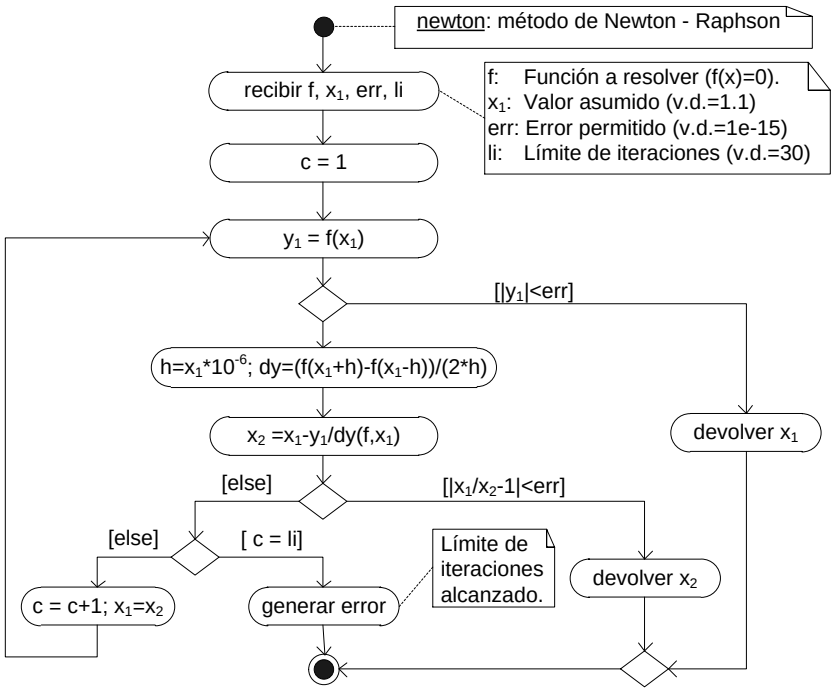
Ahora simplemente se repite el proceso, haciendo que "x1" tome el valor de "x2", hasta que la función es casi cero y/o los valores de prueba de dos iteraciones sucesivas se repiten.

```
>>x1=x2;y1=f(x1);x2=x1-y1/df1(f,x1);write(y1,x2)
0.0008702095902952678  3.6096708874878503
>>x1=x2;y1=f(x1);x2=x1-y1/df1(f,x1);write(y1,x2)
-1.5539178832568723e-9  3.60967088614005
>>x1=x2;y1=f(x1);x2=x1-y1/df1(f,x1);write(y1,x2)
-4.440892098500626e-16  3.6096708861400497
>>x1=x2;y1=f(x1);x2=x1-y1/df1(f,x1);write(y1,x2)
0  3.6096708861400497
```

Como se puede ver, en la última iteración (en la cuarta) se cumplen las dos condiciones: el valor de la función es cero (solución exacta) y se repiten los dos últimos valores de "x", por lo que se puede concluir que el resultado calculado es exacto.

4.6.1. Algoritmo y código

El algoritmo que automatiza el proceso es:



Siendo el código respectivo, añadido a la librería "mat205.js":

```
function newton(f,x1,err,li){
  if (x1==null) x1=1.1;
  if (err==null) err=1e-15;
  if (li==null) li=30;
  var c=1,y1,h,dy,x2;
  while (true) {
    y1=f(x1);
    if (abs(y1)<err) return x1;
    h=x1*1e-6; dy=(f(x1+h)-f(x1-h))/(2*h);
```

```
        x2=x1-y1/dy;
        if (abs(x1/x2-1)<err) return x2;
        if (c++ == li) throw new Error("newton no converge, x="+x2);
        x1=x2;
    }
}
```

Observe que en este y en el anterior método el error por defecto es de 15 dígitos (1e-15), esto porque ambos métodos son menos propensos a los errores de redondeo y porque además requieren menos iteraciones que los métodos estudiados previamente.

Resolviendo la ecuación del ejemplo manual, con una precisión de 16 dígitos, se obtiene:

```
>>newton(f,1.1,1e-16)
3.6096708861400497
```

Que es el mismo resultado obtenido en el proceso manual.

4.7. EJEMPLOS

1. **Encuentre las soluciones aproximadas de la siguiente función graficándola entre x=0 y x=10. Luego empleando estas soluciones como valores asumidos, encuentre soluciones más precisas, con 9 dígitos de precisión, empleando el método de Wegstein.**

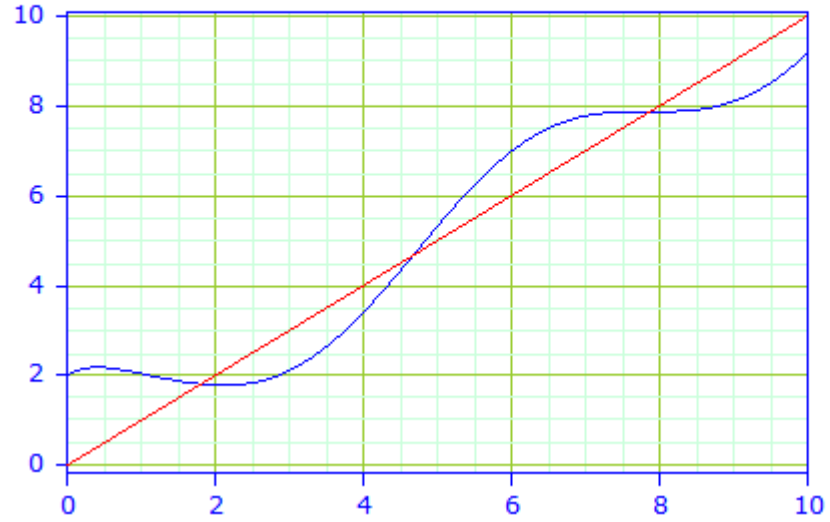
$$f(x) = \cos(x) + (1 + x^2)^{-1} = 0$$

Primero ser lleva la ecuación a la forma "x=g(x)" y la manera más sencilla de hacerlo es sumar "x" a ambos lados de la ecuación:

$$0 + x = x = g(x) = \cos(x) + (1 + x^2)^{-1} + x$$

Entonces se programa y grafica esta función (conjuntamente la recta y=x), para obtener los valores iniciales:

```
>>g1=function(x){return cos(x)+1/(1+x*x)+x;}; y=function(x){return x;}
function()
>>plot([g1,y],0,10)
```



Como se puede ver, existen tres soluciones aproximadamente en 1.7, 4.6

y 7.9. Con estos valores (como valores iniciales asumidos) se encuentran las soluciones con el método de Wegstein (con 9 dígitos de precisión):

```
>>precision(wegstein(g1,1.7,1e-9),9)
1.80737538
>>precision(wegstein(g1,4.6,1e-9),9)
4.66850552
>>precision(wegstein(g1,7.9,1e-9),9)
7.86987174
```

Y como se puede ver, a diferencia del método de sustitución directa, el método de Wegstein logra convergencia (encuentra las soluciones), en los tres casos.

2. Encuentre las soluciones aproximadas de la siguiente función, grafi-cándola entre x=1.5 y x=2, luego empleando esas soluciones como valo-res iniciales, encuentre soluciones más precisas, con 9 dígitos de precisión después del punto, empleando el método incremental ("incr2").

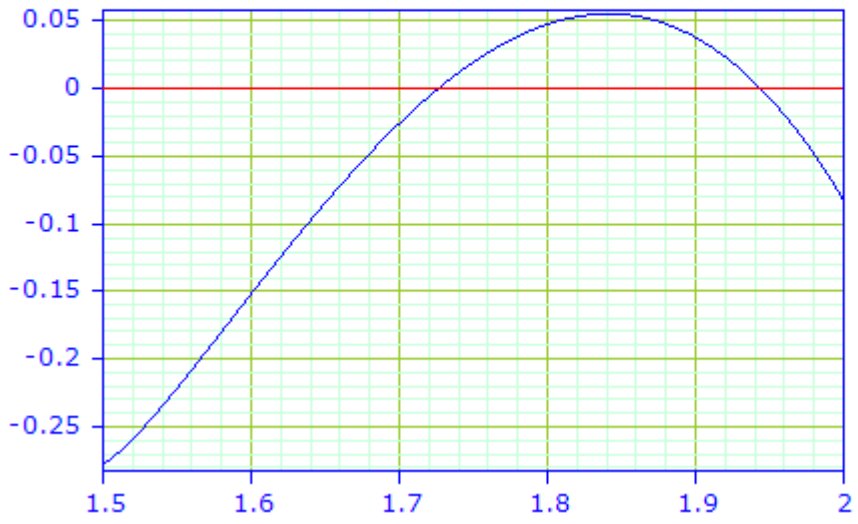
$$3x^{2.1} - 5y = 7.0$$
$$y^{1.2} + 4z = 14.3$$
$$x + y^2 + z^2 = 14.0$$

Primero se escribe la ecuación como una función de una variable igualada a cero:

$$f(x) = 14.0 - y^2 - z^2 - x = 0$$
$$y = \frac{3x^{2.1} - 7.0}{5}$$
$$z = \frac{14.3 - y^{1.2}}{4}$$

Entonces se programa y grafica la función entre los límites dados:

```
>>f2=function(x){var y=(3*pow(x,2.1)-7)/5, z=(14.3-pow(y,1.2))/4; return
14-y*y-z*z-x;}
function (x){var y=(3*pow(x,2.1)-7)/5, z=(14.3-pow(y,1.2))/4; return 14-
y*y-z*z-x;}
>>y2=function(x){return 0;}; plot([f2,y2],1.5,2)
```



Los valores más cercanos a las soluciones (con un dígito después del punto) son: 1.7 y 1.9.

Empleando estos valores en el método incremental ("incr2"), con un incremento inicial igual a 0.01, se obtiene:

```
>>round(incr2(f2,1.7,0.01,1e-10),9)
1.726526148
>>round(incr2(f2,1.9,0.01,1e-10),9)
1.943797085
```

Que, son las soluciones pedidas.

3. Calcule la frecuencia natural de una viga "p" con 9 dígitos de precisión, resolviendo la siguiente ecuación con el método de la Bisección. Se sabe que la frecuencia natural de una viga es siempre un valor positivo. El segmento de solución debe ser encontrado con el método incremental.

$$\cos(k \cdot l) \cdot \cosh(k \cdot l) = -1$$

$$k^2 = \frac{p}{a}$$

$$a^2 = \frac{E \cdot I \cdot g}{A \cdot y}$$

$$l = 120 plg (longituddelaviga)$$

$$I = 170.6 plg^4 (momentodeinercia)$$

$$E = 3 \times 10^6 \, lb/plg^2 \, (móduloelástico)$$

$$y = 0.066 \, lb/plg^3 \, (densidad)$$

$$A = 32 plg^2 (secciónтransversal)$$

$$g = 386 \, plg/s^2 \, (aceleracióndelagravedad)$$

Esta ecuación depende implícitamente de la variable "p" (la frecuencia), siendo la función a resolver (igualada a cero):

$$f(p) = \cos(k \cdot l) \cdot \cosh(k \cdot l) + 1 = 0$$

Como de costumbre primero se programa la función:

```
f=function(p){
  var g=386,A=32,y=0.066,E=3e6,I=170.6,l=120,a,k;
  a=sqrt(E*I*g/(A*y)); k=sqrt(p/a);
  return cos(k*l)*cosh(k*l)+1;
}
```

Ahora, partiendo de un valor positivo (0.1) y un incremento igual a 1, se encuentra el segmento de solución con "incr1" y la solución con "refa":

```
>>x1=incr1(f,0.1,1); x2=x1+1; precision(biseccion(f,x1,x2,1e-9),9)
74.6766963
```

Por lo tanto la frecuencia natural de la viga, para los parámetros dados es de 74.676693 Hz. En la práctica, sin embargo, no se requiere un resultado con tanta precisión, en este caso, por ejemplo es suficiente conocer la frecuencia de la viga con un dígito después del punto, es decir que un resultado de utilidad en ingeniería es 74.7 Hz.

4. Calcule la fracción final de soluto en la fase líquida (x_1), de una columna de absorción equivalente a una etapa de equilibrio, resolvien-

do, con 10 dígitos de precisión, la siguiente ecuación por el método de Regula-Falsi (se sabe que la solución es positiva y menor a 0.4).

$$f(x_1) = L_1 \cdot x_1 + V_1 \cdot y_1 - C_0 = 0$$
$$L_1 = \frac{L_c}{1 - x_1}$$
$$V_1 = M - L_1$$
$$y_1 = 1.42 \cdot x_1$$
$$L_c = L_0 \cdot (1 - x_0)$$
$$M = L_0 + V_0$$
$$C_0 = L_0 \cdot x_0 + V_0 \cdot y_0$$
$$L_0 = 300\, kgmol/h$$
$$x_0 = 0$$
$$V_0 = 100$$
$$y_0 = 0.2)$$

Como de costumbre, para resolver el problema, primero se programa la función:

```
function (x1){
  var L0=300,x0=0,V0=100,y0=0.2,C0,M,Lc,y1;
  C0=L0*x0+V0*y0; M=L0+V0; Lc=L0*(1-x0); y1=1.42*x1;
  L1=Lc/(1-x1); V1=M-L1;
  return L1*x1+V1*y1-C0;
}
```

Entonces se encuentra el segmento de solución con "incr1" comenzando la búsqueda en 0 y con un incremento igual a 0.01 (porque el segmento de búsqueda es pequeño). Con el segmento de solución se encuentra luego la solución llamando a la función "biseccion":

```
>>dx=0.01; x1=incr1(f4,0,dx); x2=x1+dx;
precision(refa(f4,x1,x2,1e-10),10)
0.04587771997
```

Que es la solución buscada. Una vez más, en la práctica no se requiere tanta precisión, siendo suficiente dos dígitos, es decir: 0.046.

5. **Un dato que se requiere en el diseño de redes de distribución de agua es el factor de fricción "f", el cual se obtiene de tablas, pero que también puede ser calculado con la ecuación:**

$$\frac{1}{\sqrt{f}} = \left(\frac{1}{k}\right) \ln\left(R_e \sqrt{(f)}\right) + \left(14 - \frac{5.6}{k}\right)$$
$$k = 0.28$$

Donde el valor de "k" puede variar en función a las características particulares del agua que se está transportando y "Re" es el número de Reynolds que depende de la velocidad del agua y diámetro de la tubería.

Elabore una función que reciba el número de Reynolds y devuelva el coeficiente de fricción respectivo, resolviendo la ecuación con el método de la Secante (valores iniciales 0.1 y 0.2, precisión de 9 dígitos). Luego emplee la función creada para calcular el factor de fricción para números de Reynolds iguales a 1.5, 12.2, 21.3, 27.6 y 33.7.

Esta es la clase de problemas que se resuelve en la práctica, es decir, el cálculo de uno o más resultados de utilidad a partir de uno o más valores conocidos, en este caso el cálculo del factor de fricción a partir del número de Reynolds.

El problema debe ser resuelto en una función que reciba el número de Reynolds y devuelva el coeficiente de fricción. Pero como para calcular el coeficiente de fricción se debe resolver la ecuación, es necesario programarla dentro de la función, algo que es posible en lenguajes como Javascript.

Como de costumbre la función a resolver debe estar igualada a cero:

$$f(f) = \frac{1}{\sqrt{f}} - \left(\frac{1}{k}\right)\ln\left(R_e\sqrt{(f)}\right) - \left(14 - \frac{5.6}{k}\right) = 0$$

La función principal recibe el número de Reynolds "Re" y "k" es una constante (0.28), por lo tanto la única variable (incógnita) es "f". Con estas consideraciones la función que resuelve el problema es:

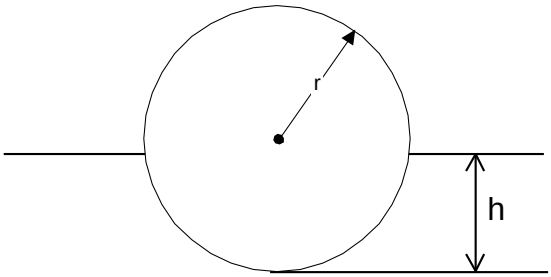
```
>>fric=function(Re){ var k=0.28; var ff=function(f){return
1/sqrt(f)-1/k*log(Re*sqrt(f))-(14-5.6/k);} return
precision(secante(ff,0.1,0.2,1e-9),9); }
function (Re){
  var k=0.28;
  var ff=function(f){return 1/sqrt(f)-1/k*log(Re*sqrt(f))-(14-
5.6/k);}
  return precision(secante(ff,0.1,0.2,1e-9),9);
}
```

Como se puede ver, a la función interna se le ha dado el nombre "ff", la misma que es resuelta con el método de la secante y el resultado es devuelto con 9 dígitos de precisión.

Entonces, los resultados pedidos (los coeficientes de fricción) se calculan con la función "frac" (empleando "map"):

```
>>map(ejem2,[1.5,12.2,21.3,27.6,33.7])
[14.8001058, 0.446974842, 0.213322816, 0.156009388, 0.1241916]
```

6. Una esfera de densidad "d" se encuentra parcialmente sumergida en agua, como se muestra en la figura:



El peso de la esfera de radio r es $(4/3)(\pi r^3 d)$ y el volumen del segmento sumergido es $(\pi/3)(3rh^2-h^3)$.

Una industria que diseña reguladores de flujo de agua, empleando flotadores, necesita conocer la altura "h" a la que se sumerge la esfera. Elabore una función que reciba como datos la densidad (d) y radio (r) de la esfera y devuelva la altura (h) a la que se sumerge.

La ecuación que rige el sistema es la igualdad entre el peso de la esfera y el volumen de líquido desplazado (principio del empuje) y debe ser resuelta con el método de Newton-Raphson con 8 dígitos de precisión (valor inicial 0.5*r).

Una vez elaborada la función, debe ser empleada para calcular la altura sumergida para los siguientes casos: d=0.1, r=3.4; d=0.4, r=1.3; d=0.6, r=3.1; d=0.8, r=2.5; d=0.95, r=3.6; d=1.2, r=1.9; d=-1.2, r=2.2.

En este problema, la ecuación que rige el sistema (tal como se indica en el enunciado) es:

$$\frac{4}{3}\pi r^3 d = \frac{\pi}{3}(3rh^2 - h^3)$$

Que colocada en la forma f(x)=0 es:

$$f(h) = \frac{4}{3}\pi r^3 d - \frac{\pi}{3}(3rh^2 - h^3) = 0$$

Esta es la ecuación que debe ser resuelta con el método de Newton-Raphson (observe que no es una ecuación lineal). Al igual que el problema anterior, esta función debe ser programada dentro otra función (la función principal) que es la que recibe como datos los valores de "d" (la densidad) y "r" (el radio).

Además es necesario verificar (antes de calcular la altura) que la densidad sea menor a 1 (porque si es mayor a uno la esfera se hundiría completamente) y que no sea menor a 0 (porque es una imposibilidad física), generando un error (con "throw") en caso de no cumplirse estas condiciones.

La función que resuelve el problema, tomando en cuenta las anteriores consideraciones, es:

```
>>altura=function(d,r){ if (d>=1) throw new Error("La densidad
debe ser menor a 1"); if (d<=0) throw new Error("La densidad
debe ser mayor a 0"); var fh=function(h){ return 4/3*PI*r*r*r*d-
PI/3*(3*r*h*h-h*h*h);} return precision(newton(fh,0.5*r,1e-
7),7); }
function (d,r){
  if (d>=1) throw new Error("La densidad debe ser menor a 1");
  if (d<=0) throw new Error("La densidad debe ser mayor a 0");
  var fh=function(h){
          return 4/3*PI*r*r*r*d-PI/3*(3*r*h*h-h*h*h);}
  return precision(newton(fh,0.5*r,1e-7),7);
}
```

Y con esta función se calculan los resultados pedidos:

```
>>map(altura,[0.1,0.4,0.5,0.8,0.95,1.2,-1.2],
[3.4,1.3,3.1,2.5,3.6,1.9,2.2])
err: La densidad debe ser menor a 1
```

Y como se puede ver se genera un error (porque dos de los datos están fuera de los límites permitidos). Suprimiendo esos datos se obtiene:

```
>>map(altura,[0.1,0.4,0.5,0.8,0.95],[3.4,1.3,3.1,2.5,3.6])
[1.331441, 1.125621, 3.1, 3.564296, 6.225477]
```

Que son los resultados buscados.

4.8. EJERCICIOS

Como en el anterior tema para presentar los ejercicios en la calculadora, copie las instrucciones en un editor de texto y luego, al momento de presentar el trabajo, vuelva a copiarlos (y ejecutarlos) en la calculadora, alternativamente, los problemas pueden ser resueltos también en páginas HTML y presentados en esa forma (lo que evita el trabajo de copiar y volver a ejecutar las instrucciones).

1. Encuentre la solución aproximada de la siguiente función, graficándola entre x=-5 y x=1. Luego, empleando la solución aproximadas como valor iniciales, encuentre una solución más precisa, con 9 dígitos de precisión, empleando el método de wegstein.

$$f(x) = x^3 + 2x^2 + 3x + 4 = 0$$

2. Encuentre las soluciones aproximadas de la siguiente función, graficándola entre x=-1 y x=2. Luego, empleando las soluciones aproximadas como valores iniciales, encuentre soluciones más precisas, con 10 dígitos de precisión, aplicando el método incremental.

$$f(x) = 2 * x^2 + 1 - \exp(x) = 0$$

3. Encuentre los segmentos de solución de la siguiente función graficándola entre z=-10 y z=10. Luego, empleando los segmentos de solución encontrados en la gráfica, calcule soluciones más precisas, con 8 dígitos de precisión, con el método de la bisección.

$$f(z) = e^{z/2} + z^2 - 60 = 0$$

4. Encuentre las soluciones aproximadas del siguiente sistema de ecuaciones (como una función de "x"), graficándola entre x=1 y x=10. Luego, empleando valores previos a las soluciones aproximadas, el método incremental y el método de Regula Falsi, encuentre soluciones más precisas, con 9 dígitos de precisión.

$$2x^2 + 5xy - 4x = 115$$
$$e^{\frac{x+y}{5}} + x^2y^2 - 70y = 15$$

5. Encuentre el volumen "V" que ocupa un gmol de metano, resolviendo la siguiente ecuación con el método de la secante, con 9 dígitos de precisión. Encuentre el segmento de solución con "incr1" (se sabe que la solución es mayor a 0.5 y menor a 10).

$$f(V) = \frac{P \cdot V}{R \cdot T} - \frac{V}{V - b} + \frac{\Omega_a}{\Omega_b}\frac{b}{V + b} \cdot F$$

$$b = \frac{\Omega_a \cdot R \cdot T_c}{P_c}$$

$$F = \frac{1}{T_r}\left[1 + (0.480 + 1.574\omega - 0.176\omega^2)(1 - \sqrt{T_r})\right]^2$$

$$T_r = \frac{T}{T_c}$$

$$\Omega_a = \left[9\left(2^{1/3}-1\right)\right]^{-1}$$
$$\Omega_b = \frac{2^{1/3}-1}{3}$$
$$T = 400K(temperatura)$$
$$P = 20atm(presión)$$
$$T_c = 190.6\text{K}(temperaturacrítica)$$
$$P_c = 45.4atm(presióncrítica)$$
$$w = 0.008(factoracéntrico)$$
$$R = 0.08206(constanteuniversaldelosgases)$$

6. En el diseño de un aislante, para una tubería que transporta un fluido caliente, se necesita conocer la temperatura a diferentes espesores del aislante. Se ha encontrado que la diferencia de temperatura (t) está relacionada con el espesor del aislante (L_C), mediante expresión:

$$e^{-t/2}\cosh^{-1}\left(e^{t/2}\right) = \sqrt{\frac{L_c}{2}}$$

Elabore una función que reciba el espesor (L_C) de aislante (en m) y devuelva la diferencia de temperatura (t en K) para ese espesor. La ecuación debe ser resuelta con el método de Newton-Raphson (valores iniciales 5.1, 5.2, precisión 9 dígitos). Con la función elaborada encuentre la temperatura para espesores iguales a: 0.01, 0.02, 0.05, 0.07, 0.08, 0.09 y 0.1 m.

7. Una fábrica de trampas para ratas, fabrica una trampa con una mandíbula de 9.5 cm. Se ha encontrado que la trampa puede ser armada fácilmente si para cerrarla se requiere un torque (T_C) de 0.625 lb.pie y que la trampa es efectiva si su momento de inercia (I_0) es de 0.0006 lb.pie.s^2, se ha determinado también que el desplazamiento angular de la mandíbula (θ_k, la distancia angular que recorre hasta quedar completamente cerrada) es de 3.27 radianes, y la que debe recorrer hasta tocar a la rata (θ) es de 2.97 radianes. Para que la trampa se cierre, sin que la rata tenga oportunidad de escapar, el tiempo debe ser inferior a 0.025 segundos, por supuesto, mientras menor sea ese tiempo menos probable es que la rata escape. La función que relaciona el tiempo con la constante del resorte es:

$$\theta = \frac{T_0}{k}\left(1-\cos\left(\sqrt{\frac{k}{I_0}}\cdot t\right)\right)$$
$$T_0 = T_c + \theta_k \cdot k$$

Elabore una función que reciba como dato el tiempo "t" (en segundos) en que debe cerrarse la trampa y devuelva la constante del resorte "k" (en lb.pie/radianes) con la que se logra dicho tiempo. La función debe ser resuelta con el método de la Newton-Raphson (valores iniciales por defecto, precisión de 7 dígitos). Con la función elaborada, encuentre

el valor de la constante del resorte para valores del tiempo iguales a: 0.025, 0.0225, 0.02, 0.0175, 0.015, 0.0125 y 0.01.

5. ECUACIONES POLINOMIALES

Si bien las ecuaciones polinomiales son también ecuaciones con una incógnita y por lo tanto pueden ser resueltas con los métodos estudiados en los temas anteriores, son una forma que se presenta con relativa frecuencia en la resolución de problemas en el campo de la ciencia y la ingeniería, por lo que existen métodos específicos para las mismas.

Dichos métodos sólo resuelven este tipo de ecuaciones por lo que están optimizados para las mismas, pero además, la mayoría de ellos, permiten encontrar no sólo las soluciones reales sino también las imaginarias.

Con los métodos estudiados en los anteriores temas es posible encontrar soluciones complejas, pero para ello se requiere trabajar con números complejos, mientras que con los métodos que se estudian en este tema sólo se trabaja con números reales.

La forma general que se tomará en cuenta para trabajar con estas ecuaciones es:

$$P_n(x)=a_0x^n+a_1x^{n-1}+a_2x^{n-2}+\cdots+a_{n-1}x+a_n=0 \qquad (5.1)$$

Donde "n" es el grado del polinomio y a_0, a_1, … a_n son los coeficientes del mismo (un vector).

5.1. VECTORES EN JAVASCRIPT

Dado que los coeficientes de un polinomio constituyen un vector, es necesario aprender a trabajar con vectores.

Un vector (o array) es un tipo de dato que permite guardar, en una variable, dos o más datos del mismo tipo. En Javascript, sin embargo, se pueden guardar dos o más datos de cualquier tipo. Esto es posible porque las matrices en Javascript no guardan los datos propiamente, sino sólo *punteros*.

Un puntero simplemente es una dirección de memoria y como tal, puede apuntar (tener la dirección de memoria) de cualquier dato (incluido otro puntero o una función), esa es la razón por la cual un vector en Javascript aparentemente guarda diferentes tipos de datos, pero en realidad sólo contiene punteros (los cuales pueden apuntar a diferentes tipos de datos).

Aun cuando el manejo y uso de punteros es transparente en la mayoría de los lenguajes modernos (es decir que se crean, actualizan y destruyen automáticamente), **es importante tener en mente** (para evitar errores inesperados) **que los punteros son sólo direcciones de memoria y no él o los datos en sí**.

Para comprender mejor el por qué es importante recordar que los punteros son sólo direcciones de memoria escriba las siguientes instrucciones en la calculadora Javascript:

```
>>a=[1,2,3]
[1, 2, 3]
>>b=a;
[1, 2, 3]
>>a.push(4)
4
>>a
[1, 2, 3, 4]
>>b
[1, 2, 3, 4]
```

Como se puede observar, se crea la variable "a" (un vector), luego se asigna la variable "a" a la variable "b", con lo que se esperaría se cree una copia, sin embargo, al añadir un elemento (el número 4) al final del vector "a" (con "a.push(4)") no solo cambia el vector "a" (como se esperaría) sino también el vector "b". Ocurre exactamente lo mismo cuando se modifica la variable "b", por ejemplo si se añade un elemento (el número 0) al principio del vector "b" (con "b.unshift(0)"), lo que cambia no sólo el vector "b", sino también el vector "a":

```
>>b.unshift(0)
5
>>a
[0, 1, 2, 3, 4]
```

Este comportamiento se comprende si se toma en cuenta que "a" y "b" son punteros, es decir que "a" no contiene los números "1,2,3", sino solo la dirección donde se encuentran dichos números, por lo tanto cuando "b" toma el valor de "a", lo que se guarda en "b" no son los números "1,2,3", sino sólo la dirección de memoria donde se encuentran dichos números. En consecuencia, cualquier modificación a los valores de "a" o "b" es una modificación en esa dirección de memoria y como ambas variables muestran (automáticamente) el contenido de dicha dirección, cualquier cambio hecho en "a" es también un cambio en "b" y viceversa.

5.2. DECLARACIÓN DE VECTORES

Como ya se vio, la forma más sencilla de crear un vector es escribir sus elementos entre corchetes (separados con comas), por ejemplo, con las siguientes instrucciones se crean tres vectores (en la calculadora Javascript):

```
>>a=[5.1,2.2,3.5,6.8]
[5.1, 2.2, 3.5, 6.8]
>>b=["a","b","c","d"]
[a, b, c, d]
>>c=[1,3,4,"a","hola",6,"mundo",5.7,8]
[1, 3, 4, a, hola, 6, mundo, 5.7, 8]
```

El tercer vector "c", aparentemente contiene diferentes tipos de datos, pero como ya se explicó en el anterior acápite, en realidad lo único que contiene son punteros y como ya se dijo, los punteros pueden apuntar a diferentes tipos de datos.

Alternativamente (y formalmente más correcto) se pueden crear los vectores con "new Array":

```
>>a=new Array(5.1,2.2,3.5,6.8)
[5.1, 2.2, 3.5, 6.8]
>>b=new Array("a","b","c","d")
[a, b, c, d]
>>c=new Array(1,3,4,"a","hola",6,"mundo",5.7,8)
[1, 3, 4, a, hola, 6, mundo, 5.7, 8]
```

Sin embargo, si se quiere crear un vector con un solo elemento numérico, ese elemento es interpretado como la dimensión del vector (el número de elementos del vector), por ejemplo, la instrucción:

```
>>d=new Array(10)
[undefined, undefined, undefined, undefined, undefined, undefined,
undefined, undefined, undefined, undefined]
```

Crea un vector con 10 elementos inicialmente indefinidos (sin valor), no, como se podría esperar, un vector con el número 10.

Para crear un vector con un elemento (con este método), se crea primero un vector vacío y luego se le asigna el valor al primer elemento:

```
>>d=new Array(); d[0]=10
10

>>d
[10]
```

Como se puede ver en este ejemplo, Javascript añade automáticamente elementos al vector en la posición que se le indica. Si existen elementos sin definir entre la nueva posición y el último elemento añadido, los elementos intermedios quedan sin definir, por ejemplo si al vector "d" se le añade un elemento en la décima posición (recuerde que el primer índice es 0) se obtiene:

```
12
>>d
[10, undefined, undefined, undefined, undefined, undefined, undefined,
undefined, undefined, 12]
```

Con "new Array", la declaración de los vectores "a", "b" y "c", es:

```
>>a=new Array(); a[0]=5.1; a[1]=2.2; a[2]=3.5; a[3]=6.8; a
[5.1, 2.2, 3.5, 6.8]
>>b=new Array(); b[0]="a"; b[1]="b"; b[2]="c"; b[3]="d"; b
[a, b, c, d]
>>c=new Array(); c[0]=1; c[1]=3; c[2]=4; c[3]="a"; c[4]="hola"; c[5]=6;
c[6]="mundo"; c[7]=5; c[8]=7; c[9]=8; c
[1, 3, 4, a, hola, 6, mundo, 5, 7, 8]
```

Donde se ha escrito "a", "b" y "c", al final de cada instrucción, simplemente para mostrar el vector creado. Se obtiene el mismo resultado si se especifica previamente el número de elementos:

```
>>a=new Array(4); a[0]=5.1; a[1]=2.2; a[2]=3.5; a[3]=6.8; a
[5.1, 2.2, 3.5, 6.8]
>>b=new Array(4); b[0]="a"; b[1]="b"; b[2]="c"; b[3]="d"; b
[a, b, c, d]
>>c=new Array(10); c[0]=1; c[1]=3; c[2]=4; c[3]="a"; c[4]="hola";
c[5]=6; c[6]="mundo"; c[7]=5; c[8]=7; c[9]=8; c
[1, 3, 4, a, hola, 6, mundo, 5, 7, 8]
```

Las cuatro formas de declarar vectores son equivalentes, pero desde el punto de vista práctico la primera (escribir los elementos entre corchetes) es la más sencilla, por lo que es la que se usa con más frecuencia.

5.3. PROPIEDADES Y MÉTODOS DE LOS VECTORES

Los vectores en Javascript son objetos y como tales tienen una serie de propiedades y métodos. En este acápite se verán algunas de dichas propiedades y métodos.

La propiedad "**length**", devuelve el número de elementos del vector, así, para el siguiente vector:

```
>>v=[3,2,4,6,2,3,1,7]
[3, 2, 4, 6, 2, 3, 1, 7]
```

Que tiene 8 elementos, la propiedad "length" devuelve dicho número:

```
>>v.length
8
```

El método "**concat(a1,a2,…)**" une los elementos de una, dos o más matrices a los elementos de la matriz desde la cual se llama al método. Por ejemplo, si se crean las siguientes matrices:

```
>>a=[1,2,3,4]
[1, 2, 3, 4]
>>b=[5,6,7]
[5, 6, 7]
>>c=[8,9,10]
[8, 9, 10]
```

Y se quiere unir los elementos de las matrices "b" y "c" a la matriz "a", se escribe:

```
>>d=a.concat(b,c)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Con lo que se crea un nuevo vector, que en este ejemplo ha sido guardado en la variable "d". El vector original "a", desde el cual se llama al método "concat", no es modificado:

```
>>a
[1, 2, 3, 4]
```

El método "**join(s)**", devuelve una cadena (un string) con los elementos del vector desde el cual se llama al método. Para separar los elementos "join" emplea la cadena de caracteres "s". Por ejemplo, con la siguiente instrucción se crea una cadena de caracteres con los elementos del vector "d" separados con una coma y un espacio ", ":

```
>>s=d.join(", ")
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

El método "**push(e1,e2,…)**" añade él o los elementos "e1", "e2",…, al final del vector desde el cual se llama al método. El método devuelve el nuevo número de elementos del vector. Por ejemplo, dado el vector:

```
>>x=[1,2,3,4,5,6]
[1, 2, 3, 4, 5, 6]
```

La siguiente instrucción añade 6 elementos al final del mismo:

```
>>x.push(7,8,9,10,11,12)
12
```

Y como se puede ver devuelve el número 12, que es el nuevo número de elementos del vector:

```
>>x
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

El método "**pop()**", remueve el último elemento del vector, devolviendo el elemento removido. Por ejemplo, la siguiente instrucción quita el último elemento (el número 12) de la matriz "x":

```
>>x.pop()
12
```

Con lo que ahora los elementos del vector "x" son:

```
>>x
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

El método "**shift()**" es similar a "pop", sólo que en lugar de remover el último elemento del vector, remueve el primero, por ejemplo, aplicando este método a la matriz "x", se obtiene:

```
>>x.shift()
1
```

Con lo que el vector "x" queda con los elementos:

```
>>x
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

El método "**unshift(e1,e2,…)**" hace lo contrario que "shift", es decir añade él o los elementos "e1", "e2",…, al principio del vector y devuelve el nuevo número de elementos. Por ejemplo con la siguiente instrucción se añaden los elementos -3, -2, -1, 0 y 1 al principio del vector "x":

```
>>x.unshift(-3,-2,-1,0,1)
15
>>x
[-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

El método "**slice(i,f)**" crea un nuevo vector con los elementos existentes entre el índice "i" y el índice "f" (sin incluir este último). Si no se especifica "f" el vector se crea con todos los elementos del vector a partir de "i". Por ejemplo, dado el siguiente vector:

```
>>v=[1,2,3,4,5,6]
[1, 2, 3, 4, 5, 6]
```

La siguiente instrucción crea el vector "u" con los elementos de "v" que se encuentra entre la posición 2 (el segundo elemento) y 5 (el último elemento, sin incluir dicho elemento):

```
>>u=v.slice(1,5)
[2, 3, 4, 5]
```

La siguiente instrucción crea el vector "w" con los elementos comprendidos entre el elemento 3 (el cuarto) y el último:

```
>>w=v.slice(3)
[4, 5, 6]
```

Cuando el índice es negativo, se crea un vector con ese número de elementos extraídos contando desde el último, por ejemplo, la siguiente instrucción crea el vector "r" con los dos últimos elementos del vector "v":

```
>>r=v.slice(-2)
[5, 6]
```

El método "slice" no modifica la matriz original:

```
>>v
[1, 2, 3, 4, 5, 6]
```

El método "**splice(i,n,e1,e2,e3,…)**", remueve, inserta y/o reemplaza elementos en la matriz a partir del índice (posición) "i". Cuando se extraen elementos "n" especifica el número de elementos a extraer, los cuales son devueltos por el método. Cuando se insertan elementos (n=0) "e1,e2,e3,…" son los elementos a insertar. Cuando se reemplazan elementos (n igual al número de elementos a reemplazar) "e1,e2,e3",…", son los elementos que reemplazan a los "n" elementos que se extraen (los cuales son devueltos por el método).

Por ejemplo la siguiente instrucción inserta los números 8, 9, 10 y 11, en la quinta posición (índice 4) de la matriz "v":

```
>>v.splice(4,0,8,9,10,11)
[]
>>v
[1, 2, 3, 4, 8, 9, 10, 11, 5, 6]
```

Mientras que la siguiente remueve tres elementos a partir de la cuarta posición (los números 4, 8 y 9):

```
>>v.splice(3,3)
[4, 8, 9]
>>v
[1, 2, 3, 10, 11, 5, 6]
```

Y la siguiente reemplaza los números 10 y 11 por 20 y 30:

```
>>v.splice(3,2,20,30)
[10, 11]
>>v
[1, 2, 3, 20, 30, 5, 6]
```

Al igual que "slice", si el índice es negativo las posiciones se cuentan comenzando del último elemento (que viene a ser el elemento -1) hacia atrás, por ejemplo, si el vector es:

```
>>v=[1,2,3,4,5,6,7,8,9,10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

La siguiente instrucción extrae los elementos 5, 6, 7 y 8

```
>>v.splice(-6,4)
[5, 6, 7, 8]
>>v
[1, 2, 3, 4, 9, 10]
```

Y la siguiente los vuelve a insertar:

```
>>v.splice(-2,0,5,6,7,8)
[]
>>v
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

El método "**sort(f)**" ordena los elementos del vector desde el cual se llama al método, el parámetro "f", que es una función, es opcional. Por defecto los elementos son ordenados ascendentemente tratándolos como texto. Por ejemplo, dado el siguiente vector:

```
>>a=["x","a","r","b","z","c"]
[x, a, r, b, z, c]
```

Al llamar al método "sort" se obtiene:

```
>>a.sort()
[a, b, c, r, x, z]
```

No obstante si el vector es numérico, como el siguiente:

```
>>b=[10,2,4,20,5,6,40,70,9]
[10, 2, 4, 20, 5, 6, 40, 70, 9]
```

Al llamar al método "sort" se obtiene:

```
>>b.sort()
[10, 2, 20, 4, 40, 5, 6, 70, 9]
```

Que no es el orden correcto. Esto se debe a que "sort", como se dijo, trata los elementos como texto (no como números). En estos casos, para obtener el orden correcto, se debe mandar también la función "f", normalmente una función literal que recibe dos parámetros y devuelve un valor positivo cuando el primer parámetro es mayor al segundo, cero si son iguales y negativo si el primer parámetro es menor que el segundo. Así en el caso de los números la función simplemente tiene que devolver la resta de los mismos, pues la resta es positiva (mayor a cero) cuando el primer número es mayor que el segundo, es cero si son iguales y es negativa si el primer número es menor al segundo:

```
>>b.sort(function(x,y){return x-y;})
[2, 4, 5, 6, 9, 10, 20, 40, 70]
```

Es importante tomar en cuenta que "sort" no devuelve un nuevo vector ordenado, sino que modifica el vector original:

```
>>b.sort(function(x,y){return x-y;})
[2, 4, 5, 6, 9, 10, 20, 40, 70]
```

Para ordenar el vector en orden inverso (descendente) simplemente se cambia el orden de la resta en la función:

```
>>b.sort(function(x,y){return y-x;})
[70, 40, 20, 10, 9, 6, 5, 4, 2]
```

Sin embargo si lo que se quiere es invertir el orden en que se encuentran los elementos (no ordenarlos), se debe emplear el método "**reverse()**", así para invertir los elementos del vector:

```
>>c=[8,1,20,4,8,54,12,9,20]
[8, 1, 20, 4, 8, 54, 12, 9, 20]
```

Se escribe:

```
>>c.reverse()
[20, 9, 12, 54, 8, 4, 20, 1, 8]
```

Al igual que "sort", "reverse" no devuelve un nuevo vector, sino que cambia el orden del vector original:

```
>>c
[20, 9, 12, 54, 8, 4, 20, 1, 8]
```

Si bien no todas estos métodos se emplean intensivamente en los programas, es conveniente conocerlos para poder emplearlos cuando sea conveniente.

5.4. ECUACIÓN CUADRÁTICA

La ecuación cuadrática es la ecuación polinomial más sencilla que existe:

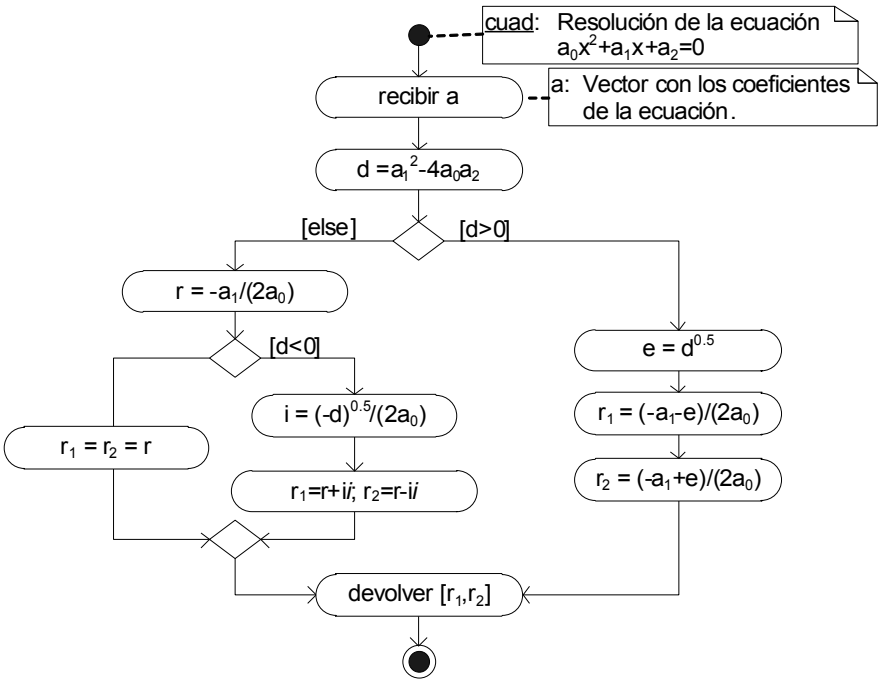
$$a_0 x^2 + a_1 x + a_2 = 0 \tag{5.2}$$

Y como se sabe, sus soluciones pueden ser encontradas con la expresión:

$$x_{1,2} = \frac{-a_1 \pm \sqrt{a_1^2 - 4 \cdot a_0 \cdot a_2}}{2 \cdot a_0} \tag{5.3}$$

El tipo de resultados que se obtiene depende del valor de la expresión contenida en la raíz cuadrada, conocida como discriminante: a) Si es positiva se tienen dos soluciones reales, b) Si es cero las dos soluciones son iguales y c) Si es negativa las dos soluciones son complejas (un par conjugado).

El algoritmo que resuelve esta ecuación, considerando esos casos, es:



Donde "*i*" (en cursiva) hace referencia al valor imaginario. El código respectivo (añadido a la librería "mat205.js"):

```
function cuad(a){
  var d,r,r1,r2,i;
  d=a[1]*a[1]-4*a[0]*a[2];
  if (d>0) {
    var e=sqrt(d); r1=(-a[1]-e)/(2*a[0]); r2=(-a[1]+e)/(2*a[0]);}
  else {
    r=-a[1]/(2*a[0]);
    if (d<0) {
      i=sqrt(-d)/(2*a[0]);r1=new Complex(r,i);r2=new Complex(r,-i);}
    else {
      r1=r2=r;
    }
  }
  return [r1,r2];
}
```

Observe que para crear números complejos se ha empleado la instrucción "new Complex(parte real, parte imaginaria)", este nuevo tipo de dato (en realidad un objeto) ha sido añadido también a la librería "mat205.js", y consta de dos propiedades (datos): la parte real "r" y la parte imaginaria "i". Este objeto tiene además muchas otras funciones que se irán presentando según se requiera.

Para probar el programa se resuelven las siguientes ecuaciones:

$$x^2-10x+21=0$$

$$x^2-14x+49=0$$

$$x^2+2x+3=0$$

```
>>cuad([1,-10,21])
[3, 7]
>>cuad([1,-14,49])
[7, 7]
>>cuad([1,2,3])
[-1+1.4142135623730951i, -1-1.4142135623730951i]
```

Tanto en la consola de Google Chrome como en la de Mozilla Firefox los resultados complejos se muestran de forma diferente.

5.5. ECUACIÓN CÚBICA (MÉTODO DE CARDANO)

La ecuación cúbica es otra de las ecuaciones polinomiales que aparece frecuentemente en la solución de problemas en el campo de la ingeniería y que puede ser resuelta con métodos directos:

$$a_0 x^3 + a_1 x^2 + a_2 x + a_3 = 0 \tag{5.4}$$

La solución de este tipo de ecuaciones fue propuesta por primera vez por Cardano (en 1545), razón por la cual se la conoce también como el método de Cardano. En este método la ecuación (54) debe ser llevada primero a la forma:

$$x^3 + a_1 x^2 + a_2 x + a_3 = 0 \tag{5.5}$$

Lo que se consigue simplemente dividiendo todos los coeficientes entre el primero (a_0), luego, con estos nuevos coeficientes, se calculan los siguientes parámetros:

$$Q = \frac{3a_2 - a_1^2}{9} \tag{5.6}$$

$$R = \frac{9a_1 a_2 - 27a_3 - 2a_1^3}{54} \tag{5.7}$$

$$S = \sqrt[3]{R + \sqrt{Q^3 + R^2}} \tag{5.8}$$

$$T = \sqrt[3]{R - \sqrt{Q^3 + R^2}} \tag{5.9}$$

Y con estos parámetros se calculan las tres soluciones:

$$x_1 = S + T - \frac{1}{3}a_1 \tag{5.10}$$

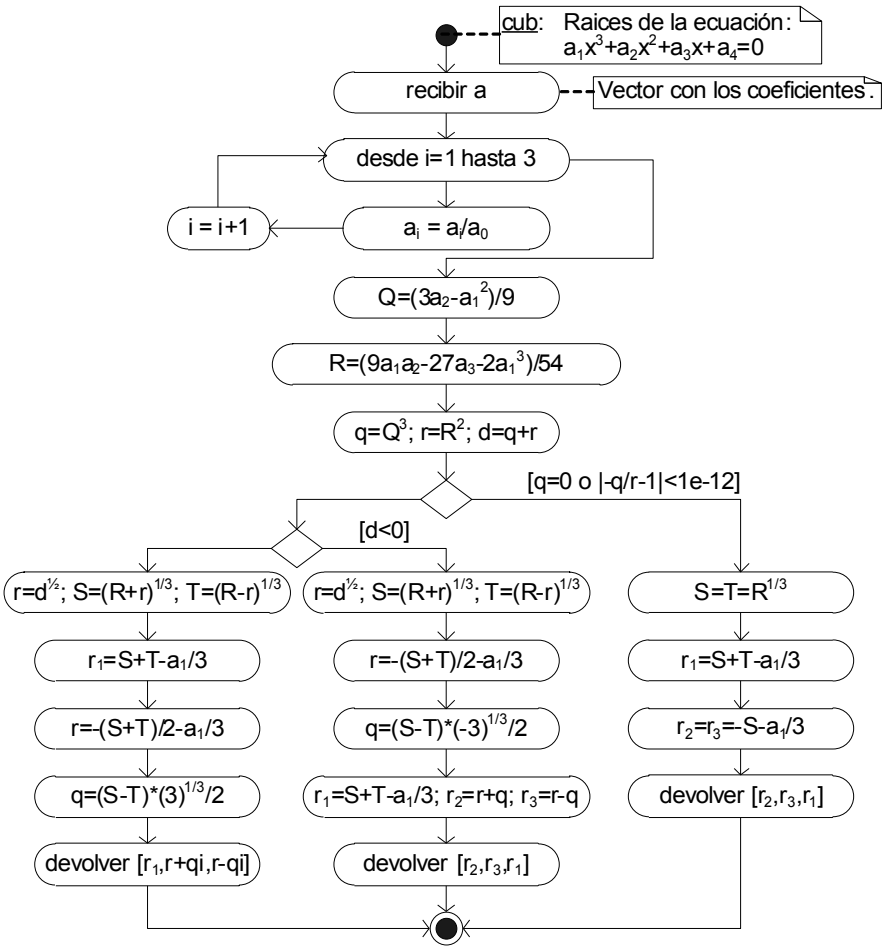
$$x_2 = -\frac{1}{2}(S + T) - \frac{1}{3}a_1 + \frac{1}{2}\sqrt{-3}(S - T) \tag{5.11}$$

$$x_2 = -\frac{1}{2}(S + T) - \frac{1}{3}a_1 - \frac{1}{2}\sqrt{-3}(S - T) \tag{5.12}$$

En este método, la suma "Q^3+R^2", constituye el discriminante: a) Si es positivo una de las soluciones es real (x_1) y las otras dos imaginarias (x_2 y x_3); b) Si es negativo, las tres soluciones son reales y distintas y c) Si es cero todas de las soluciones son reales y dos de ellas (x_2 y x_3) son iguales.

Debido a errores de redondeo, el discriminante casi nunca es exactamente cero, sino sólo aproximadamente y esta situación debe ser tomada en cuenta al momento de elaborar el programa.

El algoritmo que automatiza el cálculo es el siguiente:



Y el código respectivo (añadido a la librería "mat205.js") es:

```
function cub(a){
  var i,Q,R,S,T,d,r1,r2,r3,q,r;
  for (i=1;i<4;i++){a[i]/=a[0];}
  Q=(3*a[2]-a[1]*a[1])/9;
  R=(9*a[1]*a[2]-27*a[3]-2*a[1]*a[1]*a[1])/54;
  q=Q*Q*Q; r=R*R; d=q+r;
  if (q==0 || abs(-q/r-1)<1e-12) {
    S=T=cbrt(R); r1=S+T-a[1]/3; r2=r3=-S-a[1]/3;
    return [precision(r2),precision(r3),precision(r1)];}
  else if (d<0) {
    r=sqrt(d); S=cbrt(add(R,r)); T=cbrt(sub(R,r));
    r=-add(div(add(S,T),2),a[1]/3);
    q=div(mul(sub(S,T),sqrt(-3)),2);
    r1=sub(add(S,T),a[1]/3); r2=add(r,q); r3=sub(r,q);
    return [precision(r2),precision(r3),precision(r1)];}
  else {
```

```
    r=sqrt(d); S=cbrt(R+r); T=cbrt(R-r);
    r1=S+T-a[1]/3; r=-(S+T)/2-a[1]/3; q=(S-T)*sqrt(3)/2;
    r2=new Complex(r,q); r3=new Complex(r,-q);
    return [precision(r1),precision(r2),precision(r3)];}
}
```

Para probar el programa se resolverán las siguientes ecuaciones cúbicas (en la calculadora):

$$x^3+2x^2+3x+4=0$$
$$x^3+3x^2+7x+9=0$$
$$x^3-6x^2+11x-6=0$$
$$x^3-22x^2+145x-300=0$$
$$x^3+21x^2+147+343=0$$

```
>>cub([1,2,3,4])
[-1.65062919143939, -0.174685404280306+1.54686888723l4i,
-0.174685404280306-1.54686888723l4i]
>>cub([1,3,7,9])
[-1.84770759813957, -0.576146200930217+2.13048260470666i,
-0.576146200930217-2.13048260470666i]
>>cub([1,-6,11,-6])
[1, 2, 3]
>>cub([1,-22,145,-300])
[5, 5, 12]
>>cub([1,21,147,343])
[-7, -7, -7]
```

5.6. MÉTODO DE NEWTON-RAPHSON

Como se dijo, los métodos estudiados en los anteriores temas pueden ser empleados también para encontrar las soluciones de ecuaciones polinomiales, siendo particularmente útil el método de Newton-Raphson pues tanto el valor de la función, como el de su derivada, pueden ser calculadas por división sintética.

5.6.1. División sintética

Si se quiere calcular el valor del siguiente polinomio y el de su derivada para "x" igual a 1.2:

$$P_3(x)=x^3+x^2-3x-3=0$$

Se divide el polinomio entre "x-2" y (para obtener su derivada) el cociente resultante se vuelve a dividir entre "x-2":

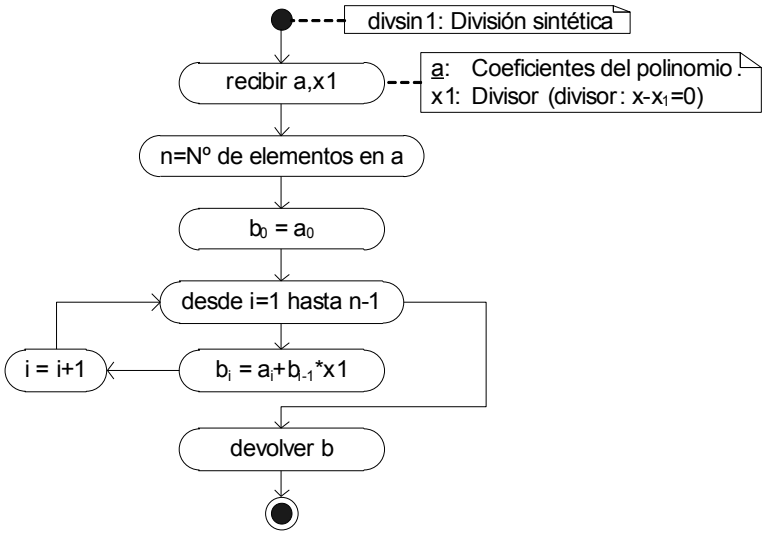
$$\begin{array}{llllll} x=2 & 1 & 1 & -3 & -3 \\ & & 2 & 6 & \underline{6} \\ \hline & 1 & 3 & 3 & \underline{3} \\ & & 2 & \underline{10} & \\ \hline & 1 & 5 & \underline{13} & \end{array}$$

El residuo de la primera división (3) es el valor de la función para "x=2" y el residuo de la segunda división (13) es el valor de la derivada primera de la función para "x=2".

Como se puede deducir fácilmente de este ejemplo, si "*a*" son los coeficientes del polinomio original "n" el grado del polinomio y "b" los coeficientes resultantes de la división sintética, dichos coeficientes se calculan con:

$$\begin{aligned} b_0 &= a_0 \\ b_i &= a_i + b_{i-1} \cdot x_1 \quad \{i = 1 \rightarrow n \end{aligned} \tag{5.13}$$

Puesto que en ocasiones puede ser útil el polinomio resultante ("b"), resulta conveniente contar con una función para este fin, siendo el algoritmo para llevar a cabo esta tarea el siguiente:



Cuyo código (añadido a la unidad "mat205.js") es:

```
function divsin1(a,x1) {
  var n=a.length, b=new Array(n);
  b[0]=a[0];
  for (var i=1;i<n;i++) b[i]=add(a[i],mul(b[i-1],x1));
  return b;
}
```

Observe que en esta función se han empleado las funciones "add" y "mul" en lugar de los operadores "+" y "*", para que sea capaz de operar con números complejos (las funciones "add", "sub", "mul" y "div" son los equivalentes a los operadores aritméticos "+", "-", "*", y "/", sólo que operan tanto con números reales como complejos.

Para probar el programa se realiza la división sintética del ejemplo manual:

```
>>a=[1,1,-3,-3]; b=divsin1(a,2)
[1, 3, 3, 3]
>>divsin1(b.slice(0,3),2)
[1, 5, 13]
```

Que son los mismos resultados del ejemplo.

Como otro ejemplo, se calcula el valor del siguiente polinomio, el de su derivada y el polinomio residual para "x" igual a 3.1:

$$P_7(x) = 7 \cdot x^7 + 3 \cdot x^5 - 8 \cdot x^4 + 3 \cdot x - 9 = 0$$

```
>>a=[7,0,3,-8,0,0,3,-9]; b=divsin1(a,3.1)
[7, 21.7, 70.27, 209.837, 650.4947, 2016.53357, 6254.254067000001,
19379.1876077]
>>divsin1(b.slice(0,6),3.1)
[7, 43.4, 204.81, 844.748, 3269.2135, 12151.09542]
```

En consecuencia el valor del polinomio para "x" igual 3.1 es: 19379.1876077, el valor de la derivada primera (para el mismo valor de "x") es: 43922.649869. Y el polinomio residual, los coeficientes del vector "b" sin el residuo, es:

$$7x^6+ 21.7x^5+ 70.27x^4+ 209.837x^3+ 650.4947x^2+ 2016.53357x+ 6254.254067000001=0$$

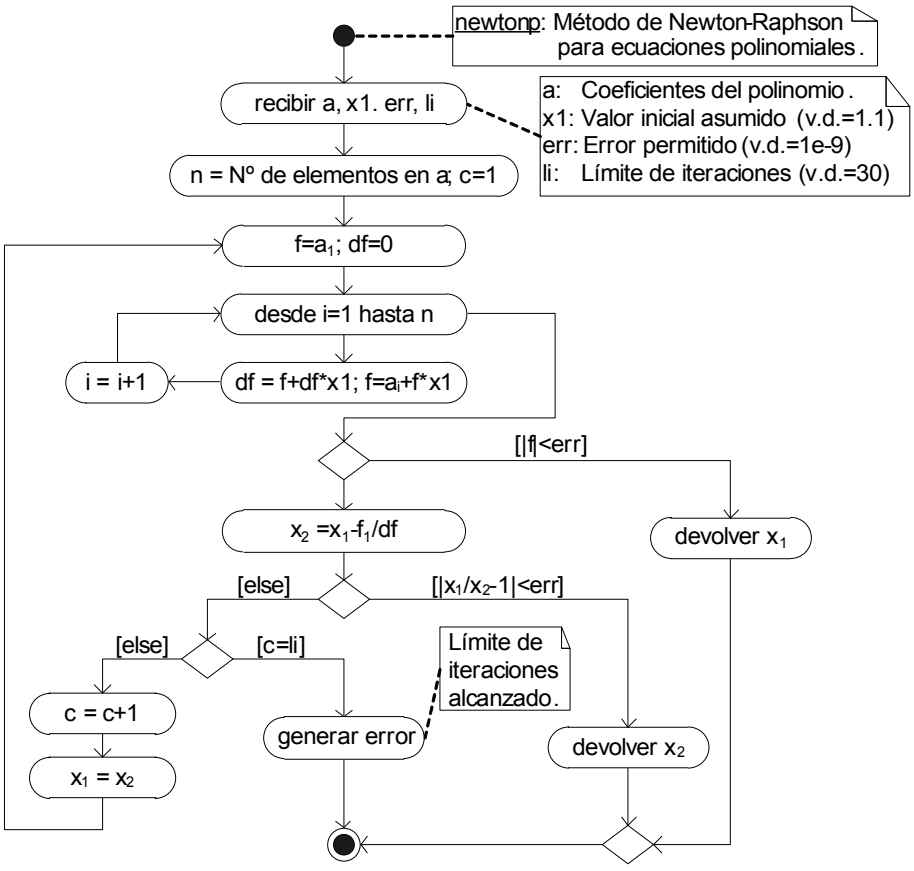
Una vez repasado el proceso de división sintética se pasa a modificar el algoritmo de Newton Raphson para el caso específico de las ecuaciones polinomiales.

5.6.2. Algoritmo y código

El método de Newton-Raphson en si no cambia, sólo lo hace la forma en que se calculan el valor de la función y el de su derivada (por división sintética).

Además, como en el método de Newton-Raphson no se requiren los polinomios residuales, no se guardan dichos valores, por lo que el proceso de división sintética se modifica en este sentido.

El algoritmo, con las consideraciones antes mencionadas, es:



Siendo el código respectivo, añadido a la unidad "mat205.js":

```
function newtonp(a,x1,err,li) {
  var i,f,df,x2,c,n;
  if (x1 == null) {x1=1.1;}
  if (err == null) {err=1e-9;}
  if (li == null) {li=30;}
  n=a.length; c=1;
  while (true) {
    f=a[0];df=0;
    for(i=1;i<n;i++){
      df=add(f,mul(df,x1));f=add(a[i],mul(f,x1));}
    if (abs(f)<err) {return x1;}
    x2=sub(x1,div(f,df));
    if (abs(sub(div(x1,x2),1))<err) {return x2;}
    if (c++ == li) throw new Error("newtonp no converge, x="+x2);
    x1=x2;
  }
}
```

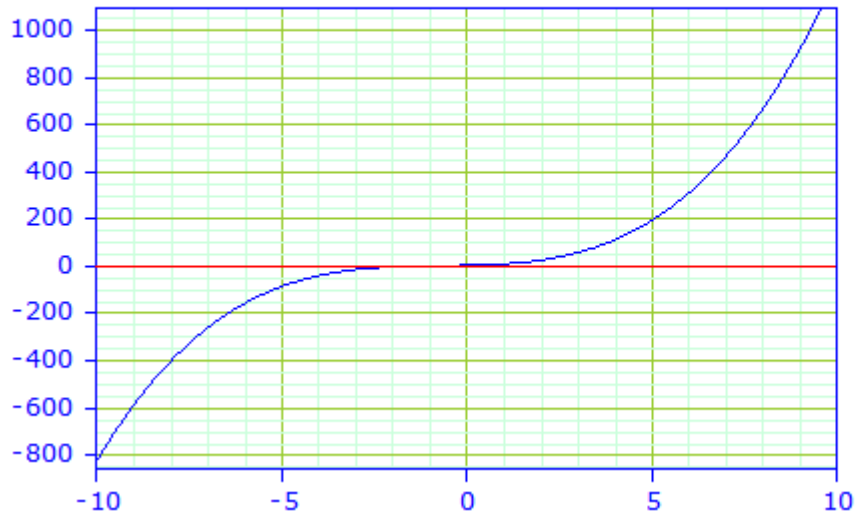
Observe que las operaciones se llevan a cabo con las funciones que trabajan con números complejos, razón por la cual esta versión del método puede encontrar soluciones tanto reales como complejas, sin embargo, al igual que el método original, requiere valores asumidos para comenzar el proceso iterativo.

Por ejemplo, para encontrar las soluciones de la ecuación cúbica:

$$P(x)=x^3+2x^2+3x+4=0$$

Que al ser de tercer grado tiene 3 soluciones, sin embargo sólo una de ellas es real (tal como se puede verificar con la gráfica de la función):

```
>>f=function(x){return x*x*x+2*x*x+3*x+4;}; y=function(x){return 0;};
plot([f,y],-10,10)
```



Primero se encuentra esta solución empleando un valor inicial obtenido de la gráfica:

```
>>precision(newtonp([1,2,3,4],-1.5),9)
-1.65062919
```

Siendo esta la solución real (con la precisión por defecto, 9 dígitos). Ahora se pueden encontrar la segunda solución (que es compleja) y para ello se debe asumir un valor inicial complejo. Para crear un número complejo se emplea la instrucción "new Complex(parte_real,parte_imaginaria)", en este caso, al no contar con ningún valor próximo a la solución, se asume un valor cualquiera, por ejemplo "1.1+1.1i":

```
>>precision(newtonp([1,2,3,4],new Complex(1.1,1.1)),9)
-0.174685404+1.54686889i
```

Esta es la segunda, de las tres soluciones, del polinomio (con los 9 dígitos de precisión por defecto), pero dado que **las soluciones complejas siempre vienen en pares conjugados**, no es necesario calcular la tercera, pues simplemente es el par conjugado de la segunda solución, es decir que la tercera solución es:

```
>>ans.conj()
-0.174685404-1.54686889i
```

5.7. MÉTODO DE BAIRSTOW

Como ya se vio en el anterior tema, el método de Newton permite encontrar las soluciones reales y complejas de una ecuación polinomial, sin embargo, en dicho proceso se realizan operaciones con números complejos. El método de Bairstow permite evitar estas operaciones al calcularlas de 2 en 2 con la ecuación cuadrática.

Con ese fin, se divide el polinomio entre una ecuación de segundo grado (x^2-rx-s), haciendo variar los coeficientes "r" y "s" hasta que el residuo es cero (o casi cero), es decir hasta que se cumple que:

$$\frac{P_n(x)}{x^2-r\cdot x-s}=Q_{n-2}(x)\cdot\left(x^2-r\cdot x-s\right)=0 \qquad (5.14)$$

Donde "Q_{n-2}" es el polinomio residual (de grado n-2). Entonces, para calcular dos de las soluciones, se resuelve la ecuación de segundo grado.

Tanto en el método de Bairstow como en el método de Newton, se puede continuar el proceso con el polinomio residual calculando otras dos soluciones (u otra solución en el caso de Newton) y continuar así hasta que el grado del polinomio residual sea menor a 4 (en cuyo caso las soluciones se calculan con la ecuación cuadrática o cúbica, según corresponda). Sin embargo, este procedimiento requiere que las soluciones sean exactas, pues de lo contrario el error del primer proceso se propaga al segundo, y así sucesivamente, dando lugar a soluciones erróneas.

La relación entre los valores iniciales asumidos (x_1 y x_2) y la ecuación cuadrática x^2-rx-s, se puede deducir fácilmente de la igualdad:

$$\left(x-x_1\right)\cdot\left(x-x_2\right)=x^2-r\cdot x-s$$
$$x^2-\left(x_1+x_2\right)\cdot x+x_1\cdot x_2=x^2-r\cdot x-s$$

En consecuencia se cumple que:

$$r=x_1+x_2$$
$$s=-x_1\cdot x_2 \qquad (5.15)$$

Para llevar a cabo la división (ecuación 514), el método más eficiente es, una vez más, el de la división sintética, por lo que primero se repasa dicho procedimiento.

5.7.1. División sintética de segundo orden

En la división sintética de segundo grado se divide el polinomio entre la ecuación cuadrática: $x^2-rx-s=0$, es decir que los divisores son los valores de "r" y "s".

Para recordar el procedimiento se dividirá el siguiente polinomio entre x^2+x+1, o $x^2-(-1)x-(-1)$, por lo tanto los divisores son r=-1 y s=-1):

$$P_4(x)=x^4-1.1\cdot x^3+2.3\cdot x^2+0.5\cdot x+3.3=0$$

$$\begin{array}{l|rrrrr} r=-1 & 1 & -1.1 & 2.3 & 0.5 & 3.3 \\ \underline{s=-1} & & -1.0 & 2.1 & -3.4 & 0.8 \\ & & & -1.0 & 2.1 & -3.4 \\ \hline & 1 & -2.1 & 3.4 & -0.8 & 0.7 \end{array}$$

Si se denomina "*b*" a los coeficientes resultantes de la primera división, entonces el residuo es: $b_{n-1}(x-r)+b_n$, donde "n" es el grado del polinomio. Con los resultados del ejemplo el residuo es: -0.8*(x-1)+0.7 = -0.8*x-0.1.

Una segunda división sintética permite calcular las derivadas parciales de la función con relación a los coeficientes "r" y "s" (estos valores son de utilidad en el método de Bairstow):

$$\begin{array}{l|rrrr} r=-1 & 1 & -2.1 & 3.4 & -0.8 \\ \underline{s=-1} & & -1.0 & 3.1 & -5.5 \\ & & & -1.0 & 3.1 \\ \hline & 1 & -3.1 & 5.5 & -3.2 \end{array}$$

Si se denomina "c" a los coeficientes resultantes de esta segunda división se cumple que: $\partial b_3/\partial r = c_2 = 5.5$; $\partial b_3/\partial s = c_1 = -3.1$ y en general:

$$\begin{aligned} \frac{\partial b_i}{\partial r} &= c_{i-1} \\ \frac{\partial b_i}{\partial s} &= c_{i-2} \end{aligned} \qquad (5.16)$$

Analizando las operaciones realizadas, se deducen fácilmente las ecuaciones generales para el cálculo de los coeficientes "b":

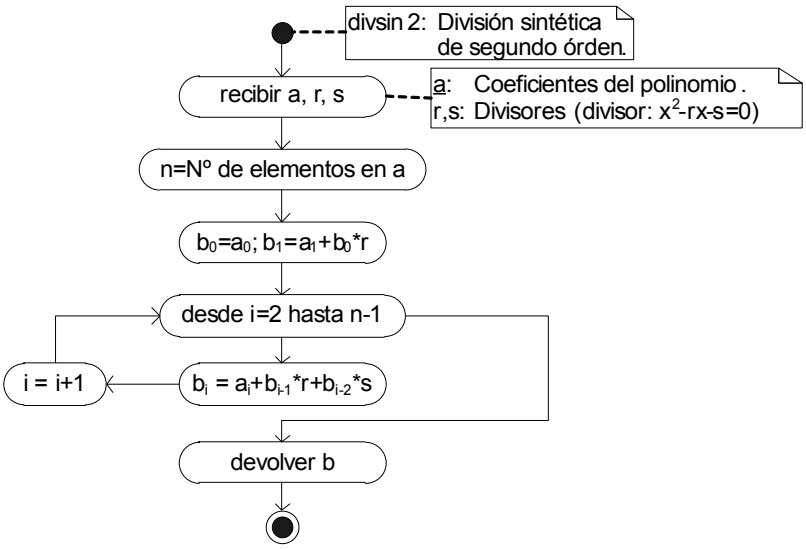
$$\begin{aligned} b_0 &= a_0 \\ b_1 &= a_1+b_0\cdot r \\ b_i &= a_i+b_{i-1}\cdot r+b_{i-2}\cdot s \quad \{i=2\rightarrow n \end{aligned} \qquad (5.17)$$

Donde "n" es el grado del polinomio, siendo el residuo de la división:

$$residuo=b_{n-1}\cdot(x-r)+b_n=b_{n-1}\cdot x+\left(b_n-b_{n-1}\cdot r\right) \qquad (5.18)$$

El algoritmo que automatiza el proceso de división sintética se presenta en la siguiente página y el código respectivo, añadido a la librería "mat205.js", es:

```
function divsin2(a,r,s) {
  var n=a.length,b=new Array(n);
  b[0]=a[0]; b[1]=add(a[1],mul(b[0],r));
  for (var i=2;i<n;i++) {
    b[i]=add(a[i],add(mul(b[i-1],r),mul(b[i-2],s)));}
```




```
    return b;
}
```

Que, como se puede observar, ha sido implementado de manera que pueda trabajar con números complejos.

Haciendo la prueba con los datos del ejemplo manual (redondeando los resultado al décimo quinto dígito después del punto) se obtiene:

```
>>b=round(divsin2([1,-1.1,2.3,0.5,3.3],-1,-1),15)
[1, -2.1, 3.4, -0.8, 0.7]
```

Que son los resultados obtenidos en el ejemplo.

Tomando los primeros "n-1" elementos del vector resultante (de "b") se puede llevar a cabo una segunda división sintética:

```
>>c=round(divsin2(b.slice(0,4),-1,-1),15)
[1, -3.1, 5.5, -3.2]
```

Que, una vez más, son los resultados obtenidos en el ejemplo manual.

Habiendo repasado el procedimiento para realizar la división sintética de segundo grado, se continua con el desarrollo del método.

5.7.2. Ecuaciones del método de Bairstow

Para que la ecuación (5.14) se cumpla, el residuo de la división (ecuación 5.18) debe ser cero. Ello se puede lograr expandiendo los coeficientes "b_n" y "b_{n+1}" en series de Taylor, como funciones de los coeficientes "r" y "s":

$$b_{n-1}(r+\Delta r, s+\Delta S)=b_{n-1}(r,s)+\frac{\partial}{\partial r}\left(b_{n-1}(r,s)\right)\Delta r+\frac{\partial}{\partial s}\left(b_{n-1}(r,s)\right)\Delta s+\cdots+\infty=0$$

$$b_{n}(r+\Delta r, s+\Delta S)=b_{n}(r,s)+\frac{\partial}{\partial r}\left(b_{n}(r,s)\right)\Delta r+\frac{\partial}{\partial s}\left(b_{n}(r,s)\right)\Delta s+\cdots+\infty=0$$

Donde Δr y Δs son los valores que deben sumarse a "r" y "s" para que tanto "b_n" como "b_{n+1}" sean cero. Por supuesto, no es posible en la práctica resolver este sistema (que es más complejo inclusive que el problema

original), por eso sólo se toman los términos de primer orden con lo que el sistema queda en la forma:

$$\begin{aligned} b_{n-1}+\frac{\partial b_{n-1}}{\partial r}\Delta r+\frac{\partial b_{n-1}}{\partial s}\Delta s=0 \\ b_n+\frac{\partial b_n}{\partial r}\Delta r+\frac{\partial b_n}{\partial s}\Delta s=0 \end{aligned} \tag{5.19}$$

Las derivadas parciales de este sistema son los coeficientes resultantes de la segunda división sintética.

Reemplazando las igualdades de la ecuación (5.16), en la ecuación (5.19), se obtiene:

$$\begin{aligned} c_{n-2}\Delta r+c_{n-3}\Delta s=-b_{n-1} \\ c_{n-1}\Delta r+c_{n-2}\Delta s=-b_n \end{aligned} \tag{5.20}$$

Que es un sistema lineal de dos ecuaciones con dos incógnitas (Δr y Δs), cuya solución es:

$$\begin{aligned} \Delta r=\frac{b_n\cdot c_{n-3}-b_{n-1}\cdot c_{n-2}}{c_{n-2}^2-c_{n-1}\cdot c_{n-3}} \\ \Delta s=\frac{-b_n-c_{n-1}\cdot\Delta r}{c_{n-2}} \end{aligned} \tag{5.21}$$

Al sumar estos valores a "r" y "s" no se consigue que "b_{n-1}" y "b_n" sean cero, pues al haber truncado la serie de Taylor en los términos de primer orden, los valores de "Δr" y "Δs" son solo aproximados. Por eso el cálculo de los valores de "r" y "s" es iterativo, es decir, partiendo de valores iniciales asumidos, se calculan nuevos valores de "r" y "s" (r=r+Δr y s=s+Δs) hasta que los valores de "b_{n-1}" y "b_n" sean casi cero o hasta que los valores de "r" y "s", de dos iteraciones consecutivas, sean casi iguales.

Para comprender mejor el procedimiento, se encontrarán dos de la soluciones de la siguiente ecuación con 7 dígitos de precisión.

$$P_4(x)=x^4-1.1\cdot x^3+2.3\cdot x^2+0.5\cdot x+3.3=0$$

Dado que es más fácil asumir valores iniciales para las soluciones, que para los coeficientes de la ecuación cuadrática ("r" y "s"), se asumirán dichos valores y los coeficientes "r" y "s" se calcularán con la ecuación (5.15).

Los valores asumidos son: x1=0.1+0.1i y x2=0.1-0.1i, en consecuencia los valores de "r" y "s" (calculados con operadores complejos, en la consola de Chrome) son:

```
> x1=new Complex(0.1,0.1); x2=x1.conj(); r=add(x1,x2); s=neg(mul(x1,x2));
  precision([r,s],7)
  [0.2, -0.02]
```

El vector de coeficientes y el grado del polinomio son:

```
> a=[1,-1.1,2.3,0.5,3.3]; n=a.length-1
  4
```

Con estos valores se calculan los valores de "b" y "c" por división sintética y se muestran los valores de "b" para verificar si los valores de "b_{n-1}" y "b_n" se van aproximando a cero:

```
> b=divsin2(a,r,s); c=divsin2(b.slice(0,n),r,s); precision(b,7)
  [1, -0.9, 2.1, 0.938, 3.4456]
```

Como era de esperar, al ser la primera iteración, los valores están aún lejos de cero, por lo que el proceso continua con el cálculo de los nuevos valores de "r" y "s":

```
> dr=(b[n]*c[n-3]-b[n-1]*c[n-2])/(sqr(c[n-2])-c[n-1]*c[n-3]); ds=(-b[n]-c[n-
  1]*dr)/c[n-2]; r+=dr; s+=ds; precision([r,s],7)
  [-0.7000425, -1.174404]
```

Como también era de esperar, los nuevos valores de "r" y "s" no son iguales a los anteriores, por lo que se realiza otra iteración.

```
> b=divsin2(a,r,s); c=divsin2(b.slice(0,n),r,s); precision(b,7)
  [1, -1.800043, 2.385703, 0.943883, -0.1625362]
> dr=(b[n]*c[n-3]-b[n-1]*c[n-2])/(sqr(c[n-2])-c[n-1]*c[n-3]); ds=(-b[n]-c[n-
  1]*dr)/c[n-2]; r+=dr; s+=ds; precision([r,s],7)
  [-0.8798293, -1.009829]
```

Una vez más, ni los valores de "b_{n-1}" y "b_n" son cercanos a cero, ni los nuevos valores de "r" y "s" son iguales a los anteriores, por lo que el proceso se repite hasta que se cumpla una de las condiciones (o las dos):

```
> b=divsin2(a,r,s); c=divsin2(b.slice(0,n),r,s); precision(b,7)
  [1, -1.979829, 3.032083, -0.1684267, 0.3863017]
> dr=(b[n]*c[n-3]-b[n-1]*c[n-2])/(sqr(c[n-2])-c[n-1]*c[n-3]); ds=(-b[n]-c[n-
  1]*dr)/c[n-2]; r+=dr; s+=ds; precision([r,s],7)
  [-0.8999029, -1.100583]
> b=divsin2(a,r,s); c=divsin2(b.slice(0,n),r,s); precision(b,7)
  [1, -1.999903, 2.999135, 0.002128613, -0.002713253]
> dr=(b[n]*c[n-3]-b[n-1]*c[n-2])/(sqr(c[n-2])-c[n-1]*c[n-3]); ds=(-b[n]-c[n-
  1]*dr)/c[n-2]; r+=dr; s+=ds; precision([r,s],7)
  [-0.8999999, -1.1]
> b=divsin2(a,r,s); c=divsin2(b.slice(0,n),r,s); precision(b,7)
  [1, -2, 3, -1.488659e-7, 8.343186e-7]
> dr=(b[n]*c[n-3]-b[n-1]*c[n-2])/(sqr(c[n-2])-c[n-1]*c[n-3]); ds=(-b[n]-c[n-
  1]*dr)/c[n-2]; r+=dr; s+=ds; precision([r,s],7)
  [-0.9, -1.1]
> b=divsin2(a,r,s); c=divsin2(b.slice(0,n),r,s); precision(b,7)
  [1, -2, 3, 3.552714e-15, -4.485301e-14]
> dr=(b[n]*c[n-3]-b[n-1]*c[n-2])/(sqr(c[n-2])-c[n-1]*c[n-3]); ds=(-b[n]-c[n-
  1]*dr)/c[n-2]; r+=dr; s+=ds; precision([r,s],7)
  [-0.9, -1.1]
```

En esta última iteración se cumplen las dos condiciones: los valores de b_{n-1} y b_n son cercanos a cero (tienen 14 y 13 ceros respectivamente) y los nuevos valores de "r" y "s" son iguales a los anteriores (en los 7 dígitos con los que se muestran los resultados), en consecuencia el proceso concluye habiéndose encontrado los valores "r" y "s" con los cuales se cumple la ecuación (5.14).

Ahora con estos valores y la ecuación cuadrática, se pueden calcular dos de las soluciones:

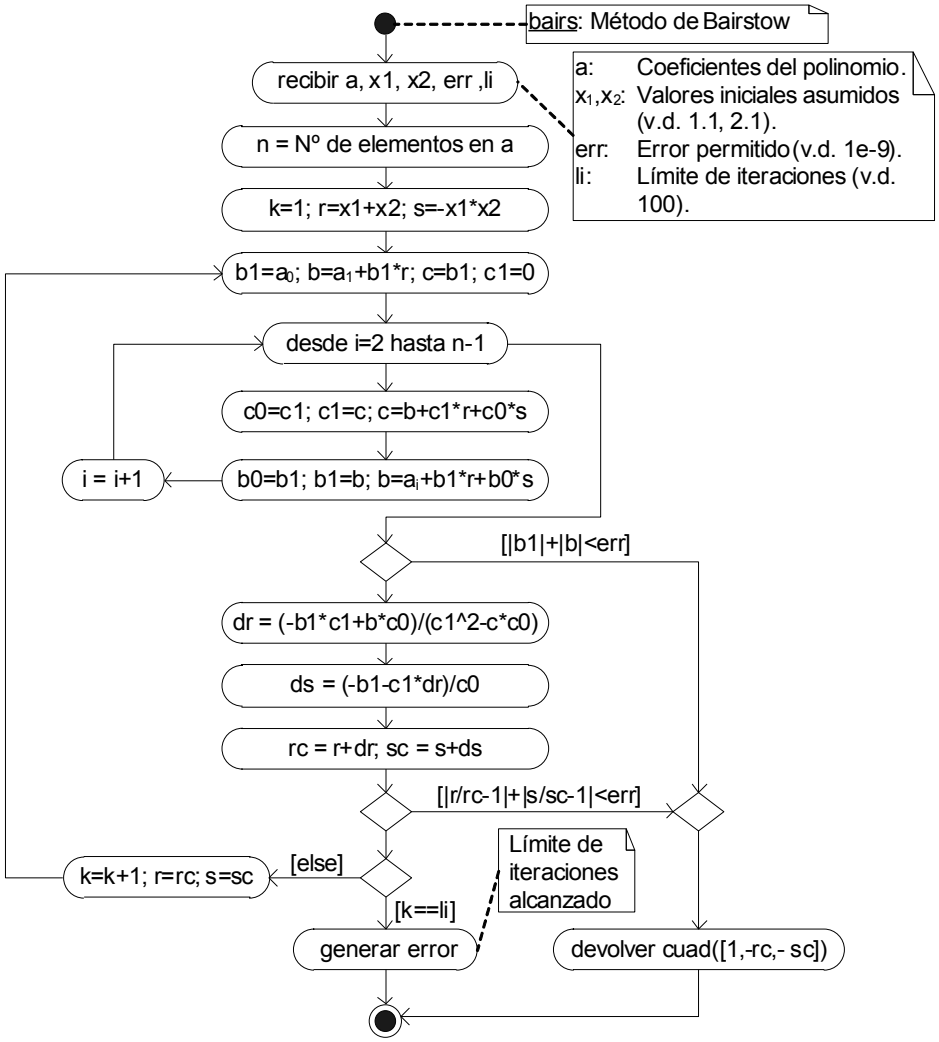
```
> cuad([1,-r,-s])
  [▼Complex                  , ▼Complex                  ]
      i: 0.9473647660748209        i: -0.9473647660748209
      r: -0.44999999999999996      r: -0.44999999999999996
```

Que con 7 dígitos de precisión es:

```
> precision(cuad([1,-r,-s]),7)
[▼Complex            , ▼Complex            ]
     i: 0.9473648            i: -0.9473648
     r: -0.45                r: -0.45
```

5.7.3. Algoritmo y código

Como se ha visto en el ejemplo manual, en el método de Bairstow sólo se emplean los dos últimos elementos de la primera división sintética (el residuo) y los tres últimos elementos de la segunda (las derivadas parciales), por lo que en realidad no es necesario guardar los otros elementos. Ese aspecto se ha tomado en cuenta en el siguiente algoritmo, donde sólo se guardan dichos valores.



El código respectivo, añadido a la unidad "mat205.js", es:

```
function bairstow(a,x1,x2,err,li) {
  var i,k,r,s,b0,b1,b,c0,c1,c,dr,ds,rc,sc,n;
  if (x1==null) {x1=1.1;}
  if (x2==null) {x2=2.1;}
```

```
  if (err==null) {err=1e-9;}
  if (li==null) {li=100;}
  n=a.length;
  k=1; r=add(x1,x2); s=neg(mul(x1,x2));
  while (true) {
    b1=a[0];b=a[1]+b1*r;c=b1;c1=0;
    for (i=2;i<n;i++) {
      c0=c1;c1=c;c=b+c1*r+c0*s;
      b0=b1;b1=b;b=a[i]+b1*r+b0*s;}
    if (abs(b1)+abs(b)<err) break;
    dr=(-b1*c1+b*c0)/(c1*c1-c*c0); ds=(-b1-c1*dr)/c0;
    rc=r+dr; sc=s+ds;
    if (abs(r/rc-1)+abs(s/sc-1)<err) break;
    if (k++ == li) throw new Error("Bairstow no converge");
    r=rc; s=sc; k++;
  }
  return cuad([1,-r,-s]);
}
```

Haciendo correr el programa con los datos del ejemplo manual (encontrando y mostrando las soluciones con 7 dígitos de precisión) se obtiene:

```
> precision(bairstow([1,-1.1,2.3,0.5,3.3],new Complex(0.1,0.1),new
  Complex(0.1,-0.1),1e-7),7)
  [▼Complex            , ▼Complex            ]
      i: 0.9473648           i: -0.9473648
      r: -0.45               r: -0.45
```

Que coincide con los resultados encontrados en el ejemplo.

Como se dijo, una vez encontradas dos soluciones, se puede emplear el polinomio residual para calcular otras dos, pero, para minimizar los errores, se debe trabajar con la máxima precisión posible (16 dígitos):

```
> a=[1,-1.1,2.3,0.5,3.3]; n=a.length
  5
> x=bairstow(a,new Complex(0.1,0.1),new Complex(0.1,-0.1),1e-16)
  [▼Complex                , ▼Complex                ]
      i: 0.9473647660748208      i: -0.9473647660748208
      r: -0.45                   r: -0.45
```

Con estas dos soluciones, guardadas en el vector "x", se calculan los valores de "r" y "s":

```
> r=add(x[0],x[1]); s=neg(mul(x[0],x[1]))
  -1.0999999999999999
```

Y con los mismos se calcula el polinomio residual:

```
> b=divsin2(a,r,s).slice(0,n-2)
  [1, -2, 3]
```

Que puede ser resuelto con la ecuación cuadrática:

```
> cuad(b)
  [▼Complex                , ▼Complex                ]
      i: 1.4142135623730951      i: -1.4142135623730951
      r: 1                       r: 1
```

Por lo tanto, las cuatro soluciones del polinomio son:

```
x1 = 0.45+0.9473648i
x2 = 0.45-0.9473648i
x3 = 1+1.414214i
x4 = 1-1.414214i
```

5.8. MÉTODO QD (QUOTIENT DIVISOR)

Tanto el método de Newton, como el de Bairstow, requieren valores iniciales para comenzar el proceso de búsqueda y como dichos valores pueden ser complejos, no siempre es posible encontrarlos gráficamente. El método "QD", permite encontrar todas las soluciones (reales e imaginarias) sin necesidad de valores iniciales, sin embargo, tiene la desventaja de requerir un elevado número de iteraciones, razón por la cual este método se emplea casi exclusivamente para encontrar valores iniciales para métodos más eficientes como el de Newton y Bairstow.

Tomando en cuenta el polinomio general (5.1), en este método se calculan primero los vectores iniciales "p" y "d" con las siguientes expresiones:

$$p_0 = -\frac{a_1}{a_0};\ p_i = 0 \quad \{i = 1 \rightarrow n-1 \tag{5.22}$$

$$d_i = \frac{a_{i+2}}{a_{i+1}} \quad \{i = 0 \rightarrow n-2 \tag{5.23}$$

Donde "n" es el grado del polinomio, luego, con estos valores, se calculan los vectores "q" y "e" con las siguientes expresiones:

$$\begin{aligned} q_0 &= d_0 + p_0 \\ q_i &= d_i - d_{i-1} + p_i \quad \{i = 1 \rightarrow n-2 \\ q_{n-1} &= p_{n-1} - d_{n-2} \end{aligned} \tag{5.24}$$

$$e_i = \frac{q_{i+1}}{q_i} d_i \quad \begin{cases} i = 0 \rightarrow n-2 \end{cases} \tag{5.25}$$

Si los elementos de "e" son cercanos a cero, el proceso concluye, caso contrario se repite pero empleando ahora los vectores "q" y "e" como valores iniciales (es decir, haciendo que "p=q" y "d=e"). Cuando el proceso concluye las soluciones aproximadas se encuentran en el parámetro "q". No obstante, si existen soluciones complejas algunos de los valores de "e" no convergen hacia cero, en ese caso las soluciones complejas se encuentran resolviendo la ecuación cuadrática:

$$x^2 - r\,x - s = 0 \tag{5.26}$$

Donde "r" y "s" se calculan con:

$$\begin{aligned} r &= q_i + q_{i+1} \\ s &= -p_i q_{i+1} \end{aligned} \tag{5.27}$$

Siendo "i" el índice del elemento "e_i" que no converge hacia cero.

El proceso en sí es muy sencillo: una vez calculados los parámetros "p" y "d" (con las ecuaciones 5.22 y 5.23), se aplica repetidamente las ecuaciones (5.24) y (5.25) hasta que los valores de "e" (o algunos de ellos) sean cercanos a cero.

Para comprender mejor el proceso, se encontrarán las soluciones del siguiente polinomio aplicando manualmente el método:

$$P_4(x)=128x^4-256x^3+160x^2-32x+1=0$$

El vector y el grado del polinomio son (consola de Mozilla Firefox):

```
>>> a=[128,-256,160,-32,1]; n=a.length-1
4
```

Se calcula primero los parámetros "p" (ecuaciones 5.22):

```
>>> p=new Array(); p[0]=-a[1]/a[0]; for (i=1;i<n;i++) p[i]=0; p
[ 2, 0, 0, 0 ]
```

Y con la ecuación (5.23) los valores de "d":

```
>>> d=new Array(); for (i=0;i<n-1;i++) d[i]=a[i+2]/a[i+1]; d;
[ -0.625, -0.2, -0.03125 ]
```

Con estos valores se calculan los parámetros "q" (ecuaciones 5.24):

```
>>> q=new Array(); q[0]=d[0]+p[0]; for(i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i];
q[n-1]=p[n-1]-d[n-2]; q
[ 1.375, 0.425, 0.16875, 0.03125 ]
```

Y los parámetros "e" (ecuaciones 5.25):

```
>>> e=new Array(); for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.19318181818181818, -0.07941176470588236, -0.005787037037037037 ]
```

Como era de esperar, al ser la primera iteración, los valores de "e" no son aún cercanos a cero, por lo que se realiza un cambio de variables y se vuelven a calcular nuevos valores de "q" y "e" (recordando que en Javascript los vectores se copian por referencia, por lo que para hacer una verdadera copia se debe emplear el método "slice" o la función "copy"):

```
>>> p=q.slice(); d=e.slice();
[ -0.19318181818181818, -0.07941176470588236, -0.005787037037037037 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 1.1818181818181819, 0.5387700534759358, 0.24237472766884532, 0.037037037037037035 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.08806818181818181, -0.03572471172091666, -0.0008843112775697043 ]
```

Ahora los valores de "e" comienzan a acercarse a cero, repitiendo el proceso unas cuantas veces más se obtiene:

```
>>> p=q.slice(); d=e.slice();
[ -0.08806818181818181, -0.03572471172091666, -0.0008843112775697043 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 1.09375, 0.5911135235732009, 0.2772151281121923, 0.037921348314606744 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.04759615384615384, -0.016753855463532887, -0.0001209684197382038 7 ]
```

```
>>> p=q.slice(); d=e.slice();
[ -0.04759615384615384, -0.016753855463532887, -0.00012096841973820387 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 1.0461538461538462, 0.6219558219558219, 0.29384801515598696, 0.03804231673434494 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.028296703296703286, -0.00791549335238656, -0.00001566088148014558 5 ]

>>> p=q.slice(); d=e.slice();
[ -0.028296703296703286, -0.00791549335238656, -0.00001566088148014558 5 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 1.017857142857143, 0.6423370319001386, 0.3017478476268934, 0.038057977615825085 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.017857142857142846, -0.0037184265632671082, -0.00000197523025102880 5 ]

>>> p=q.slice(); d=e.slice();
[ -0.017857142857142846, -0.0037184265632671082, -0.00000197523025102880 5 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 1.0000000000000002, 0.6564757481940143, 0.30546429895990945, 0.03805995284607611 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.011722781217750246, -0.0017302186204243531, -2.461078773207667e-7 ]

>>> p=q.slice(); d=e.slice();
[ -0.011722781217750246, -0.0017302186204243531, -2.461078773207667e-7 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 0.9882772187822499, 0.6664683107913403, 0.3071942714724565, 0.038060198953953434 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.007905537077539262, -0.0007975071582296813, -3.0491827630983013e-8 ]

>>> p=q.slice(); d=e.slice();
[ -0.007905537077539262, -0.0007975071582296813, -3.0491827630983013e-8 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 0.9803716817047107, 0.6735763407106499, 0.3079917481388585 5, 0.038060229445781066 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.005431595827800699 5, -0.0003646589242687295, -3.768042367593548e-9 ]

>>> p=q.slice(); d=e.slice();
[ -0.005431595827800699 5, -0.0003646589242687295, -3.768042367593548e-9 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 0.97494008587691, 0.6786432776141819, 0.3083564032950849, 0.03806023321382343 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.0037808641255515766, -0.00016569075098226592, -4.6508705425824306e-10 ]
```

```
>>> p=q.slice(); d=e.slice();
[ -0.0037808641255515766, -0.00016569075098226592, -4.650870542582430 6e-10 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 0.9711592217513584, 0.6822584509887512, 0.3085220935809801, 0.03806023367891048 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.002656131398357027, -0.00007492652865782728, -5.737456841630906e-11 ]

>>> p=q.slice(); d=e.slice();
[ -0.002656131398357027, -0.00007492652865782728, -5.737456841630906e-11 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 0.9685030903530013, 0.6848396558584503, 0.3085970200522633 6, 0.03806023373628505 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.0018781810103493325, -0.00003376279873524895, -7.07618461148256e-12 ]

>>> p=q.slice(); d=e.slice();
[ -0.0018781810103493325, -0.00003376279873524895, -7.07618461148256e-12 ]
>>> q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i]; q[n-1]=p[n-1]-d[n-2];
q
[ 0.966624909342652, 0.6866840740700644, 0.3086307828439224, 0.03806023374336124 ]
>>> for(i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i]; e;
[ -0.0013342476234186583, -0.00001517472065851125 2, -8.726324634325295e-13 ]
```

Como se puede ver, el último valor de "e" es casi cero, tiene 12 ceros después del punto, mientras que el primero sólo tiene dos. Este es el comportamiento normal del método, los últimos valores de "e" convergen rápidamente pero los primeros requieren decenas, cientos e inclusive miles de iteraciones para acercarse a 0 y esta es la razón por la cual el método se emplea en la práctica sólo para obtener valores iniciales aproximados.

Deteniendo el proceso en este punto las soluciones (raíces) aproximadas del polinomio son los elementos del vector "q", es decir:

```
[ 0.966624909342652, 0.6866840740700644, 0.3086307828439224, 0.03806023374336124 ]
```

En concordancia con los valores de "e" (del error) los primeros valores de "q" son menos aproximados que los últimos, sin embargo, aún son buenos valores iniciales. Por ejemplo empleando estos valores con el método de Newton se pueden calcular resultados más precisos:

```
>>> x=[0.966624909342652, 0.6866840740700644, 0.3086307828439224,
0.03806023374336124]
[ 0.966624909342652, 0.6866840740700644, 0.3086307828439224, 0.03806023374336124 ]
>>> newtonp(a,x[0])
0.9619397662556441
>>> newtonp(a,x[1])
0.6913417161825444
>>> newtonp(a,x[2])
0.30865828381745514
>>> newtonp(a,x[3])
0.038060233744356624
```

Y de la misma forma con el método de Bairstow:

```
>>> bairstow(a,x[0],x[1])
[ 0.691341716182545, 0.9619397662556433 ]
>>> bairstow(a,x[2],x[3])
[ 0.03806023374435663, 0.3086582838174551 ]
```

Como otro ejemplo se encontrarán las soluciones del siguiente polinomio:

$$P_4(x)=x^4-6x^3+12x^2-19x+12=0$$

El vector de coeficientes y el orden son:

```
>>> a=[1,-6,12,-19,12]; n=a.length-1;
4
```

Los valores iniciales de "p" y "d" se calculan, pero no se muestran:

```
>>> p=[]; p[0]=-a[1]/a[0]; for (i=1;i<n;i++) p[i]=0; d=[]; for (i=0;i<n-1;i++)
d[i]=a[i+2]/a[i+1];
-0.631578947368421
```

Con estos valores se calculan los valores de "q" y "e", pero sólo se muestran los valores de "e" pues son los que tienen que aproximarse a 0:

```
>>> q=[]; q[0]=d[0]+p[0]; for (i=1;i<n-1;i++) q[i]=d...[]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ -0.20833333333333337, -3.616666666666666, -0.41911229687121027 ]
```

Ahora, se repite el proceso unas cuantas veces (la consola de Firefox sólo muestra tres puntos cuando la instrucción es muy larga):

```
>>> p=q.slice(); d=e.slice(); q[0]=d[0]+p[0]; for (i...2]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ 0.16437728937728932, 5.016155988857938, -0.10612794723082625 ]

>>> p=q.slice(); d=e.slice(); q[0]=d[0]+p[0]; for (i...2]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ 0.0772893772893773, -2.623817916188112, 0.1261808611259194 ]

>>> p=q.slice(); d=e.slice(); q[0]=d[0]+p[0]; for (i...2]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ -0.01611570247933882, 5.544129163797414, 0.07318238894117786 ]

>>> p=q.slice(); d=e.slice(); q[0]=d[0]+p[0]; for (i...2]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ -0.01893201453306186, -4.339585512684148, -0.01896869797336019 ]

>>> p=q.slice(); d=e.slice(); q[0]=d[0]+p[0]; for (i...2]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ -0.0018873657420800404, -6.8229231957093095, -0.029554319669608076 ]

>>> p=q.slice(); d=e.slice(); q[0]=d[0]+p[0]; for (i...2]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ 0.0030331042986204864, 7.882754657134322, -0.004006842757045116 ]
```

```
>>> p=q.slice(); d=e.slice(); q[0]=d[0]+p[0]; for (i...2]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ 0.0011051790596397887, -2.5244764168300495, 0.008671230080417355 ]

>>> p=q.slice(); d=e.slice(); q[0]=d[0]+p[0]; for (i...2]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ -0.0002951260462936007, 4.883180804383195, 0.004201715340321471 ]

>>> p=q.slice(); d=e.slice(); q[0]=d[0]+p[0]; for (i...2]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ -0.00028147247694370264, -3.599858567423578, -0.0014896138869402225 ]

>>> p=q.slice(); d=e.slice(); q[0]=d[0]+p[0]; for (i...2]; for (i=0;i<n-1;i++)
e[i]=q[i+1]/q[i]*d[i]; e
[ -0.00001517145301904898 7, -13.121149110924334, -0.0018929233692072359 ]
```

Como se puede ver en ese caso dos de los valores de "e" (el primero y el tercero) convergen lentamente hacia cero, mientras que uno (el segundo) no lo hace. Como se dijo, cuando esto sucede se sabe que existen soluciones complejas, en este caso son la segunda y tercera (porque es el segundo elemento el que no converge hacia cero).

Entonces las soluciones reales son la primera y cuarta, cuyos valores aproximados son los elementos respectivos del vector "q":

```
>>> q
[ 3.9999599354138753, 0.21559906992379752, 0.7858385243927515, 0.9986024702695752 ]
```

Las dos soluciones complejas se calculan aplicando las ecuaciones (5.26) y (5.27) (conjuntamente "cuad"), donde el valor de "i" es 1, porque es el índice del elemento que no converge hacia cero (el segundo elemento):

```
>>> i=1; r=q[i]+q[i+1]; s=-p[i]*q[i+1]; x=cuad([1,-r,-s])
[ Complex { r=0.5007187971582745, i=1.6575261970395967, toString=function(), más... },
Complex { r=0.5007187971582745, i=-1.6575261970395967, toString=function(), más... } ]
```

Al igual que en el anterior ejemplo, los valores calculados con QD pueden ser empleados como valores iniciales del método de Newton:

```
>>> newtonp(a,q[0])
4
>>> newtonp(a,q[3])
0.9999999999999999
>>> r=newtonp(a,x[0])
Complex { r=0.49999999999856215, i=1.6583123951773877, toString=function(), más... }
```

No es necesario calcular el otro resultado complejo porque las soluciones complejas siempre vienen en pares conjugados, es decir:

```
>>> r.conj()
Complex { r=0.49999999999856215, i=-1.6583123951773877, toString=function(), más... }
```

Empleando los resultados de QD con el método de Bairstow se obtiene:

```
>>> bairstow(a,q[0],q[3])
[ 0.9999999999999571, 4.000000000000018 ]
>>> bairstow(a,x[0],x[1])
[ Complex { r=0.5000000000000296, i=1.6583123951750793, toString=function(), más... },
Complex { r=0.5000000000000296, i=-1.6583123951750793, toString=function(), más... } ]
```

5.8.1. Algoritmo y código

Como se vio en los ejemplos y ejercicios del anterior acápite, la aplicación manual del método QD es bastante sencilla, sin embargo, su automatización no lo es tanto, porque es necesario considerar varios aspectos.

En primer lugar se deben evitar las divisiones entre cero y algunas situaciones en las cuales se presentan dichas divisiones son: a) Uno de los coeficientes es cero; b) Tres o más coeficientes consecutivos son iguales y el grado del polinomio es menor a 5; c) Tres o más coeficientes están igualmente espaciados y d) El espaciamiento de dos coeficientes es el cuadrado de los dos coeficientes anteriores.

En segundo lugar, y como se ha visto en los ejemplos, la convergencia hacia cero depende de que las soluciones sean reales o complejas, además las últimas raíces convergen más rápidamente que las primeras por lo que deben tomarse en cuenta estos factores para determinar si se ha encontrado o no la solución con el error permitido.

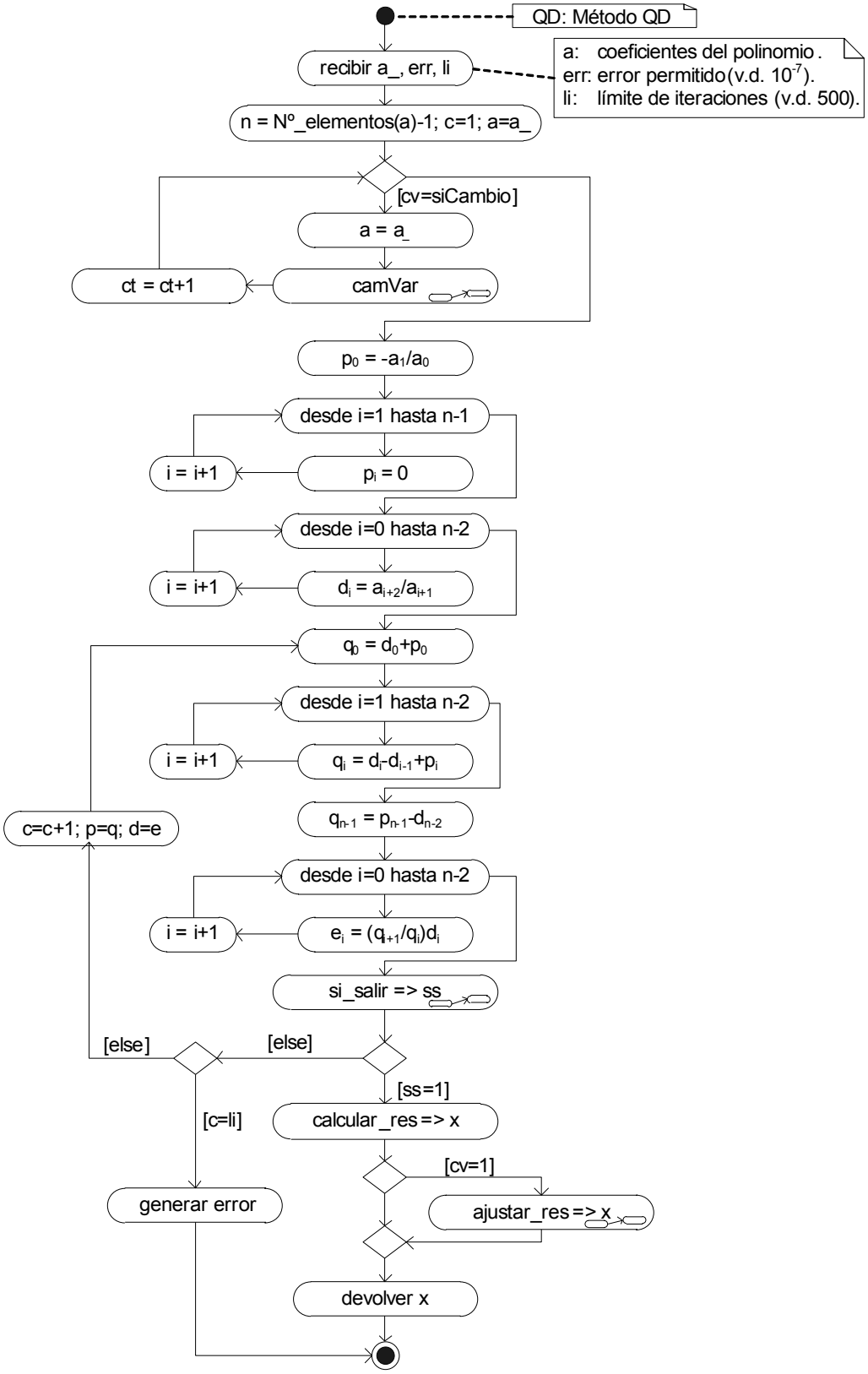
Para resolver el primer problema se realiza un cambio de variable. Por comodidad, el cambio de variable más utilizado es y=x-1. Por supuesto, cuando se realiza un cambio de variable, a las soluciones calculadas se les debe sumar la constante que se restó (normalmente 1) para obtener las soluciones correctas.

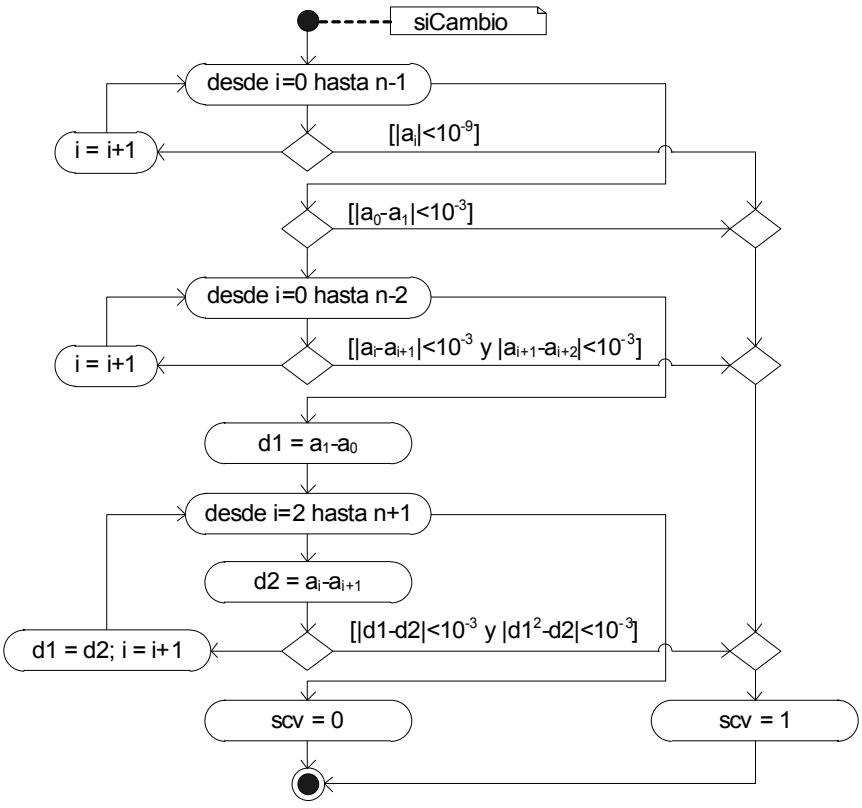
El algoritmo general, donde se toman en cuenta los aspectos antes mencionados, se presenta en la siguiente página y las subrutinas que forman parte del mismo se muestran en las subsiguientes.

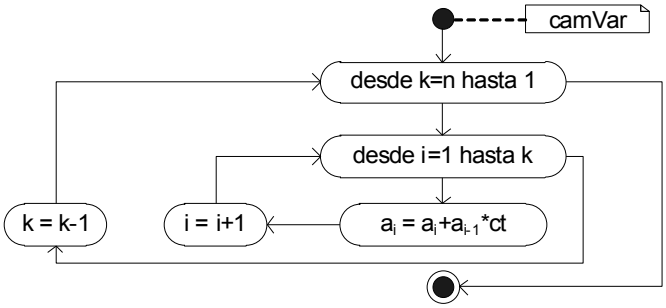
Como se puede ver en el algoritmo, cuando es necesario un cambio de variable, se prueba primero con 1 (ct=1), pero si con dicho cambio continua el problema, se prueba con 2, 3 y así hasta eliminar el problema.

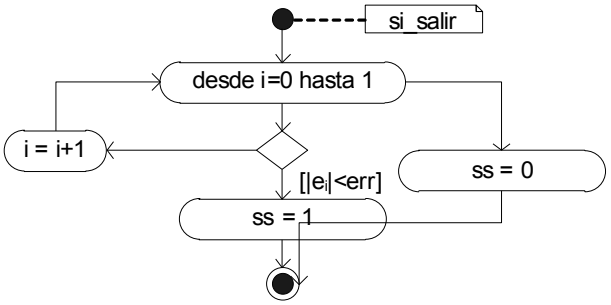
El código respectivo, añadido a la unidad "mat205.js", es:

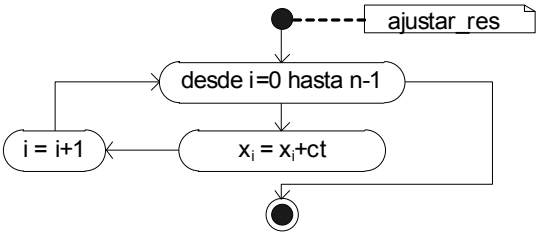
```
function qd(a_,err,li) {
  //Sub-función que determina si es necesario (true) o no (false)
  //un cambio de variable
  function siCambio() {
    for (i=0;i<n;i++) if (abs(a[i])<1e-9) return true;
    if (abs(a[0]-a[1])<1e-3) return true;
    for (i==0;i<n-1;i++)
      if (abs(a[i]-a[i+1])<1e-3 && abs(a[i+1]-a[i+2])<1e-3) return true;
    var d1=a[1]-a[0];
    for (i=2;i<=n;i++) {
      d2=a[i]-a[i-1];
      if (abs(d1-d2)<1e-3 || abs(d1*d1-d2)<1e-3) return true;
      d1=d2;}
    return false;}
```

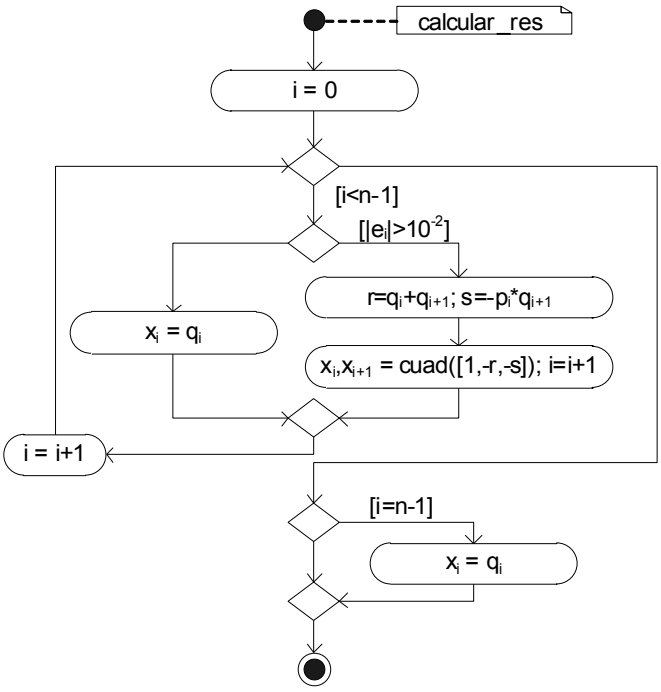

QD: Método QD
recibir a_, err, li
a: coeficientes del polinomio.
err: error permitido(v.d. 10^-7).
li: límite de iteraciones (v.d. 500).
n = Nº_elementos(a)-1; c=1; a=a_
[cv=siCambio]
a = a_
ct = ct+1
camVar
p0 = -a1/a0
desde i=1 hasta n-1
i = i+1
pi = 0
desde i=0 hasta n-2
i = i+1
di = ai+2/ai+1
q0 = d0+p0
desde i=1 hasta n-2
i = i+1
qi = di-di-1+pi
qn-1 = pn-1-dn-2
c=c+1; p=q; d=e
desde i=0 hasta n-2
i = i+1
ei = (qi+1/qi)di
si_salir => ss
[else]
[else]
[ss=1]
[c=li]
calcular_res => x
[cv=1]
generar error
ajustar_res => x
devolver x


siCambio
desde i=0 hasta n-1
[|a_i|<10^-9]
i = i+1
[|a_0-a_1|<10^-3]
desde i=0 hasta n-2
[|a_i-a_i+1|<10^-3 y |a_i+1-a_i+2|<10^-3]
i = i+1
d1 = a_1-a_0
desde i=2 hasta n+1
d2 = a_i-a_i+1
[|d1-d2|<10^-3 y |d1^2-d2|<10^-3]
d1 = d2; i = i+1
scv = 0
scv = 1


camVar
desde k=n hasta 1
desde i=1 hasta k
k = k-1
i = i+1
a_i = a_i+a_i-1*ct


si_salir
desde i=0 hasta 1
i = i+1
[|e_i|<err]
ss = 0
ss = 1


ajustar_res
desde i=0 hasta n-1
i = i+1
x_i = x_i+ct




```
//Sub-función que realiza el cambio de variable restando "ct"
//a la variable actual
function camVar() {
  for (k=n;k>0;k--)
    for (i=1;i<=k;i++) a[i]=a[i]+a[i-1]*ct;}
//Variables locales
var cv,b,i,k,p,q,d,e,x,r,s,n,vx,a=a_.slice(),n=a.length-1,c=1,ct=1;
//Valores por defecto
if (err==null) err=1e-9;
if (li==null) li=500;
//Se realiza un cambio de variable si es necesario
while (cv=siCambio()){a=a_.slice(); camVar(); ct++;}
//Valores iniciales de "p" y "d"
p=new Array(n); p[0]=-a[1]/a[0];
for (i=1;i<n;i++) p[i]=0;
d=new Array(n-1); q=new Array(n); e=new Array(n-1);
for (i=0;i<n-1;i++) d[i]=a[i+2]/a[i+1];
//Cálculo iterativo de "q" y "e"
while (true) {
  q[0]=d[0]+p[0];
  for (i=1;i<n-1;i++) q[i]=d[i]-d[i-1]+p[i];
  q[n-1]=p[n-1]-d[n-2];
  for (i=0;i<n-1;i++) e[i]=q[i+1]/q[i]*d[i];
  if (abs(e[0])<err || abs(e[1])<err) break;
  if (c++ == li) throw new Error("QD no converge.");
  p=q.slice(); d=e.slice();}
//Cálculo de los resultados
x=new Array(n);
for (i=0;i<n-1;i++) {
```

```
    if (abs(e[i])>1e-2) {//resultados complejoas
      r=-q[i]-q[i+1]; s=p[i]*q[i+1];
      vx=cuad([1,r,s]);
      x[i]=vx[0];x[i+1]=vx[1];i++;
    } else x[i]=q[i];//resultado real
  }
  if (i==n-1) x[i]=q[i];//si queda un resultado es real
  //Ajuste de resultados en caso de cambio de variable
  if (cv) {
    for (i=0;i<n;i++) x[i]=add(x[i],ct);
    a=b.slice();}//Se reponen los valores originales de "a"
  return x;
}
```

Haciendo correr el programa para los polinomios de los ejemplos manuales se obtiene:

```
>>> a=[128,-256,160,-32,1]
[ 128, -256, 160, -32, 1 ]
>>> precision(qd(a),7)
[ 0.9623967, 0.6908849, 0.3086582, 0.03806023 ]
>>> a=[1,-6,12,-19,12]
[ 1, -6, 12, -19, 12 ]
>>> precision(qd(a),7)
[ 3.999999, Complex { r=0.5000405, i=1.658263, toString=function(), más... }, Complex {
r=0.5000405, i=-1.658263, toString=function(), más... }, 0.9999198 ]
```

Que son los resultados obtenidos en los ejemplos.

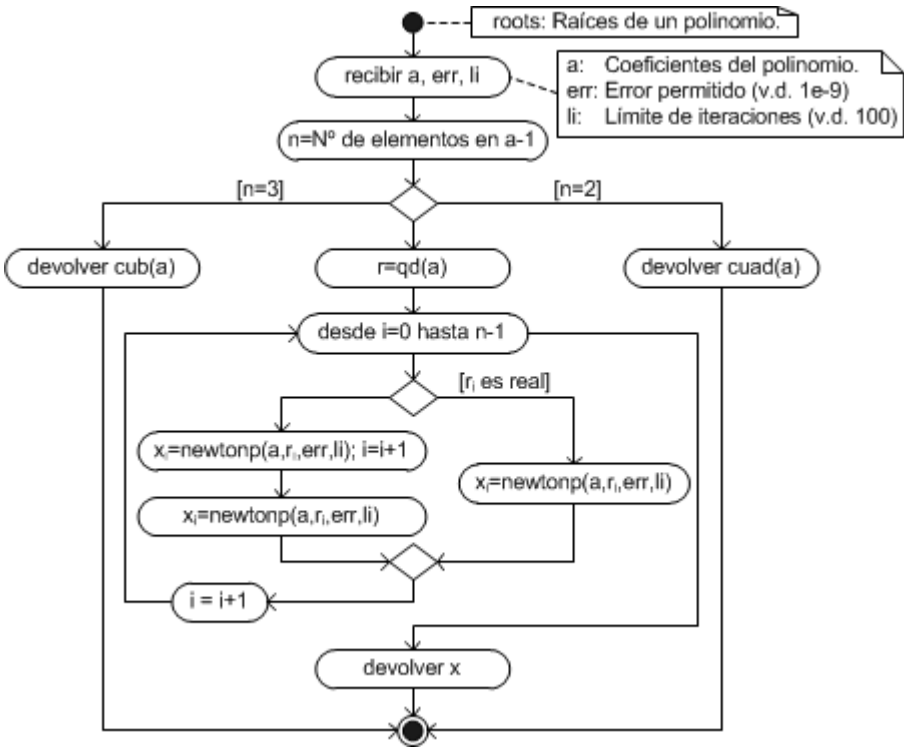
5.9. PROGRAMA GENERAL PARA RESOLVER ECUACIONES POLINOMIALES

Con los métodos elaborados hasta ahora se puede crear un programa general que encuentre todas las raíces (reales y complejas) de un polinomio.

Dicho programa se denominará "roots" y el mismo emplea el programa "cuad" si la ecuación es de segundo grado, "cub" si la ecuación es cúbica y "newtonp", con valores iniciales calculados con "qd" en cualquier otro caso (esto porque el método de Newton es más estable que el método de "bairstow").

El algoritmo de dicho programa se presenta en la siguiente página y el código respectivo, añadido a la librería "mat205.js", es:

```
function roots(a,err,li) {
  var n=a.length-1,v,r,x=[];
  if (n==2) return cuad(a);
  if (n==3) return cub(a);
  r=qd(a);
  for (var i=0;i<n;i++)
    if (!isNaN(r[i]))
      x[i]=newtonp(a,r[i],err,li);
    else {
      x[i]=newtonp(a,r[i],err,li);
```




```
            x[i+1]=x[i].conj();i++;}
    return x;
}
```

Haciendo correr el programa con las ecuaciones polinomiales del ejemplo manual y mostrando los resultados con 9 dígitos de precisión (que es el error por defecto), se obtiene:

```
>>a=[128,-256,160,-32,1]
[128, -256, 160, -32, 1]
>>precision(roots(a),9)
[0.961939766, 0.691341716, 0.308658284, 0.0380602337]
>>a=[1,-6,12,-19,12]
[1, -6, 12, -19, 12]
>>precision(roots(a),9)
[4, 0.5+1.6583124i, 0.5-1.6583124i, 1]
```

5.10. EJEMPLOS

1, Encuentre las soluciones de las siguientes ecuaciones (soluciones reales y complejas) empleando el método más adecuado para cada caso.

$$x^2 - x + 3 = 0$$
$$x^3 - 15x^2 + 71x - 105 = 0$$
$$x^4 - 4x^2 + x - 5 = 0$$

La primera ecuación se resuelve con "cuad", la segunda con "cub" y la tercera con "roots" (por supuesto todas pueden ser resueltas también con roots). El problema puede ser resuelto fácilmente en la calculadora, pero en este caso y simplemente para recordar, se resuelve el problema creando una página HTML:

```
<html>
<head>
  <title>Ejemplo 1</title>
  <script src="mat205.js"></script>
  <script>
    document.write("<h3>Soluciones de x^2-x+3=0:</h3>");
    document.write(cuad([1,-1,3]));
    document.write("<h3>Soluciones de x^3-15*x^2+71*x-105=0</h3>");
    document.write(cub([1,-15,71,-105]));
    document.write("<h3>Soluciones de x^4-4*x^2+x-5</h3>");
    document.write(precision(roots([1,0,-4,1,-5],1e-12),9));
  </script>
</head>
<body>

</body>
</html>
```

Donde, como se puede ver, los resultados de "roots" se muestran con 9 dígitos de precisión (que es la precisión por defecto del método). Abriendo la página en un navegador se obtienen los siguientes resultados:

Soluciones de x^2-x+3=0:

[0.5+1.6583123951777i, 0.5-1.6583123951777i]

Soluciones de x^3-15*x^2+71*x-105=0

[3, 5, 7]

Soluciones de x^4-4*x^2+x-5

[-2.31601568, 0.0835924361+0.998843786i, 0.0835924361-0.998843786i, 2.1488308]

2, Encuentre las raíces (los valores de "z") del siguiente sistema empleando el método de Newton Raphson para resolver la ecuación no lineal y el método más adecuado para la ecuación polinomial. Estime el valor inicial de "x" con el método incremental (se sabe que el valor de "x" es positivo y no mayor a 15).

$$f(x)=\frac{x^{2.1}-3.1\,x-10}{x^{0.7}-2\,x^{2}-7}=0$$

$$p(z)=10.8\,z^{3}+38.97\,z^{2}-55.13\,z-x=0$$

Para resolver este problema, primero se calcula el valor de "x" resolviendo la primera ecuación con el método de Newton Raphson (obteniendo el valor inicial con le método incremental) luego se resuelve la ecuación polinomial y como es de tercer grado, se emplea para ello "cub". Una vez más, aunque el problema puede ser resuelto en la calculadora, se resuelve el mismo en una página HTML:

```
<html>
<head>
  <title>Ejemplo 2</title>
  <script src="mat205.js"></script>
  <script>
    //Función a resolver con el método de Newton-Raphson
    function f(x){
      return (pow(x,2.1)-3.1*x-10)/(pow(x,0.7)-2*sqr(x)-7);
    }
    //Cálculo del valor inicial con el método incremental
    var x1=incr1(f,0.1,1);
    //Cálculo del valor de "x" con el método de Newton-Raphson
    var x=precision(newton(f,x1),15);
    document.write("<b>Valor de x:</b> "+x+"<br>");
    //Resolución de la ecuación polinomial
    var a=[10.8,38.97,-55.13,x];
    var z=cub(a);
    document.write("<b>Valores de z:</b> "+z+"<br>");
  </script>
</head>
<body>

</body>
</html>
```

Note que sólo se guardan 15 dígitos de "x", pues esa es la precisión por defecto del método de Newton Raphson. Observe también que para mostrar el texto en negrita se ha empleado la etiqueta "<b>" (bold=negrita) y para insertar un salto de línea se ha empleado la etiqueta "
" (break line).

Abriendo la página en un navegador se obtienen los siguientes resultados:

Valor de x: 4.55195912900217
Valores de z: [-4.71090290346701, 0.0882012562529995, 1.01436831388068]

3, Encuentre la solución del siguiente sistema de ecuaciones (el valor de "x") empleando el método de la Secante para resolver la ecuación no lineal y el método más adecuado para la ecuación polinomial (r_0 a r_3 son las raíces ordenadas de la ecuación polinomial). Estime el valor inicial para la ecuación no lineal de la gráfica de la función.

$$f(x)=r_0 x^{2.1}+ r_1 * x^{0.3}-r_2 x-r_3=0$$
$$p(r)=r^4-10.8r^3+ 38.97r^2-55.13r+ 26.4=0$$

En este caso, primero se encuentran las raíces del polinomio con "roots" (pues el grado del polinomio es mayor a 3), los resultados se ordenan con el método "sort" y se emplean para resolver la ecuación no lineal con el método de la Secante.

Como el valor inicial para el método de la Secante proviene de la gráfica y aún no se han estudiado los elementos HTML que permiten leer datos desde el teclado, se resolverá el problema en la calculadora, encontrando primero los valores de "r":

```
>>a=[1,-10.8,38.97,-55.13,26.4]
[1, -10.8, 38.97, -55.13, 26.4]
>>r=precision(roots(a),9)
[5, 3.2, 1.5, 1.1]
```

Entonces se ordena estos resultados de forma ascendente con "sort":

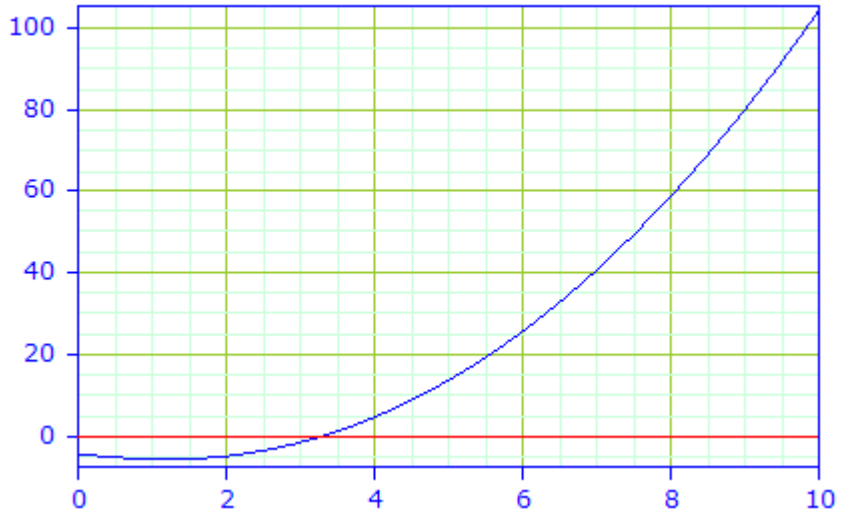
```
>>r.sort(function(a,b){return a-b;})
[1.1, 1.5, 3.2, 5]
```

Con estos resultados se puede programar la ecuación no lineal:

```
>>f=function(x){ return r[0]*pow(x,2.1)+r[1]*pow(x,0.3)-r[2]*x-r[3]; }
function (x){
  return r[0]*pow(x,2.1)+r[1]*pow(x,0.3)-r[2]*x-r[3];
}
```

Entonces se elabora la gráfica de la función para encontrar él o los valores iniciales para el método de la Secante:

```
>>plot([f,function(x){return 0;}],0,10)
```



Como se puede ver, la recta "y=0" ha sido programada directamente como una función literal. De esta gráfica se obtiene un valor inicial igual a 3.2. Se puede emplear directamente ese valor para calcular la solución pedida:

```
>>precision(secante(f,3.2),15)
3.28487331638493
```

O también se puede emplear dos valores iniciales cercanos a la solución, por ejemplo 3.2 y 3.3:

```
>>precision(secante(f,3.2,3.3),15)
3.28487331638493
```

O dos valores ubicados a ambos lados de la solución, por ejemplo 2 y 4:

```
>>precision(secante(f,2,4),15)
3.28487331638493
```

Obteniéndose en todos los casos el mismo resultado.

4 El siguiente polinomio tiene dos soluciones reales entre 1 y 4. Encuentre dichas soluciones con los métodos de Newton y Bairstow, empleando valores iniciales obtenidos de la gráfica de la función.

$$p_5(x)=x^5-5.9x^4-0.9x^3+34x^2-0.4x-38.2=0$$

Una vez más, como los valores iniciales provienen de la gráfica, se resuelve el problema en la calculadora y para ello primero se crea un vector con los coeficientes del polinomio:

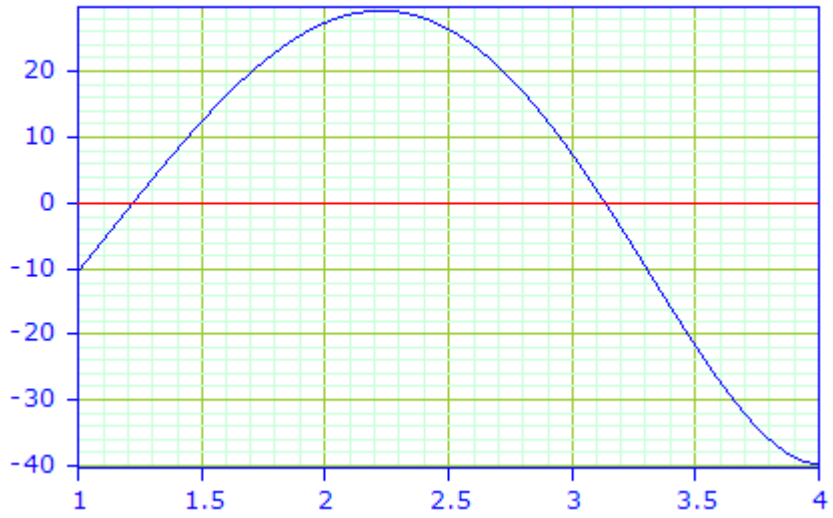
```
>>a=[1,-5.9,-0.9,34,-0.4,-38.2]
[1, -5.9, -0.9, 34, -0.4, -38.2]
```

El valor del polinomio se calcula por división sintética ("divsin1"), recordando que el último elemento de la división (el residuo) es el valor de la función. En este caso, al ser un polinomio de quinto grado, el vector tiene 6 elementos, por lo tanto el último elemento de la división es el sexto (el elemento número 5, recuerde que el primer elemento es el número 0):

```
>>f=function(x){return divsin1(a,x)[5];}
function (x){return divsin1(a,x)[5];}
```

Graficando esta función y la recta "y=0" se obtiene los valores iniciales para las soluciones pedidas:

```
>>plot([f,function(x){return 0;}],1,4)
```



Por lo tanto los valores iniciales (las soluciones aproximadas) son 1.2 y 3.1. Empleando estos valores con el método de Newton se obtiene:

```
>>precision(newtonp(a,1.2),9)
1.2218057
>>precision(newtonp(a,3.1),9)
3.13315034
```

Y con el método de bairstow:

```
>>precision(bairstow(a,1.2,3.1),9)
[1.2218057, 3.13315034]
```

5 Encuentre el valor de "z" resolviendo la siguiente ecuación no lineal por el método de regula falsi. En esta ecuación x_0 es la solución real del polinomio de quinto grado, y_0 y y_1 son las soluciones reales del polinomio de cuarto grado. La solución real del polinomio de quinto grado debe ser encontrada con el método de Newton (valor inicial calculado con QD), las soluciones reales de la ecuación de cuarto grado deben ser encontradas con el método de Bairstow (valores iniciales calculados con QD), el segmento de solución de la ecuación no lineal debe ser encontrado con el método incremental (se sabe que la solución es positiva).

$$f(z)=x_0 z^{3.1}-y_0 z^{0.2}+y_1 z^{0.75}-370.1=0$$
$$p_5(x)=x^5-9.5x^4+47x^3-137x^2+226x-227.5=0$$
$$p_4(y)=y^4-7.4y^3+23.16y^2-41.414y+33.9171=0$$

Si bien en este caso los valores iniciales no provienen de la gráfica, deben ser seleccionados de los resultados devueltos por "qd", por lo que es más sencillo resolver el problema en la calculadora.

Se comienza resolviendo la ecuación de quinto grado, para lo cual se crea el vector de coeficientes:

```
>>a=[1,-9.5,47,-137,226,-227.5]
[1, -9.5, 47, -137, 226, -227.5]
```

Y se encuentran los valores iniciales con QD:

```
>>r=precision(qd(a),9)
err: QD no converge.
```

Como se puede ver, en este caso el método QD no converge con el error por defecto (9 dígitos), por lo que se disminuye el mismo:

```
>>r=precision(qd(a,1e-7),7)
[2.000003+3.000001i, 2.000003-3.000001i, 3.499995, 1+2i, 1-2i]
```

Tal como se indica en el enunciado, una de estas soluciones es real y como se puede ver es la tercera. Empleando esta solución como valor inicial del método de Newton se obtiene la solución real del primer polinomio:

```
>>x0=precision(newtonp(a,r[2]),9)
3.5
```

Ahora se resuelve la ecuación de cuarto grado, siendo el vector de coeficientes:

```
>>b=[1,-7.4,23.16,-41.414,33.9171]
[1, -7.4, 23.16, -41.414, 33.9171]
```

Se encuentran los valores iniciales con el método QD:

```
>>r=precision(qd(b),9)
[3.1, 1.09908697+2.00173798i, 1.09908697-2.00173798i, 2.10182605]
```

Como se puede ver, en este caso las soluciones reales son la primera y cuarta, empleando esta soluciones como valores iniciales del método de Bairstow se obtienen las soluciones reales del segundo polinomio:

```
>>y=precision(bairstow(b,r[0],r[3]),9)
[2.1, 3.1]
```

Una vez encontradas las soluciones reales de los dos polinomios se puede programar la ecuación no lineal:

```
>>f=function(z){ return x0*pow(z,3.1)-y[0]*pow(z,0.2)-
y[1]*pow(z,0.75)-370.1; }
function (z){
  return x0*pow(z,3.1)-y[0]*pow(z,0.2)-y[1]*pow(z,0.75)-370.1;
}
```

Y para resolver la misma se encuentra el segmento de solución con el método incremental, siendo el valor inicial del segmento:

```
>>z1=incr1(f,0.1,1)
4.1
```

Y como el incremento empleado es 1, el valor final del segmento es:

```
>>z2=z1+1
5.1
```

Finamente, con estos valores y el método de Regula Falsi se encuentran la solución del sistema (el valor de z):

```
>>z=precision(refa(f,z1,z2),12)
4.54606433266
```

Sólo para demostrar que es posible, se ha resuelto el problema en una página HTML:

```
<html>
<head>
  <title>Ejemplo 5</title>
  <script src="mat205.js"></script>
  <script>
    //Cálculo de la solución real de la ecuación de quinto grado
    var a=[1,-9.5,47,-137,226,-227.5];
    var r=qd(a,1e-7);
    //Búsqueda del valor real
    for (var i=0;i<r.length;i++)
      if (isReal(r[i])) break;
    x0=precision(newtonp(a,r[i]),9);
    document.write("<b>Solución de la ecuación de quinto "+
      "grado: </b>"+x0+"<br>");
    //Cálculo de las soluciones reales de la ecuación de cuarto grado
    var b=[1,-7.4,23.16,-41.414,33.9171];
    r=qd(b);
    //Búsqueda del primer valor real
    for (var i=0;i<r.length;i++)
      if (isReal(r[i])) break;
    var y0=r[i];
    //Búsqueda del segundo valor real
    for (i++;i<r.length;i++)
      if (isReal(r[i])) break;
    var y1=r[i];
    y=precision(bairstow(b,y0,y1),9);
    document.write("<b>Soluciones de la ecuación de cuarto "+
      "grado: </b>"+y+"<br>");
    //Resolución de la ecuación no lineal
    function f(z){
      return x0*pow(z,3.1)-y[0]*pow(z,0.2)-y[1]*pow(z,0.75)-370.1;
    }
    var x1=incr1(f,1);
    var z=precision(refa(f,x1,x1+1),12);
    document.write("<b>Valor de z: </b>"+z);
  </script>
</head>
<body>
```

```
</body>
</html>
```

Al abrir la página en un navegador se obtienen (como era de esperar) los mismos resultados que en la calculadora:

Solución de la ecuación de quinto grado: 3.5
Soluciones de la ecuación de cuarto grado: [2.1, 3.1]
Valor de z: 4.54606433266

5.11. EJERCICIOS

Como en el anterior tema para presentar los ejercicios que resuelva en la calculadora, copie las instrucciones en un editor de texto y luego, al momento de presentar el trabajo.

1. Encuentre las soluciones de las siguientes ecuaciones polinomiales (soluciones reales y complejas) empleando el método más adecuado para cada caso.

$$3x^2-5x+3=0$$
$$x^2-15x+50=0$$
$$x^3-19x^2+115x-225=0$$
$$x^3+3x+4=0$$
$$x^8-60x^7+1554x^6-22680x^5+203889x^4-1155420x^3+4028156x^2-7893840x+6652800=0$$

2. Encuentre las raíces (los valores de "y") del siguiente sistema. Resuelva las ecuaciones polinomiales con el método más adecuado ("r_1" y "r_2" son las raíces de la primera ecuación de segundo grado), "w" es la solución de la ecuación no lineal y para resolverla se debe emplear el método de la Bisección, encontrando el segmento de solución con el método incremental (se sabe que el valor de "w" es mayor a 0.3 y menor a 20).

$$5\ln(w)-3.2w^{2.1}+7.5=0$$
$$r^2-r-6=0$$
$$r_1y^2+r_2y+w=0$$

3. Encuentre las soluciones del siguiente sistema (los valores de "z") empleando el método de Regula Falsi para resolver la ecuación no lineal y el método más adecuado para la ecuación polinomial (x_0 a x_3 son las raíces de la ecuación polinomial). Encuentre los segmentos de solución (para el método de Regula Falsi) graficando la función.

$$f(z)=e^{x_0z+x_1}-x_2z^{2.45}+x_3z-3.9=0$$
$$p(x)=x^4+3.69x^3-63.2149x^2-78.6403x+200.261=0$$

4. El siguiente polinomio tiene dos soluciones reales entre 1 y 5. Encuentre dichas soluciones con el método de Newton y con el método de Bairstow. Los valores iniciales deben ser calculados con el método QD.

$$x^4-10.43x^3+47.86x^2-120.14x+118.78=0$$

5. Encuentre los valores de "z" resolviendo la siguiente ecuación cúbica ($p_3(z)$). En esta ecuación y_0, y_1, y_2, y_3, y_4, son las soluciones (raíces) del polinomio de quinto grado ($p_5(y)$). En la ecuación de quinto grado "x" es la primera solución positiva de la ecuación no lineal (f(x)). Esta ecuación debe ser resuelta con el método de la secante y el segmento de solución debe ser encontrando de graficándola. Las ecuaciones polinomiales deben ser resueltas empleando los método más adecuados para cada caso.

$$p_3(z) = z^3 + y_1 y_2 z^2 - y_3 y_4 z + y_0 = 0$$
$$f(x) = x^{2.2} - 5.1\, x^{1.1} - 8x + 35 = 0$$
$$p_5(y) = y^5 + 2y^4 - 3y^3 + y^2 - 2y + x = 0$$

6. ECUACIONES LINEALES

Con frecuencia en el campo de la ingeniería es necesario resolver sistemas de ecuaciones lineales que tienen entre 2 y cientos de miles de ecuaciones lineales.

Los métodos que permiten resolver sistemas de ecuaciones lineales pueden dividirse en dos grupos: a) *No iterativos o directos*, con los cuales se pueden resolver sistemas con algunos cientos de ecuaciones lineales (no por las limitaciones de los métodos, sino por errores de redondeo) y b) *Los métodos iterativos*, que permiten resolver sistemas con hasta cientos de miles de ecuaciones lineales, aunque, como normalmente ocurre con los métodos iterativos, la convergencia no está garantizada.

6.1. MÉTODO DE ELIMINACIÓN DE GAUSS

El método de eliminación de Gauss, reduce el sistema de ecuaciones lineales a un sistema triangular superior. Así al aplicar el método al siguiente sistema de 4 ecuaciones lineales:

$$\begin{aligned} a_{11}x_1 &+ a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = c_1 \\ a_{21}x_1 &+ a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = c_2 \\ a_{31}x_1 &+ a_{32}x_2 + a_{33}x_3 + a_{34}x_4 = c_3 \\ a_{41}x_1 &+ a_{42}x_2 + a_{43}x_3 + a_{44}x_4 = c_4 \end{aligned} \qquad (6.1)$$

Se reduce a:

$$\begin{aligned} x_1 &+ a'_{12}x_2 + a'_{13}x_3 + a'_{14}x_4 = c'_1 \\ 0 &+ x_2 + a'_{23}x_3 + a'_{24}x_4 = c'_2 \\ 0 &+ 0 + x_3 + a'_{34}x_4 = c'_3 \\ 0 &+ 0 + 0 + x_4 = c'_4 \end{aligned} \qquad (6.2)$$

Donde se han escrito apostrofes simplemente para hacer notar que son valore diferentes a los originales. Como se puede observar en la última ecuación, la cuarta solución "x_4" es igual a "c'_4". Entonces, el valor de la tercera solución "x_3" puede ser calculado con la penúltima ecuación:

$$x_3 = c'_3 - a'_{34}x_4$$

Ahora, con "x_3" y "x_4", se puede calcular "x_2" con la segunda ecuación:

$$x_2 = c'_2 - a'_{23}x_3 - a'_{24}x_4$$

Finalmente con "x_2", "x_3" y "x_4" se calcula "x_1" con la primera ecuación:

$$x_1 = c'_1 - a'_{12}x_2 - a'_{13}x_3 - a'_{14}x_4$$

En forma matricial (trabajando con la matriz aumentada), la aplicación del método de Gauss da lugar al siguiente cambio:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \end{bmatrix} \xrightarrow{gauss} \begin{bmatrix} 1 & a'_{12} & a'_{13} & a'_{14} & a'_{15} \\ 0 & 1 & a'_{23} & a'_{24} & a'_{25} \\ 0 & 0 & 1 & a'_{34} & a'_{35} \\ 0 & 0 & 0 & 1 & a'_{45} \end{bmatrix} \qquad (6.3)$$

Para convertir la matriz aumentada en una matriz diagonal superior, se deben efectuar reducciones de filas (donde se convierten en unos los elemen-

tos de la diagonal principal) y reducciones de columnas (donde se convierten en ceros los elementos que se encuentran por debajo de la diagonal principal). Estas operaciones se realizan primero para la fila y columna 1, luego para la fila y columna 2, luego para la fila y columna 3 y así sucesivamente hasta llegar a la última fila del sistema.

Puesto que los valores de la matriz se emplean sólo una vez en los cálculos, la matriz resultante puede ser almacenada en la matriz original, ahorrando así memoria.

En las siguientes ecuaciones se denominará *"p" (pivote)* al contador que determina la fila y columna que se está reduciendo, *"n"* al número de ecuaciones del sistema y "m" al número de columnas de la matriz aumentada.

Para reducir los elementos de la diagonal principal a 1, simplemente se divide la fila pivote (a_p) entre el elemento que se encuentra en la diagonal principal (a_{pp}):

$$a_{p,j} = \frac{a_{p,j}}{a_{p,p}} \quad \{ j = p+1 \rightarrow m-1 \tag{6.4}$$

Observe que en realidad no se calcula el primer elemento $a_{p,p}$, pues si se hiciera los elementos restantes se dividirían entre 1 (porque $a_{p,p}$ sería 1) y en consecuencia no cambiarían. Una vez realizada la división se puede asignar al elemento $a_{p,p}$ el valor 1, aunque ello no es necesario.

Luego, para reducir a cero los elementos de la columna "p", por debajo del elemento a_{pp}, se resta a cada fila, la fila resultante de multiplicar el elemento que se encuentra en la columna pivote (a_{ip}) por la fila pivote (a_p):

$$a_{i,j} = a_{i,j} - a_{p,j} \cdot a_{i,p} \quad \begin{cases} i = p+1 \rightarrow n-1 \\ j = p+1 \rightarrow m-1 \end{cases} \tag{6.5}$$

Al igual que en la anterior ecuación, en esta tampoco se calculan los elementos que se encuentran debajo del elemento pivote, pues si se hiciera los mismos tomarían el valor cero y en consecuencia no se restaría nada a los elementos originales. Una vez calculados los elementos restantes, se puede asignar el valor cero a los elementos $a_{i,p}$, aunque ello no es realmente necesario.

Las ecuaciones (6.4) y (6.5) se aplican una tras otra de forma intercalada para valores de "p" que van desde 0 hasta "n-1", sin embargo, la ecuación (6.5), sólo se aplica hasta "n-2", porque no existen filas por debajo del elemento $a_{n-2,n-2}$.

Una vez reducida la matriz a la forma triangular superior, las soluciones del sistema se calculan por sustitución inversa y se almacenan en la columna de las constantes:

$$a_{i,j} = a_{i,j} - \sum_{k=i+1}^{n-1} a_{i,k} \cdot a_{k,j} \quad \begin{cases} i = n-2 \rightarrow 0 \\ j = n \rightarrow m-1 \end{cases} \tag{6.6}$$

Para comprender mejor el método de Gauss, se resolverá manualmente el siguiente sistema de ecuaciones:

$$\begin{aligned} 2x_1 + 8x_2 + 2x_3 &= 14 \\ x_1 + 6x_2 - x_3 &= 15 \\ 2x_1 - x_2 + 2x_3 &= 5 \end{aligned} \tag{6.7}$$

Que escrita en forma matricial es:

$$\begin{bmatrix} 2 & 8 & 2 \\ 1 & 6 & -1 \\ 2 & -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 14 \\ 15 \\ 5 \end{bmatrix}$$

Siendo la matriz aumentada:

$$\begin{bmatrix} 2 & 8 & 2 & 14 \\ 1 & 6 & -1 & 15 \\ 2 & -1 & 2 & 5 \end{bmatrix}$$

En JavaScript las matrices son simplemente vectores de vectores, es decir son vectores donde cada elemento es a su vez un vector. La forma más sencilla de crear una matriz es escribir directamente sus elementos entre corchetes separados con comas, pero también se puede emplear el método "new Array()" para crear cada uno de los elementos (así como el vector inicial).

En este caso, la matriz aumentada y los parámetros "n", "m" y "p" son:

```
>>a=[[2,8,2,14],[1,6,-1,15],[2,-1,2,5]]; n=3; m=4; p=0;
0
```

Para acceder a los elementos de la matriz se escribe el nombre de la matriz seguido del índice de la fila (entre corchetes) y el índice de la columna (también entre corchetes). Por ejemplo para averiguar el valor del elemento de la segunda fila y tercera columna, el elemento $a_{1,2}$ (recuerde que en Javascript los vectores y en consecuencia las matrices, comienzan en 0) se escribe:

```
>>a[1][2]
-1
```

Así, se reduce la fila "p" (en este caso la primera fila, la fila 0), aplicando la ecuación (6.4):

```
>>for (j=p+1;j<m;j++) a[p][j]/=a[p][p]; a[p][p]=1; a
[[1, 4, 1, 7], [1, 6, -1, 15], [2, -1, 2, 5]]
```

Luego se reduce la columna "p" (0), aplicando la ecuación (6.5):

```
>>for (i=p+1;i<n;i++) {for (j=p+1;j<m;j++) a[i][j]=a[i][j]-a[p][j]*a[i]
[p]; a[i][p]=0;} a
[[1, 4, 1, 7], [0, 2, -2, 8], [0, -9, 0, -9]]
```

Con ello se ha reducido la primera fila y la primera columna, ahora se incrementa el valor del pivote "p" y se repite el proceso:

```
>>p++; for (j=p+1;j<m;j++) a[p][j]/=a[p][p]; a[p][p]=1; for
(i=p+1;i<n;i++) { for (j=p+1;j<m;j++) a[i][j]=a[i][j]-a[p][j]*a[i][p];
a[i][p]=0;} a
[[1, 4, 1, 7], [0, 1, -1, 4], [0, 0, -9, 27]]
```

Finalmente se incrementa el valor de "p" una vez más y se reduce la última fila ecuación (64):

```
>>p++; for (j=p+1;j<m;j++) a[p][j]/=a[p][p]; a[p][p]=1; a
[[1, 4, 1, 7], [0, 1, -1, 4], [0, 0, 1, -3]]
```

Así la matriz queda reducida a la forma triangular superior. Ahora, para calcular las soluciones se aplica la ecuación (66):

```
>>for (i=n-2;i>=0;i--) for (j=n;j<m;j++) {s=0; for (k=i+1;k<n;k++)
s+=a[i][k]*a[k][j]; a[i][j]-=s;} a
[[1, 4, 1, 6], [0, 1, -1, 1], [0, 0, 1, -3]]
```

Por lo tanto las soluciones (que se encuentran en la columna 4) son: x_1= 6; x_2=1; x_3=-3.

6.1.1. Pivotaje

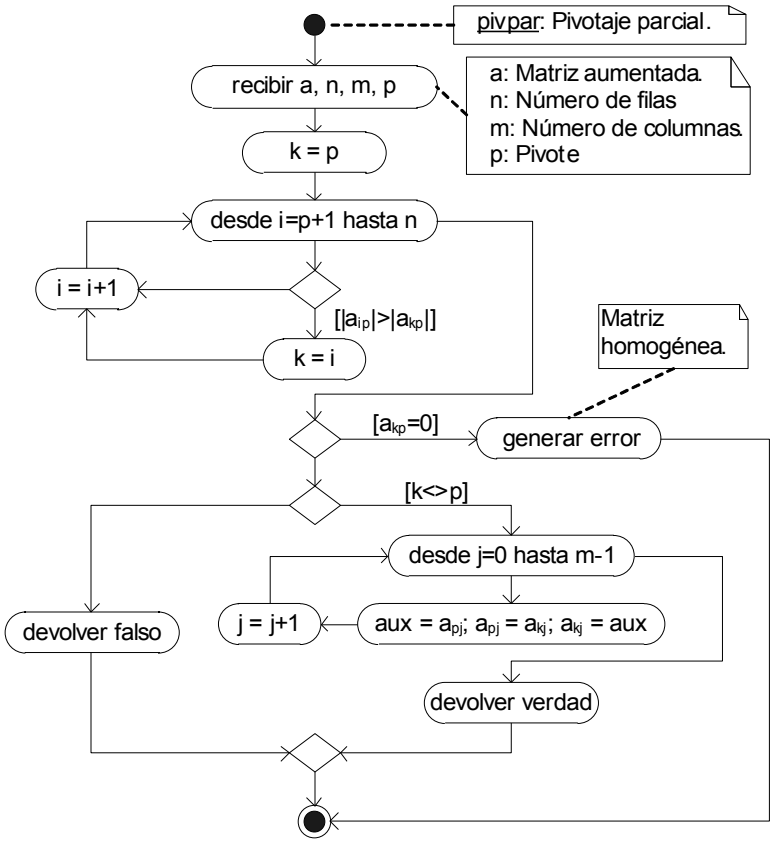
Al aplicar el método de eliminación de *Gauss* puede ocurrir que el elemento de la diagonal principal (el elemento pivote) sea cero, lo que daría lugar a un error por división entre cero. Para evitar este tipo de errores, se debe realizar un intercambio de filas y/o columnas de manera que el elemento pivote sea siempre diferente de cero. A este proceso se conoce con el nombre de "pivotaje".

El pivotaje no sólo es útil para evitar una división entre cero, sino también para disminuir el error de redondeo: cuanto mayor sea el valor del elemento pivote (en valor absoluto) menor será el error de redondeo.

En general es posible aplicar dos tipos de pivotaje: a) "Parcial", cuando el mayor valor absoluto se busca sólo en la columna pivote y b) "Total" cuando el mayor valor absoluto se busca en todas las filas y columnas no reducidas.

6.1.2. Pivotaje Parcial

El algoritmo para llevar a cabo el pivotaje parcial es el siguiente:



Como se puede ver, el proceso para llevar a cabo el pivotaje parcial es

muy sencillo: en la columna pivote, se ubica la fila donde se encuentra el elemento con el mayor valor absoluto (buscando sólo por debajo de la fila pivote). Una vez ubicado, se intercambia la fila pivote con la fila correspondiente al mayor valor absoluto.

Si el mayor valor absoluto es 0, entonces todos los términos de la columna pivote (por debajo del elemento pivote) son cero y en consecuencia el sistema es homogéneo y como tal no tiene soluciones únicas, por lo que en ese caso se generar un error.

El código respectivo, añadido a la librería "mat205.js", es:

```
function pivpar(a,p) {
    var k=p, n=a.length, m=a[0].length, aux, i, j;
    for (i=p+1;i<n;i++)
      if (abs(a[i][p])>abs(a[k][p])) k=i;
    if (a[k][p]==0) throw new Error("Matriz Homogenea");
    if (k!=p) {
      for (j=0;j<m;j++) {aux=a[p][j]; a[p][j]=a[k][j]; a[k][j]=aux;}
      return true;}
    return false;
}
```

Se puede probar esta función con la siguiente matriz:

```
>>a=[[3,2,4,6,3],[2,2,6,4,1],[8,10,0,2,8],[1,-13,0,4,3]]
[[3, 2, 4, 6, 3], [2, 2, 6, 4, 1], [8, 10, 0, 2, 8], [1, -13, 0, 4, 3]]
```

Si el pivote es 1 (p=1, segunda fila y segunda columna), deberían intercambiarse la segunda con la cuarta fila, pues en la cuarta fila se encuentra el mayor valor (-13):

```
>>pivpar(a,1)
true
```

Que devuelve "true" porque ha existido un intercambio, lo que se pude verificar viendo como han quedado los elementos de la matriz "a":

```
>>a
[[3, 2, 4, 6, 3], [1, -13, 0, 4, 3], [8, 10, 0, 2, 8], [2, 2, 6, 4, 1]]
```

Volviendo a la matriz original, para p=2 (tercera fila y tercera columna), se debería generar un error, pues el único elemento restante, debajo del pivote, es también cero:

```
>>a=[[3,2,4,6,3],[2,2,6,4,1],[8,10,0,2,8],[1,-13,0,4,3]]
[[3, 2, 4, 6, 3], [2, 2, 6, 4, 1], [8, 10, 0, 2, 8], [1, -13, 0, 4, 3]]
>>pivpar(a,2)
err: Matriz Homogenea
```

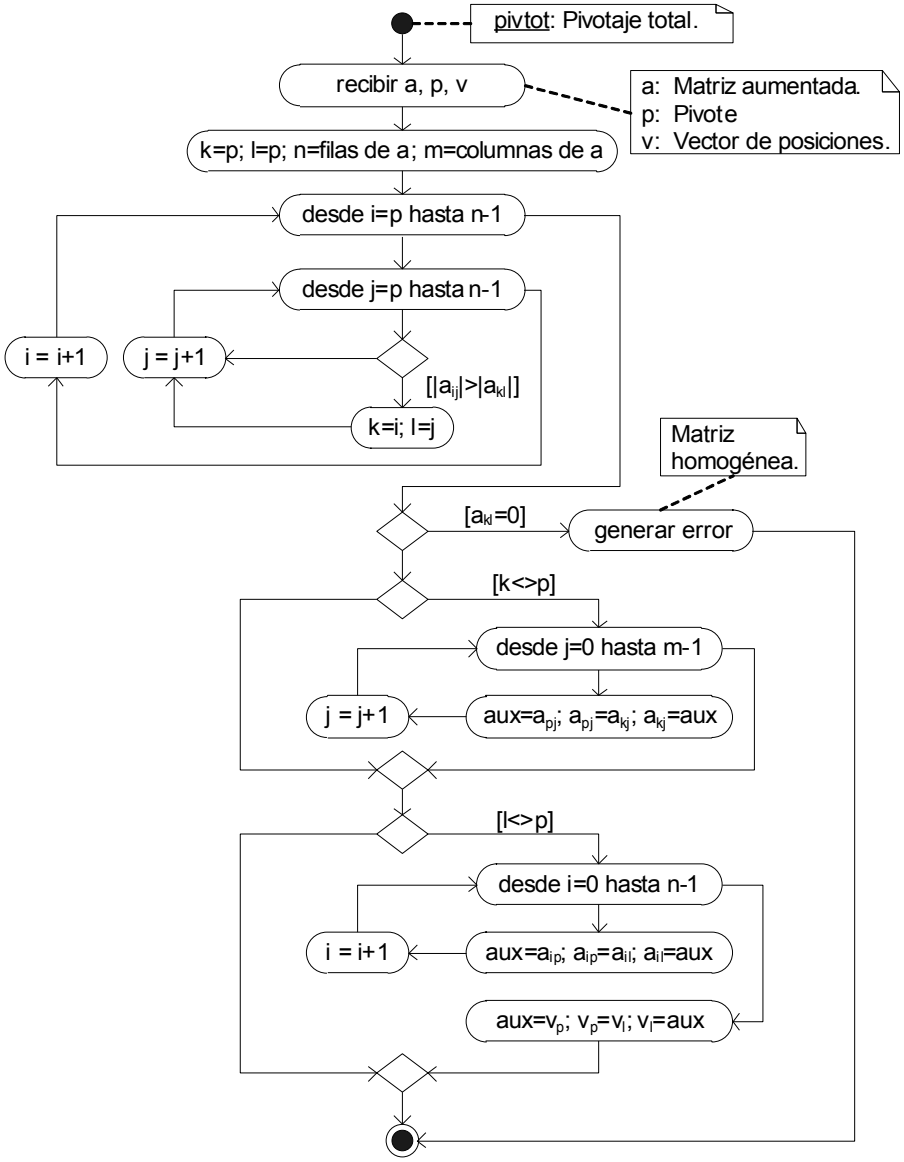
6.1.3. Pivotaje Total

Para realizar el *pivotaje total* se debe ubicar la posición del elemento con el mayor valor absoluto buscando en las filas y columnas no reducidas. Una vez ubicada la posición del elemento con el mayor valor absoluto se realizan intercambios de filas y/o columnas, según sea necesario, para llevar el mayor valor absoluto a la posición pivote.

Además, en el pivotaje total, se debe hacer un seguimiento de las columnas intercambiadas, pues un intercambio de columnas implica también un in-

tercambio en la posición de los resultados, por ejemplo, si se intercambian las columnas 2 y 4, el segundo resultado x_2 se encontrará en la fila 4 (y no en la fila 2), mientras que el cuarto resultado x_4 se encontrará en la fila 2 (y no en la fila 4).

El algoritmo para realizar el pivotaje total es el siguiente:



En este algoritmo, "v" es el vector de posiciones, esto es un vector cuyos elementos iniciales son números secuenciales de 0 a n-1 y que al final del proceso contiene el orden de los resultados.

El código respectivo, añadido a la librería "mat205.js", es:

```
function pivTot(a,p,v) {
  var aux, k=p, l=p, n=a.length, m=a[0].length;
  for (var i=p;i<n;i++)
    for (var j=p;j<n;j++)
      if (abs(a[i][j])>abs(a[k][l])) {k=i;l=j;}
  if (a[k][l]==0) throw new Error("Matriz homogenea");
```

```
    if (k!=p)
      for (j=0;j<m;j++) {aux=a[p][j]; a[p][j]=a[k][j]; a[k][j]=aux;}
    if (l!=p){
      for (i=0;i<n;i++) {
        aux=a[i][p]; a[i][p]=a[i,l]; a[i][l]=aux;}
      aux=v[p];v[p]=v[l];v[l]=aux;}
  }
```

Se puede probar el programa con la misma matriz empleada en el pivotaje parcial. Como el sistema tiene 4 ecuaciones el vector de posiciones tendrá cuatro elementos (del 0 al 3):

```
>>a=[[3,2,4,6,3],[2,2,6,4,1],[8,10,0,2,8],[1,-13,0,4,3]]; v=[0,1,2,3]
[0, 1, 2, 3]
```

Con "p=0" (primera fila y columna), el programa debería ubicar el mayor elemento (-13) e intercambiar la primera con la cuarta fila y la primera con la segunda columna:

```
>>pivtot(a,0,v); a
[[-13, 1, 0, 4, 3], [2, 2, 6, 4, 1], [10, 8, 0, 2, 8], [2, 3, 4, 6, 3]]
```

Al haberse intercambiado la primera con la segunda columna, en el vector de posiciones "v" se ha intercambiado también el primer con el segundo elemento:

```
>>v
[1, 0, 2, 3]
```

Volviendo a la matriz original, pero con "p=1" (segunda fila y segunda columna), una vez más el mayor valor absoluto es -13, pero el mismo se encuentra ya en la columna pivote (1), por lo que al no existir intercambio de columnas, no se intercambian los elementos del vector de posiciones:

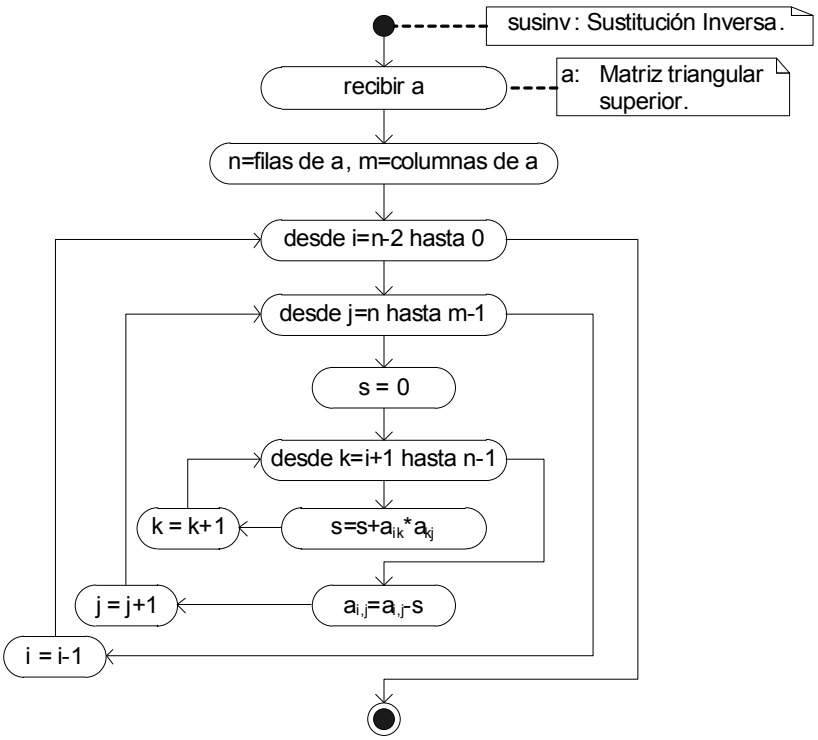
```
>>a=[[3,2,4,6,3],[2,2,6,4,1],[8,10,0,2,8],[1,-13,0,4,3]]; v=[0,1,2,3]
[0, 1, 2, 3]
>>pivtot(a,1,v); a
[[3, 2, 4, 6, 3], [1, -13, 0, 4, 3], [8, 10, 0, 2, 8], [2, 2, 6, 4, 1]]
>>v
[0, 1, 2, 3]
```

Finalmente, y trabajando con la matriz original, para "p=2" (tercera fila y tercera columna) el mayor valor es 4, por lo que deberían intercambiarse la tercera y cuarta filas y la tercera y cuarta columnas:

```
>>a=[[3,2,4,6,3],[2,2,6,4,1],[8,10,0,2,8],[1,-13,0,4,3]]; v=[0,1,2,3]
[0, 1, 2, 3]
>>pivtot(a,2,v); a
[[3, 2, 6, 4, 3], [2, 2, 4, 6, 1], [1, -13, 4, 0, 3], [8, 10, 2, 0, 8]]
>>v
[0, 1, 3, 2]
```

6.1.4. Sustitución Inversa

Una vez reducido el sistema de ecuaciones las soluciones deben ser encontradas por sustitución inversa (ecuación 6.6). El algoritmo se presenta en la siguiente página y el código respectivo, añadido a la librería "mat205.js" a continuación de la misma.




```
function susinv(a) {
  var n=a.length, m=a[0].length;
  for (var i=n-2;i>=0;i--)
    for (var j=n;j<m;j++) {
      var s=0;
      for (k=i+1;k<n;k++) s+=a[i][k]*a[k][j];
      a[i][j]-=s;}
}
```

Probando el programa con la matriz reducida del ejemplo manual, se obtiene:

```
>>a=[[1,4,1,7],[0,1,-1,4],[0,0,1,-3]]
[[1, 4, 1, 7], [0, 1, -1, 4], [0, 0, 1, -3]]
>>susinv(a); a
[[1, 4, 1, 6], [0, 1, -1, 1], [0, 0, 1, -3]]
```

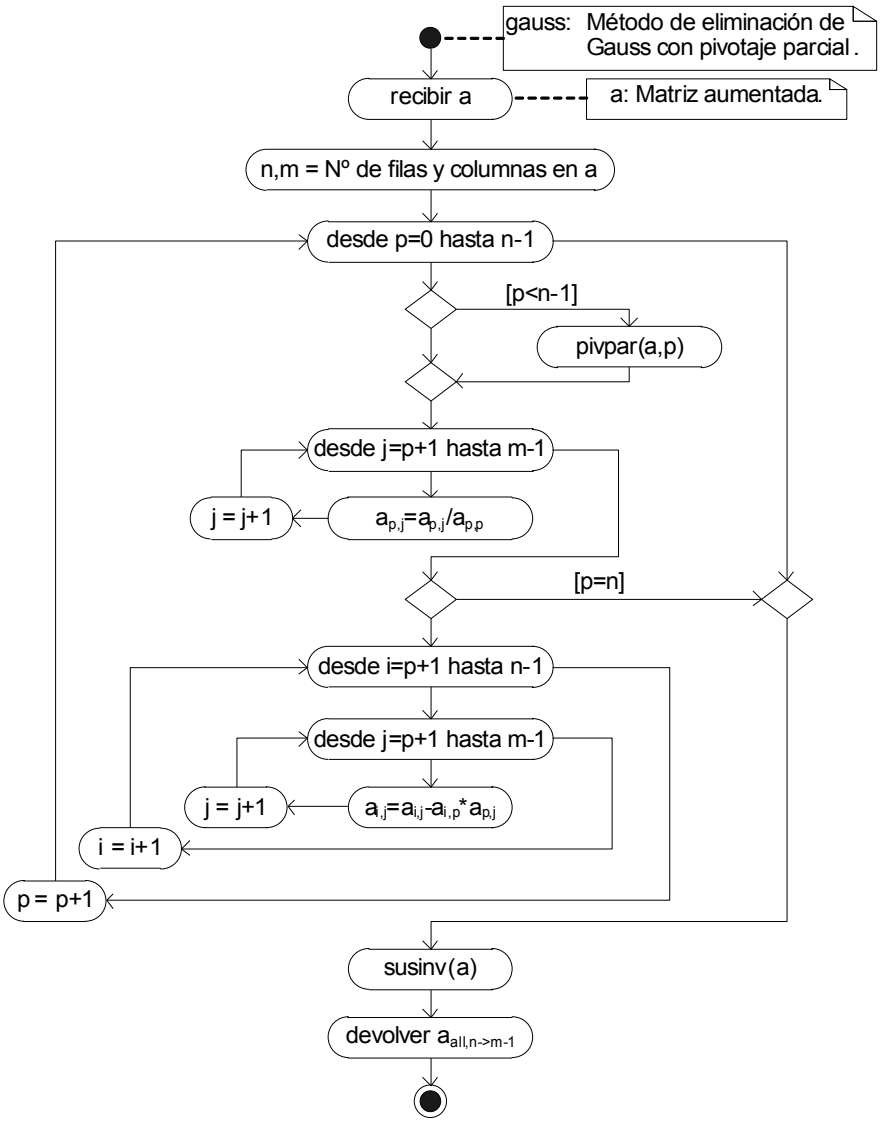
Que son los resultados obtenidos en dicho ejemplo.

6.1.5. Método de Eliminación de Gauss con pivotaje parcial

Con las funciones antes elaboradas, se puede programar el método de Gauss con pivotaje parcial.

El algoritmo se presenta en la siguiente página y el código respectivo, añadido a la librería "mat205.js", es:

```
function gauss(b) {
  var i, j, p, a=b.slice(), c;
  var n=a.length, m=a[0].length;
  for (p=0;p<n;p++) {
    if (p<n-1) pivpar(a,p);
```



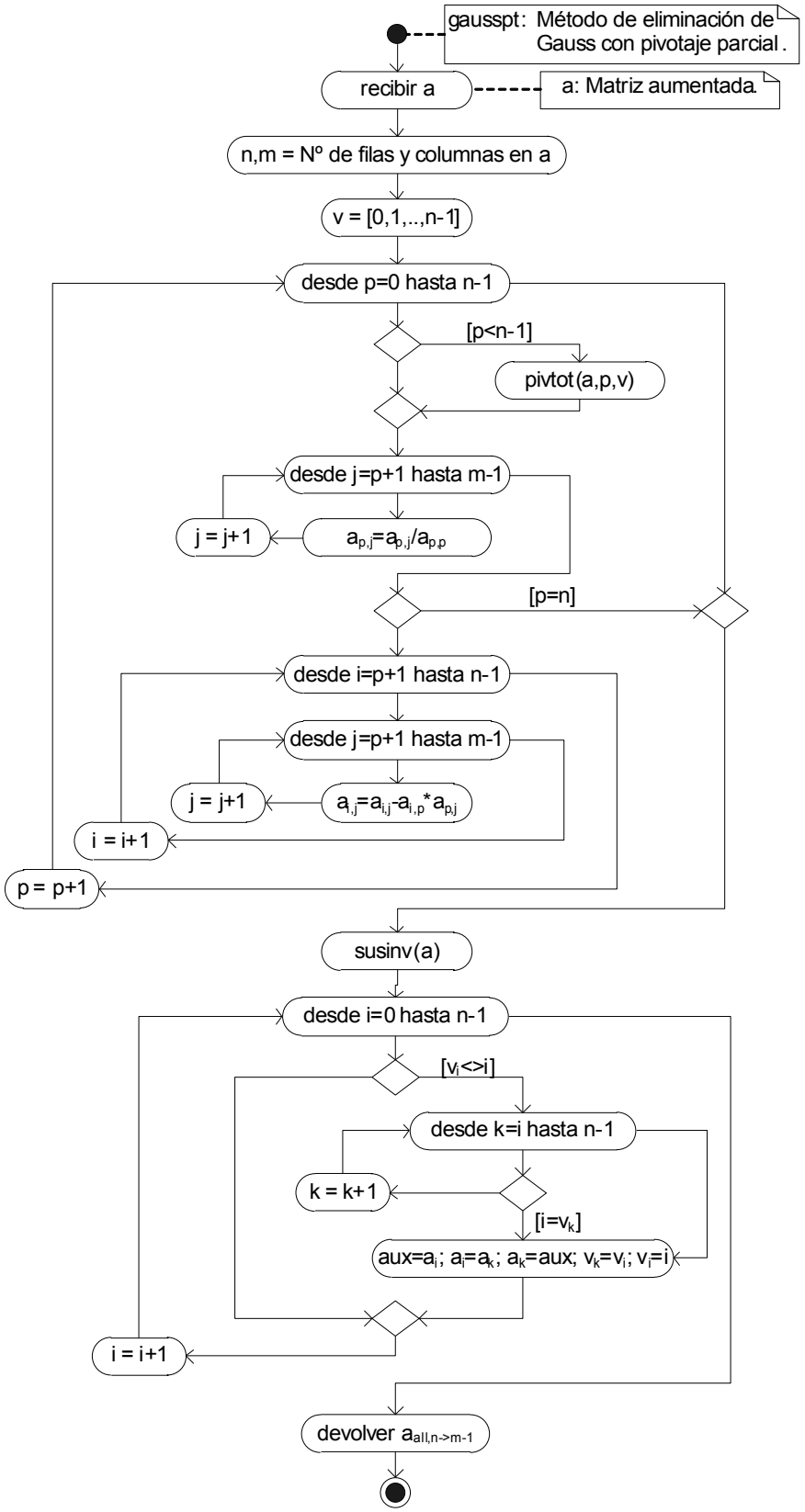

```
    for (j=p+1;j<m;j++) a[p][j]/=a[p][p];
    for (i=p+1;i<n;i++)
      for (j=p+1;j<m;j++) a[i][j]-=a[i][p]*a[p][j];}
  susinv(a);
  c=new Array(n);
  for (i=0;i<n;i++){
    c[i]=new Array(m-n);
    for (j=n;j<m;j++)
      c[i][j-n]=a[i][j];}
  return c;
}
```

Probando el programa con la matriz del ejemplo manual se obtienen los mismos resultados que en dicho ejemplo:

```
>>a=[[2,8,2,14],[1,6,-1,15],[2,-1,2,5]]
[[2, 8, 2, 14], [1, 6, -1, 15], [2, -1, 2, 5]]
>>gauss(a)
[[6], [1], [-3]]
```

6.1.6. Método de eliminación de Gauss con pivotaje total

El algoritmo cuando se realiza pivotaje total es:



Siendo el código respectivo, añadido a la librería "mat205.js":

```
function gausspt(b) {
  var i, j, p, k, l, aux, a=b.slice(), c;
  var n=a.length, m=a[0].length, v=new Array();
  for (i=0;i<n;i++) v[i]=i;
  for (p=0;p<n;p++) {
    if (p<n-1) pivtot(a,p,v);
    for (j=p+1;j<m;j++) a[p][j]/=a[p][p];
    for (i=p+1;i<n;i++)
      for (j=p+1;j<m;j++) a[i][j]-=a[i][p]*a[p][j];}
  susinv(a);
  for (i=0;i<n;i++)//Se ordena la matriz resultante
    if (v[i]!=i) {
      for (k=i;k<n;k++) if (i==v[k]) break;
      aux=a[i];a[i]=a[k];a[k]=aux;v[k]=v[i];v[i]=i;}
    c=new Array(n);
  for (i=0;i<n;i++){//Se devuelven los
    c[i]=new Array(m-n);
    for (j=n;j<m;j++)
      c[i][j-n]=a[i][j];}
  return c;
}
```

Probando el programa con la matriz del ejemplo manual se obtiene:

```
>>a=[[2,8,2,14],[1,6,-1,15],[2,-1,2,5]]
[[2, 8, 2, 14], [1, 6, -1, 15], [2, -1, 2, 5]]
>>gausspt(a)
[[6], [1], [-3]]
```

Que son los resultados obtenidos en el ejemplo manual.

Gauss y gausspt, permiten calcular también la matriz inversa, para ello simplemente se añade a la matriz de coeficientes (o cualquier matriz cuadrada) la matriz unidad.

Por ejemplo, para calcular la inversa de la matriz de coeficientes del ejemplo manual, se crea la matriz:

```
>>a=[[2,8,2,1,0,0],[1,6,-1,0,1,0],[2,-1,2,0,0,1]]
[[2, 8, 2, 1, 0, 0], [1, 6, -1, 0, 1, 0], [2, -1, 2, 0, 0, 1]]
```

Entonces, mandando esta matriz a "gauss" se obtiene la matriz inversa:

```
>>precision(gauss(a),9)
[[-0.305555556, 0.5, 0.555555556], [0.111111111, 0, -0.111111111],
[0.361111111, -0.5, -0.111111111]]
```

El mismo resultado se obtiene con "gausspt":

```
>>precision(gausspt(a),9)
[[-0.305555556, 0.5, 0.555555556], [0.111111111, 0, -0.111111111],
[0.361111111, -0.5, -0.111111111]]
```

Dado que la matriz inversa es de utilidad práctica en algunas operaciones, es conveniente elaborar una función para la misma. Dicha función se presenta en la siguiente página y ha sido añadida a la librería "mat205.js".

```
function mInv(a){
  if (isMatrixr(a))
    if (a.length==a[0].length) {
      var b=mCopy(a), n=a.length, i, j;
      for (i=0;i<n;i++)
        for (j=0;j<n;j++)
          b[i][j+n]= i==j ? 1: 0;
      return gauss(b);
    }
  else throw new Error("matriz no cuadrada (inv)");
}
```

Esta función ha sido integrada en la función "inv", que al igual que "add", "sub", "mul", "div", "abs", "copy" y algunas otras más, operan tanto con números complejos como con matrices (además de números reales).

Volviendo a calcular la inversa de la matriz de coeficientes con "inv" se obtiene (como era de esperar) el mismo resultado:

```
>>a=[[2,8,2],[1,6,-1],[2,-1,2]];
[[2, 8, 2], [1, 6, -1], [2, -1, 2]]
>>b=inv(a); precision(b,9)
[[-0.305555556, 0.5, 0.555555556], [0.111111111, 0, -0.111111111],
[0.361111111, -0.5, -0.111111111]]
```

Como se sabe, si se multiplica la matriz inversa por el vector de constantes, se obtienen las soluciones sel sistema:

```
>>c=[[14],[15],[5]];
[[14], [15], [5]]
>>precision(mul(b,c),9)
[[6], [1], [-3]]
```

Que constituye otra forma de resolver sistemas de ecuaciones lineales.

Otra propiedad que resulta de utilidad cuando se trabaja con matrices es el determinante. Tradicionalmente el determinante se calcula con la regla de Cramer, sin embargo, la misma sólo es de utilidad práctica para sistemas de 2 ecuaciones con 2 incógnitas. Para sistemas con un mayor número de ecuaciones el método de Cramer es extremadamente ineficiente, así por ejemplo, para calcular el determinante de un sistema de 10 ecuaciones lineales con 10 incógnitas se requieren aproximadamente 70 millones de operaciones, un número exorbitante, inclusive para las computadoras actuales.

Por ello el determinante se calcula en la práctica recurriendo a uno de los métodos de eliminación, siendo el método de Gauss ligeramente más eficiente que los otros. Para calcular el determinante con este método simplemente se multiplican los elementos que quedan en la diagonal principal, cuidando de no dividirlos entre sí mismos. Además se debe cambiar el signo con cada intercambio de filas (o cambiar el signo final si el número de intercambios es impar). Para ello se emplea el resultado lógico devuelto por "pivpar", el cual informa si ha existido cambio (true) o no (false).

La función que calcula el determinante, añadida a la unidad "mat205.js", es:

```
function mDet(b) {
  var i, j, p, c, a=mCopym(b), d=1;
```

```
  var n=a.length, m=a[0].length;
  for (p=0;p<n-1;p++) {
    if (p<n-1) if (pivpar(a,p)) d*=-1;
    d*=a[p][p];
    for (j=p+1;j<m;j++) a[p][j]/=a[p][p];
    for (i=p+1;i<n;i++)
      for (j=p+1;j<m;j++) a[i][j]-=a[i][p]*a[p][j];}
  d*=a[p][p];
  return d;
}
function det(a) {
  if (isMatrixr(a))
    if (a.length==a[0].length) return mDet(a);
    else throw new Error("matriz no cuadrada (det)");
  throw new Error("matriz irregular (det)");
}
```

Así, para calcular el determinante de la matriz de coeficientes se escribe:

```
>>a=[[2,8,2],[1,6,-1],[2,-1,2]]
[[2, 8, 2], [1, 6, -1], [2, -1, 2]]
>>det(a)
-36
```

6.2. MÉTODO GAUSS-JORDÁN

El método de eliminación de Gauss - Jordán, es muy similar al método de eliminación de Gauss, sólo que en este caso la matriz de los coeficientes es transformada en una matriz identidad (o unidad). Así si se aplica el método de eliminación de Gauss-Jordán al siguiente sistema:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= c_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= c_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= c_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= c_4 \end{aligned} \qquad (6.8)$$

Se transforma en:

$$\begin{aligned} x_1 + 0 + 0 + 0 &= c'_1 \\ 0 + x_2 + 0 + 0 &= c'_2 \\ 0 + 0 + x_3 + 0 &= c'_3 \\ 0 + 0 + 0 + x_4 &= c'_4 \end{aligned} \qquad (6.9)$$

Por lo tanto las soluciones se encuentran directamente en la columna de las constantes. En forma matricial, la aplicación del método de Gauss-Jordán da lugar a la siguiente transformación:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \end{bmatrix} \xrightarrow{gauss-jordán} \begin{bmatrix} 1 & 0 & 0 & 0 & a'_{15} \\ 0 & 1 & 0 & 0 & a'_{25} \\ 0 & 0 & 1 & 0 & a'_{35} \\ 0 & 0 & 0 & 1 & a'_{45} \end{bmatrix} \qquad (6.10)$$

Las operaciones que se efectúan para llevar la matriz aumentada a la matriz identidad son prácticamente las mismas que en el método de Gauss, excepto que ahora la eliminación de las columnas se la realiza no sólo en la parte inferior de la diagonal principal, sino también en la parte superior de la misma. Por lo tanto, para la reducción de las filas se emplea la misma ecuación que en el método de Gauss:

$$a_{p,j} = \frac{a_{p,j}}{a_{p,p}} \qquad \{ j = p+1 \rightarrow m-1 \tag{6.11}$$

Y para la reducción de columnas se cambia el límite inferior de las filas a 0:

$$a_{i,j} = a_{i,j} - a_{p,j} \cdot a_{i,p} \quad \begin{cases} i = 0 \rightarrow n-1 \\ j = p+1 \rightarrow m-1, \;\; si \;\; i \neq p \end{cases} \tag{6.12}$$

Al igual que en el método de Gauss, no es necesario calcular los valores 1 (de la diagonal principal) ni ceros (por encima y debajo de la misma), sin embargo, dichos valores pueden ser asignados, después de aplicar la ecuación (6.11) y al final de cada iteración en la ecuación (6.12).

Otra diferencia es que ahora el pivote va desde 0 hasta n-1,

Para comprender mejor el método, se volverá a resolver el sistema de ecuaciones (6.7)

Los datos son:

```
>>a=[[2,8,2,14],[1,6,-1,15],[2,-1,2,5]];p=0;n=a.length;m=a[0].length;
4
```

Siendo "n" y "m" el número de filas y columnas de la matriz. Ahora se reduce la primera fila y la primera columna (p=0), aplican las ecuaciones (6.11) y (6.12):

```
>>for(j=p+1;j<m;j++) a[p][j]/=a[p][p]; a[p][p]=1; for(i=0;i<n;i++) if
(i!=p) {for(j=p+1;j<m;j++) a[i][j]-=a[p][j]*a[i][p]; a[i][p]=0;} a
[[1, 4, 1, 7], [0, 2, -2, 8], [0, -9, 0, -9]]
```

Se incrementa el valor del pivote (p=1) y se reduce la segunda fila y la segunda columna:

```
>>p++; for(j=p+1;j<m;j++) a[p][j]/=a[p][p]; a[p][p]=1; for(i=0;i<n;i++)
if (i!=p) {for(j=p+1;j<m;j++) a[i][j]-=a[p][j]*a[i][p]; a[i][p]=0;} a
[[1, 0, 5, -9], [0, 1, -1, 4], [0, 0, -9, 27]]
```

Finalmente se incrementa una vez más el valor del pivote (p=2) y se reduce la tercera fila y tercera columna:

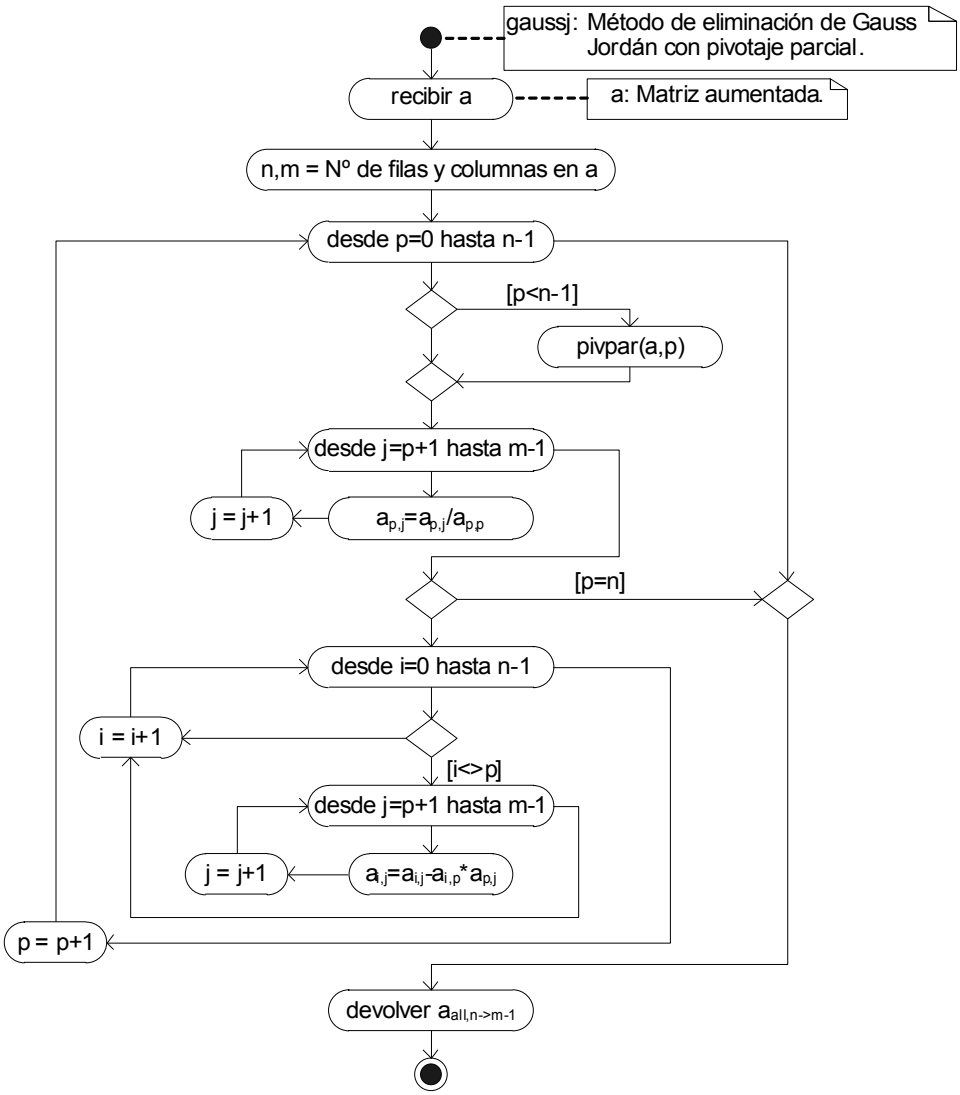
```
>>p++; for(j=p+1;j<m;j++) a[p][j]/=a[p][p]; a[p][p]=1; for(i=0;i<n;i++)
if (i!=p) {for(j=p+1;j<m;j++) a[i][j]-=a[p][j]*a[i][p]; a[i][p]=0;} a
[[1, 0, 0, 6], [0, 1, 0, 1], [0, 0, 1, -3]]
```

Como se puede ver, la matriz de coeficientes queda reducida a una matriz unidad y en la última columna se encuentran las soluciones del sistema: $x_1=6$; $x_2=1$; $x_3=-3$.

6.2.1. Algoritmo y código - pivotaje parcial

El pivotaje en sí, tanto parcial como total, no cambia, por lo que en este método se emplean las funciones desarrolladas en el anterior método.

El algoritmo del método se presenta en la siguiente página.



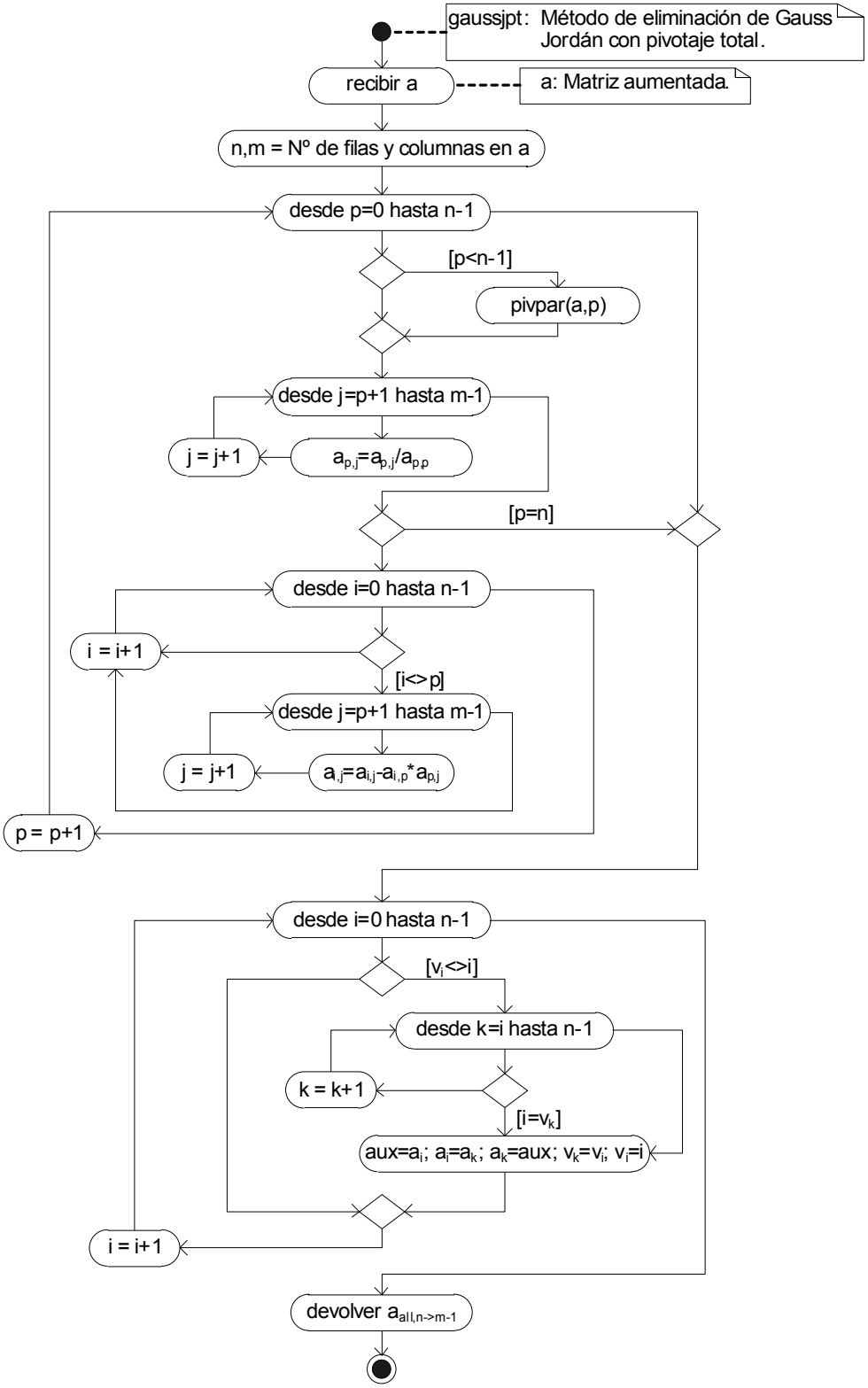
Y el codigo respectivo, añadido a la librería "mat205.js" es:

```
function gaussj(b) {
  var i, j, p, a, c;
  var n=b.length, m=b[0].length;
  a=new Array(n); for (i=0;i<n;i++) a[i]=b[i].slice();
  for (p=0;p<n;p++) {
    if (p<n-1) pivpar(a,p);
    for (j=p+1;j<m;j++) a[p][j]/=a[p][p];
    for (i=0;i<n;i++)
      if (i!=p) for (j=p+1;j<m;j++) a[i][j]-=a[i][p]*a[p][j];}
  c=new Array(n);//Matriz resultante
  for (i=0;i<n;i++){
    c[i]=new Array(m-n);
    for (j=n;j<m;j++)
      c[i][j-n]=a[i][j];}
  return c;
}
```

Que puede ser probado con la matriz del ejemplo manual:

```
>>a=[[2,8,2,14],[1,6,-1,15],[2,-1,2,5]]; gaussj(a)
[[6], [1], [-3]]
```

6.2.2. Algoritmo y código - pivotaje total



Y el código respectivo, añadido a la unidad "mat205.js", es:

```
function gaussjpt(b) {
  var i, j, p, k, l, aux, a, c;
  var n=b.length, m=b[0].length, v=new Array();
  a=new Array(n); for (i=0;i<n;i++) a[i]=b[i].slice();
  for (i=0;i<n;i++) v[i]=i;
  for (p=0;p<n;p++) {
    if (p<n-1) pivtot(a,p,v);
    for (j=p+1;j<m;j++) a[p][j]/=a[p][p];
    for (i=0;i<n;i++)
      if (i!=p) for (j=p+1;j<m;j++) a[i][j]-=a[i][p]*a[p][j];}
  for (i=0;i<n;i++)//Ordenamiento de filas
    if (v[i]!=i) {
      for (k=i;k<n;k++) if (i==v[k]) break;
      aux=a[i];a[i]=a[k];a[k]=aux;v[k]=v[i];v[i]=i;}
  c=new Array(n);//Matriz resultante
  for (i=0;i<n;i++){
    c[i]=new Array(m-n);
    for (j=n;j<m;j++)
      c[i][j-n]=a[i][j];}
  return c;
}
```

Haciendo correr el programa con los datos del ejemplo manual, se obtienen los resultados correctos:

```
>>a=[[2,8,2,14],[1,6,-1,15],[2,-1,2,5]]; gaussjpt(a)
[[6], [1], [-3]]
```

6.3. MÉTODO DE ELIMINACIÓN DE CROUT

El método de Crout es el primero de los métodos de factorización (o descomposición) que se estudia en este tema. El fundamento general de estos métodos es el siguiente:

Dada la matriz A:

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad (6.13)$$

Se puede demostrar que la misma puede ser expresada como el producto de dos matrices triangulares, una inferior "L" y otra superior "U":

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}=\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix}\times\begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} \\ 0 & 1 & u_{23} & u_{24} \\ 0 & 0 & 1 & u_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}=L\times U \qquad (6.14)$$

A este proceso, el de calcular los elementos de las matrices triangulares "L" y "U" para una matriz cuadrada "A", se conoce como *descomposición LU* o

factorización LU y es el proceso en el que se basan los métodos de Crout, Doolittle y Cholesky.

El proceso de descomposición no es único, las posibles combinaciones de "L" y "U" son infinitas. En general, sin embargo, son tres las formas de descomposición más empleadas en la práctica.

La descomposición de **Crout**, que corresponde a la descomposición mostrada en el ejemplo, es decir aquella en la cual los elementos de la diagonal principal en "U" son unos.

La descomposición de **Doolittle**, en la cual los elementos de la diagonal principal en "L" son unos, es decir:

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}=\begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix}\times\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}=L\times U \qquad (6.15)$$

Y la descomposición de **Cholesky**, en la cual U es la transpuesta de L, es decir:

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}=\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix}\times\begin{bmatrix} l_{11} & l_{21} & l_{31} & l_{34} \\ 0 & l_{22} & l_{32} & l_{34} \\ 0 & 0 & l_{33} & l_{43} \\ 0 & 0 & 0 & l_{44} \end{bmatrix}=L\times U \qquad (6.16)$$

Tomando en cuenta el siguiente sistema de ecuaciones:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= b_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= b_4 \end{aligned} \qquad (6.17)$$

Una vez realizada la factorización, las soluciones pueden ser calculadas, empleando las matrices triangulares obtenidas:

$$A\cdot x=L\cdot U\cdot x=b \qquad (6.18)$$

Multiplicando ambos lados de la ecuación por L^{-1}, se obtiene:

$$L^{-1}\cdot L\cdot U\cdot x=I\cdot U\cdot x=U\cdot x=L^{-1}\cdot b=c \qquad (6.19)$$

De donde se obtienen las igualdades:

$$U\cdot x=c \qquad (6.20)$$

$$L^{-1}\cdot b=c \qquad (6.21)$$

Que corresponden a dos sistemas de ecuaciones lineales. Para resolverlos primero se calculan los valores de "c" y con ese fin se multiplican ambos lados de la ecuación (6.21) por L:

$$L\cdot L^{-1}\cdot b=I\cdot b=b=L\cdot c \qquad (6.22)$$

Con lo que se forma el sistema de ecuaciones lineales:

$$L\cdot c=b \qquad (6.23)$$

Que puede ser resuelto directamente, porque es un sistema triangular inferior. Así para el sistema (6.17) el sistema triangular inferior que se forma (asumiendo la descomposición de Crout) es:

$$\begin{aligned} l_{11}c_1 &= b_1 \\ l_{21}c_1 + l_{22}c_2 &= b_2 \\ l_{31}c_1 + l_{32}c_2 + l_{33}c_3 &= b_3 \\ l_{41}c_1 + l_{42}c_2 + l_{43}c_3 + l_{44}c_4 &= b_4 \end{aligned} \qquad (6.24)$$

Como se puede ver, c_1 se calcula con la primera ecuación, c_2 con la segunda y así sucesivamente. En general, para un sistema con "n" ecuaciones los valores de "c" se calculan con:

$$c_i = \frac{b_i - \sum_{k=1}^{i-1} l_{ik} \cdot c_k}{l_{ii}} \quad \{ i = 1 \rightarrow n \qquad (6.25)$$

Esta ecuación puede ser generalizada para el caso en el que "b" es una matriz en lugar de un vector (que es lo que ocurre cuando se resuelven, simultáneamente, sistemas de ecuaciones lineales o se calcula la inversa):

$$c_{ij} = \frac{b_{ij} - \sum_{k=1}^{i-1} l_{ik} \cdot c_{kj}}{l_{ii}} \quad \begin{cases} i = 1 \rightarrow n \\ j = 1 \rightarrow m \end{cases} \qquad (6.26)$$

Donde "m" es el número de columnas de la matriz "b" y "n" es el número de filas del sistema de ecuaciones.

Una vez calculado el vector "c", se encuentran las soluciones resolviendo el sistema de ecuaciones lineales (6.20). En este caso se trata de un sistema triangular superior y por lo tanto puede ser resuelto por sustitución hacia atrás (igual que en el método de Gauss). Así para el sistema (6.17) (asumiendo la descomposición de Crout) se tiene:

$$\begin{aligned} x_1 + u_{12}x_2 + u_{13}x_3 + u_{14}x_4 &= c_1 \\ x_2 + u_{23}x_3 + u_{24}x_4 &= c_2 \\ x_3 + u_{34}x_4 &= c_3 \\ x_4 &= c_4 \end{aligned} \qquad (6.27)$$

Es decir se calcula x_4 con la última ecuación, x_3 con la penúltima y así sucesivamente. En general, para un sistema con "n" ecuaciones los valores de "x_i" se calculan con:

$$x_i = \frac{c_i - \sum_{k=i+1}^{n} u_{ik} \cdot x_k}{u_{ii}} \quad \{ i = n \rightarrow 1 \qquad (6.28)$$

Para el caso general en el que "c" es una matriz, se tiene:

$$x_{ij} = \frac{c_{ij} - \sum_{k=i+1}^{n} u_{ik} \cdot x_{kj}}{u_{ii}} \quad \begin{cases} i = n \rightarrow 1 \\ j = 1 \rightarrow m \end{cases} \qquad (6.29)$$

Donde "m" es el número de columnas de la matriz "c" y "n" es el número de filas.

La factorización (o descomposición) matricial es particularmente útil cuando en un problema dado sólo cambian las constantes del sistema. En ese caso se calculan las matrices "L" y "U" (se realiza la factorización) una sola vez y luego si el valor de las constantes cambia, se pueden calcular las nuevas soluciones directamente con las ecuaciones (6.23) y (6.26) (sin necesidad de recalcular "L" y "U").

En base a esta exposición, las ecuaciones que permiten calcular las matrices "L" y "U" en la descomposición de Crout, son las siguientes:

$$\left.\begin{array}{l} l_{ip}=a_{ip}-\sum_{k=1}^{p-1} l_{ik}\cdot u_{kp} \quad \{i=p\rightarrow n \\ u_{pp}=1 \\ u_{pj}=\dfrac{a_{pj}-\sum_{k=1}^{p-1} l_{pk}\cdot u_{kj}}{l_{pp}} \quad \{j=p+1\rightarrow n \end{array}\right\} \; p=1\rightarrow n \qquad (6.30)$$

Estas ecuaciones han sido deducidas tomando en cuenta que la multiplicación de las filas de la matriz "L" por la primera columna de la matriz "U" es igual a la primera columna de la matriz "A"; que la multiplicación de la primera fila de la matriz "L" por las columnas 2 a "n" de la matriz "U" es igual a los elementos 2 a "n" de la primera fila de la matriz "A"; que la multiplicación de las filas de la matriz "L" por la segunda columna de "U" es igual a la segunda columna de "A"; que la multiplicación de la segunda fila de la matriz "L" por las columnas 3 a "n" de la matriz "U" es igual a los elementos 3 a "n" de "A" y así sucesivamente.

Si no se almacenan los ceros ni los unos de las matrices "L" y "U", ambas matrices pueden ser almacenadas en una sola, así las dos matrices de la ecuación (6.15), pueden ser almacenadas como:

$$\begin{bmatrix} l_{11} & u_{12} & u_{13} & u_{14} \\ l_{21} & l_{22} & u_{23} & u_{24} \\ l_{31} & l_{32} & l_{33} & u_{34} \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \qquad (6.31)$$

Donde se sabe que la matriz triangular "U" tiene unos en su diagonal principal y que el resto de los elementos son cero. Así se ahorra memoria y, más importante aún, se simplifican los cálculos.

Si estos elemento se almacenan en la matriz "A", las ecuaciones (6.30) se transforman en:

$$\left.\begin{array}{l} a_{ip}=a_{ip}-\sum_{k=1}^{p-1} a_{ik}\cdot a_{kp} \quad \{i=p\rightarrow n \\ a_{pj}=\dfrac{a_{pj}-\sum_{k=1}^{p-1} a_{pk}\cdot a_{kj}}{a_{pp}} \quad \{j=p+1\rightarrow n \end{array}\right\} \; p=1\rightarrow n \qquad (6.32)$$

Las ecuaciones (6.23) y (6.26), con las cuales se calculan las soluciones del sistema, deben ser reescritas para tomar en cuenta este cambio, pero, si además las soluciones se almacenan en la matriz de constantes "b" y se toma

en cuenta que los elementos de la diagonal principal en "U" son unos, se transforman en:

$$b_{ij} = \frac{b_{ij} - \sum_{k=1}^{i-1} a_{ik} \cdot b_{kj}}{a_{ii}} \quad \begin{cases} i = 1 \rightarrow n \\ j = 1 \rightarrow m \end{cases} \tag{6.33}$$

$$b_{ij} = b_{ij} - \sum_{k=i+1}^{n} a_{ik} \cdot b_{kj} \quad \begin{cases} i = n \rightarrow 1 \\ j = 1 \rightarrow m \end{cases} \tag{6.34}$$

A esta forma se denomina esquema compacto y es la forma más empleada en la práctica. Al momento de programar estas ecuaciones, se debe recordar que en Javascript el primer índice de las matrices y vectores es 0, por lo que los límites de las ecuaciones deben ser cambiados para tomar en cuenta este hecho.

Para comprender mejor el método se aplicará manualmente el mismo al siguiente sistema de ecuaciones:

$$\begin{array}{rcrcrcl} 2x_1 & + & 8x_2 & + & 2x_3 & = & 14 \\ x_1 & + & 6x_2 & - & x_3 & = & 13 \\ 2x_1 & - & x_2 & + & 2x_3 & = & 5 \end{array} \tag{6.35}$$

Los datos son:

```
>>a=[[2,8,2],[1,6,-1],[2,-1,2]]; b=[[14],[13],[5]]; n=a.length; p=0;
0
```

Con los mismos se calculan los elementos de la primera columna de "L" y de la primera fila de "U" (ecuaciones (6.32)):

```
>>for(i=p;i<n;i++) {s=0; for(k=0;k<p;k++) s+=a[i][k]*a[k][p]; a[i][p]-
=s;} for(j=p+1;j<n;j++) {s=0; for(k=0;k<p;k++) s+=a[p][k]*a[k][j]; a[p]
[j]=(a[p][j]-s)/a[p][p];} a
[[2, 4, 1], [1, 6, -1], [2, -1, 2]]
```

Luego, se incrementa "p" y se calculan los elementos de la segunda columna de "L" y la segunda fila de "U":

```
>>p++; for(i=p;i<n;i++) {s=0; for(k=0;k<p;k++) s+=a[i][k]*a[k][p]; a[i]
[p]-=s;} for(j=p+1;j<n;j++) {s=0; for(k=0;k<p;k++) s+=a[p][k]*a[k][j];
a[p][j]=(a[p][j]-s)/a[p][p];} a
[[2, 4, 1], [1, 2, -1], [2, -9, 2]]
```

Finalmente se incrementa el valor de "p" (a 2) y se calcula la última columna (de "L"):

```
>>p++; for(i=p;i<n;i++) {s=0; for(k=0;k<p;k++) s+=a[i][k]*a[k][p]; a[i]
[p]-=s;} for(j=p+1;j<n;j++) {s=0; for(k=0;k<p;k++) s+=a[p][k]*a[k][j];
a[p][j]=(a[p][j]-s)/a[p][p];} a
[[2, 4, 1], [1, 2, -1], [2, -9, -9]]
```

De esa manera se obtienen las matrices triangulares "L" (los elementos de la diagonal principal e inferiores) y "U" (los elementos encima de la diagonal principal).

Ahora se calcula las soluciones aplicando primero la ecuación (6.33) (con j=0 pues en este caso "b" es un vector columna):

```
>>j=0; for(i=0;i<n;i++){s=0;for(k=0;k<i;k++) s+=a[i][k]*b[k][j]; b[i][j]=
(b[i][j]-s)/a[i][i];}; b
[[7], [3], [-2]]
```

Y luego la ecuación (6.34):

```
>>for(i=n-1;i>=0;i--){s=0;for(k=i+1;k<n;k++) s+=a[i][k]*b[k][j]; b[i][j]-
=s;} b
[[5], [1], [-2]]
```

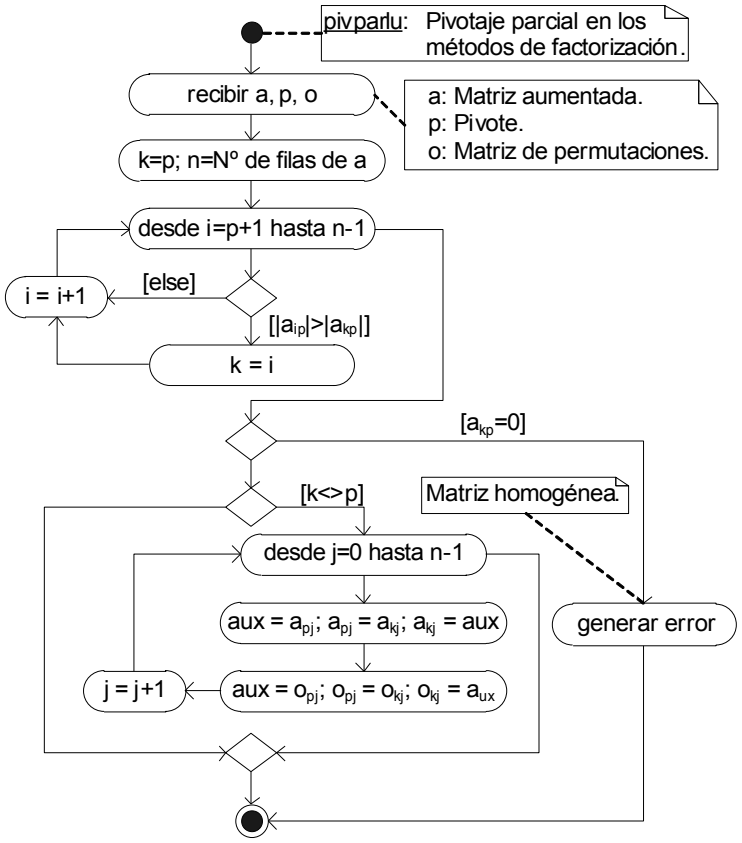
Que son las soluciones del sistema (x_1=5; x_2=1; x_3=-2).

6.3.1. Pivotaje Parcial en los métodos de factorización

Al igual que en los métodos de Gauss y Gauss Jordán en los métodos de factorización se debe evitar que el elemento pivote sea cero y con ese fin es necesario realizar un pivotaje parcial.

En estos métodos, sin embargo, se debe mantener un registro de los intercambios de filas realizados, pues es necesario ordenar la matriz de las constantes ("b") en función a dichos intercambios. Alternativamente se puede trabajar con la matriz aumentada, pero con ello se pierde la principal ventaja de estos métodos: la de calcular nuevas soluciones empleando las matrices "l" y "u" ya calculadas (matriz factorizada).

El algoritmo para el pivotaje parcial en estos casos es:



Como se puede ver, para el seguimiento de los intercambios realizados, se emplea una matriz de permutaciones "o", que inicialmente será igual a la matriz unidad. Cada vez que se intercambien dos filas en la matriz "A", se intercambiarán las mismas filas en la matriz de permutaciones "o", entonces,

los intercambios de filas hechos en la matriz "A", podrán ser replicados en la matriz "b", simplemente multiplicándola por la matriz "o".

El código, añadido a la librería "mat205.js", es:

```
function pivparlu(a,p,o) {
    var aux,i,k=p,n=a.length;
    for (i=p+1;i<n;i++)
      if (abs(a[i][p])>abs(a[k][p])) k=i;
    if (a[k][p]==0) throw new Error("Sistema homogéneo (pivparlu)");
    if (k!=p)
      for (j=0;j<n;j++) {
        aux=a[p][j]; a[p][j]=a[k][j]; a[k][j]=aux;
        aux=o[p][j]; o[p][j]=o[k][j]; o[k][j]=aux;
      }
}
```

Haciendo correr el programa para la matriz de coeficientes del siguiente sistema (donde evidentemente se requieren intercambios de filas):

$$\begin{aligned} 2x_2 + x_3 &= 5 \\ x_1 &= -1 \\ 3x_1 + x_3 &= -2 \end{aligned} \qquad (6.36)$$

Se obtiene:

```
>>a=[[0,2,1],[1,0,0],[3,0,1]];o=[[1,0,0],[0,1,0],[0,0,1]];
pivparlu(a,0,o); write(a); o
[[3, 0, 1], [1, 0, 0], [0, 2, 1]]
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

Donde, como se puede ver, se han intercambiado las filas 1 y 3, cambio que se refleja en la matriz de permutaciones "o" (recuerde que en las consolas se debe emplear "console.log" en lugar de "write").

Haciendo correr el programa con la matriz resultante, para "p=1", se obtiene:

```
>>pivparlu(a,1,o); write(a); o
[[3, 0, 1], [0, 2, 1], [1, 0, 0]]
[[0, 0, 1], [1, 0, 0], [0, 1, 0]]
```

Que una vez más corresponde al intercambio de filas esperado (filas 2 y 3) y el respectivo intercambio en la matriz de permutaciones ("o").

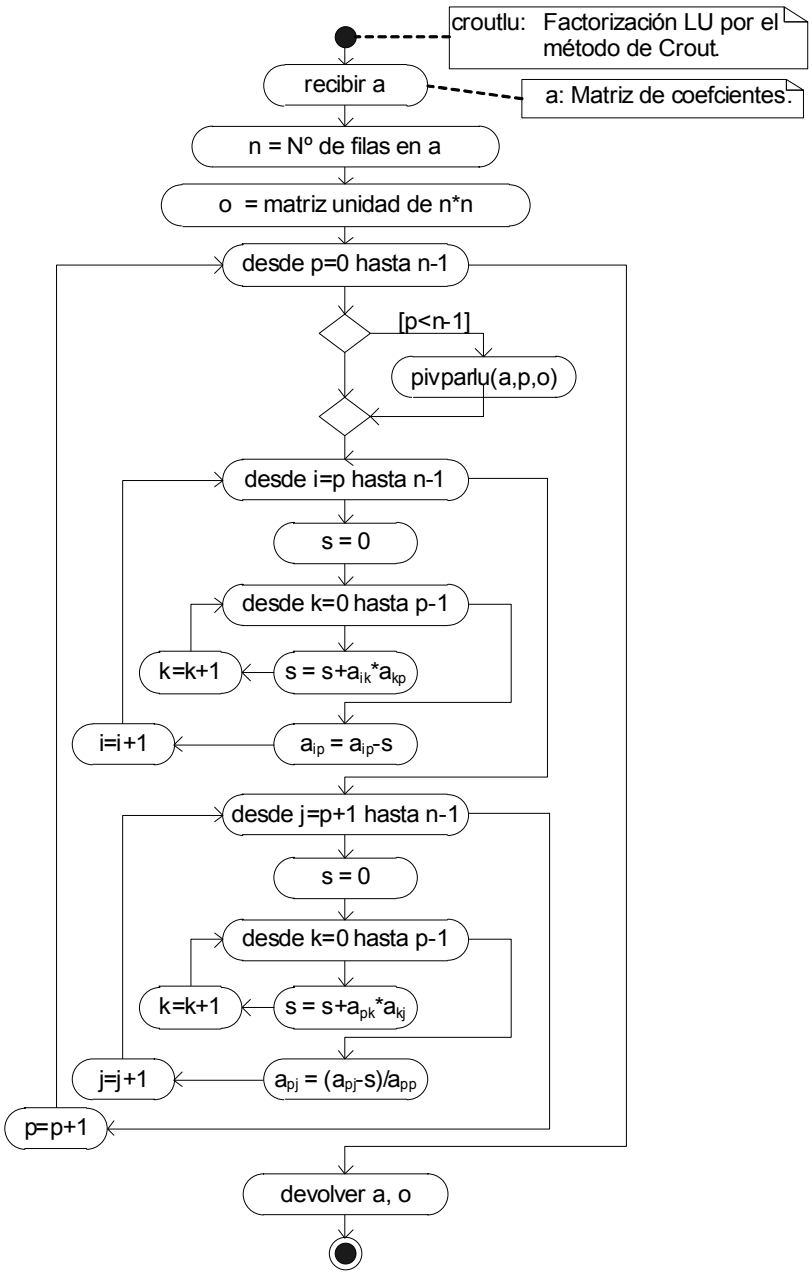
Simplemente para demostrar que la matriz de permutaciones "o" permite reproducir los intercambios de filas realizados, se multiplica la matriz de permutaciones por la matriz original:

```
>>mul(o,[[0,2,1],[1,0,0],[3,0,1]])
[[3, 0, 1], [0, 2, 1], [1, 0, 0]]
```

Y como se puede ver, se obtiene la matriz resultante de los intercambios.

6.3.2. Factorización LU

El algoritmo para factorizar (descomponer) una matriz por el método de Crout y el código respectivo añadido a la librería "mat205.js", se presenta en la siguiente página.




```
function croutlu(b) {
  var i,j,k,p,s,a=mCopym(b),n=a.length,o=new Array(n);
  for (i=0;i<n;i++) {
    o[i]=new Array(n);
    for (j=0;j<n;j++) o[i][j]= i==j ? 1 : 0;}
  for (p=0;p<n;p++) {
    if (p<n-1) pivparlu(a,p,o);
    for (i=p;i<n;i++) {
      for (s=0,k=0;k<p;k++) s+=a[i][k]*a[k][p];
      a[i][p]-=s;}
    for (j=p+1;j<n;j++) {
      for (s=0,k=0;k<p;k++) s+=a[p][k]*a[k][j];
```

```
        a[p][j]=(a[p][j]-s)/a[p][p];}
    }
    return [a,o];
}
```

Haciendo correr el programa con los datos del ejemplo manual se obtienen los mismos resultados que en dicho ejemplo:

```
>>a=[[2,8,2],[1,6,-1],[2,-1,2]];r=croutlu(a)
[[[2, 4, 1], [1, 2, -1], [2, -9, -9]], [[1, 0, 0], [0, 1, 0], [0, 0, 1]]]
```

Y con la matriz de coeficientes del sistema (6.36):

```
>>a=[[0,2,1],[1,0,0],[3,0,1]]; r=croutlu(a)
[[[3, 0, 0.3333333333333333], [0, 2, 0.5], [1, 0, -0.3333333333333333]],
[[0, 0, 1], [1, 0, 0], [0, 1, 0]]]
```

6.3.3. Cálculo de las soluciones

Una vez que se tiene la matriz factorizada y la matriz de permutaciones, se pueden calcular los resultados aplicando los procesos de sustitución hacia adelante y hacia atrás (ecuaciones (6.33) y (6.34)).

El algoritmo se presenta en la siguiente página y el código respectivo, añadido a la librería "mat205.js", es:

```
function croutsol(r,c) {
  var i,j,k,s,a=r[0],o=r[1],n=a.length,m=c[0].length,b;
  b=mul(o,c);
  for (i=0;i<n;i++)
    for (j=0;j<m;j++) {
      for (s=0,k=0;k<i;k++) s+=a[i][k]*b[k][j];
      b[i][j]=(b[i][j]-s)/a[i][i];}
  for (i=n-1;i>=0;i--)
    for (j=0;j<m;j++) {
      for (s=0,k=i+1;k<n;k++) s+=a[i][k]*b[k][j];
      b[i][j]-=s;}
  return b;
}
```

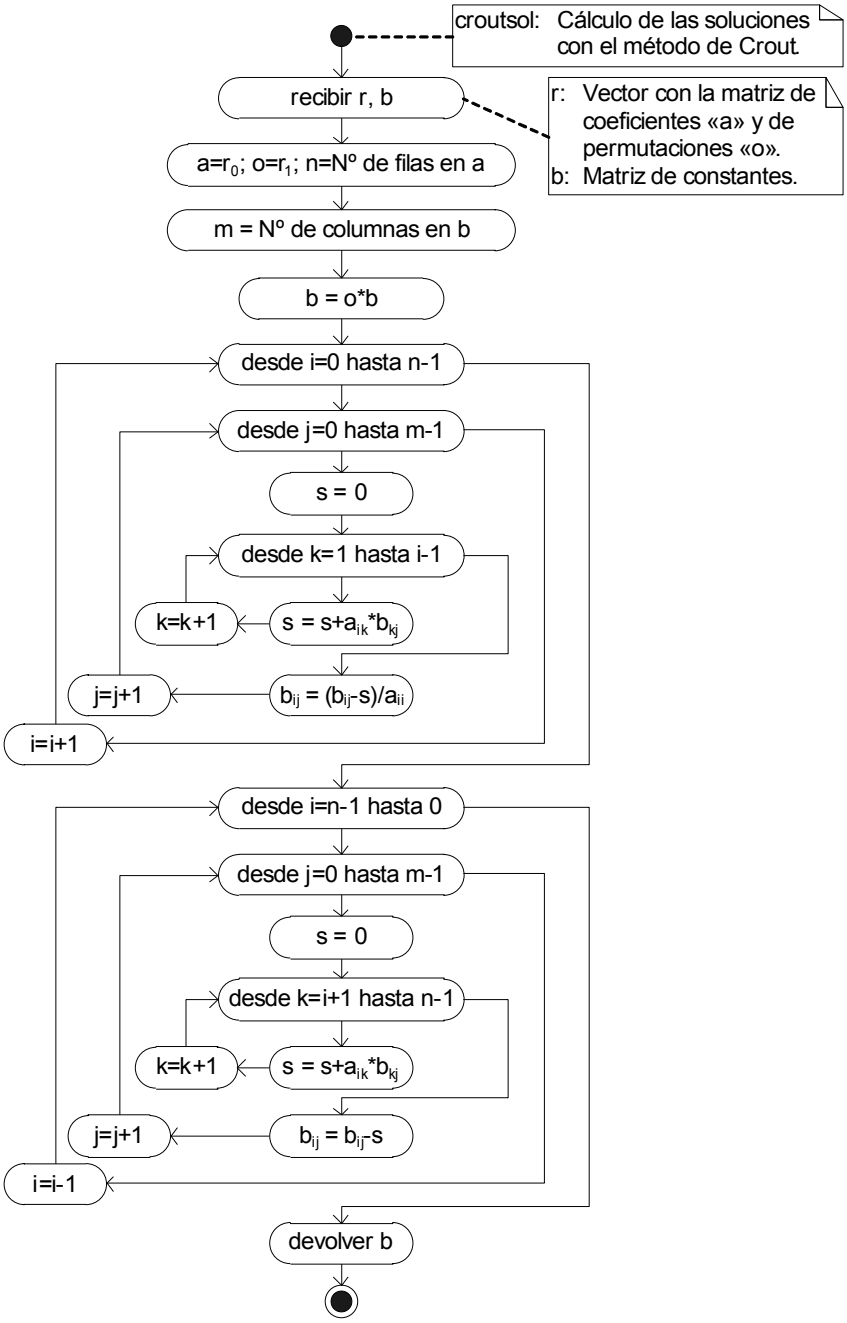
Haciendo correr el programa con los datos del ejemplo manual se obtiene:

```
>>a=[[2,8,2],[1,6,-1],[2,-1,2]]; b=[[14],[13],[5]]; r=croutlu(a);
croutsol(r,b)
[[5], [1], [-2]]
```

Que son los resultados correctos, lo mismo sucede con los datos del sistema (6.36):

```
>>a=[[0,2,1],[1,0,0],[3,0,1]]; b=[[5],[-1],[-2]]; r=croutlu(a);
croutsol(r,b)
[[-1], [2], [1.0000000000000002]]
```

Si el método se emplea principalmente para resolver sistemas de ecuaciones lineales independientes, en lugar de sistemas de ecuaciones lineales relacionados (los cuales difieren únicamente en la columna de las constantes) resulta más práctico agrupar las dos funciones en una sola. Dicha función, que ha sido creada y añadida a la librería "mat205.js", tiene el siguiente código:




```
function crout(a,b){
  return croutsol(croutlu(a),b);
}
```

Haciendo correr el programa con los anteriores sistemas, se obtiene, como era de esperar, las mismas soluciones, pero con una instrucción:

```
>>a=[[2,8,2],[1,6,-1],[2,-1,2]]; b=[[14],[13],[5]]; r=crout(a,b)
[[5], [1], [-2]]
>>a=[[0,2,1],[1,0,0],[3,0,1]]; b=[[5],[-1],[-2]]; crout(a,b)
[[-1], [2], [1.0000000000000002]]
```

Sin embargo, como se dijo, la principal ventaja de los métodos de facto-

rización es la de permitir resolver sistemas de ecuaciones lineales que sólo cambian en las constantes. Por ejemplo, para resolver los siguientes sistemas de ecuaciones:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 21 \\ 5x_1 - 9x_2 + 2x_3 &= 12 \\ 3x_1 + x_2 - 4x_3 &= 15 \end{aligned}$$

$$\begin{aligned} y_1 + 2y_2 + 3y_3 &= 11 \\ 5y_1 - 9y_2 + 2y_3 &= 2 \\ 3y_1 + y_2 - 4y_3 &= 4 \end{aligned} \qquad (637)$$

$$\begin{aligned} z_1 + 2z_2 + 3z_3 &= 1 \\ 5z_1 - 9z_2 + 2z_3 &= 2 \\ 3z_1 + z_2 - 4z_3 &= 3 \end{aligned}$$

Se factoriza (una sola vez) la matriz de coeficientes:

```
>>a=[[1,2,3],[5,-9,2],[3,1,-4]]; r=croutlu(a)
[[[5, -1.8, 0.4], [1, 3.8, 0.6842105263157895], [3, 6.4,
-9.578947368421053]], [[0, 1, 0], [1, 0, 0], [0, 0, 1]]]
```

Entonces con la matriz factorizada (y la matriz de permutaciones guardadas en "r"), se calculan las soluciones del primer sistema:

```
>>b=[[21],[12],[15]]; croutsol(r,b)
[[7.203296703296704], [3.2142857142857144], [2.456043956043956]]
```

Es decir que x_1=7.203296703296704, x_2= 3.2142857142857144 y x_3=2.456043 956043956. Se procede de forma similar para el segundo sistema:

```
>>b=[[11],[2],[4]]; croutsol(r,b)
[[2.857142857142857], [1.7142857142857144], [1.5714285714285716]]
```

Por lo tanto: y_1=2.857142857142857, y_2=1.7142857142857144 y y_3=1.57142857142 85716. Y para el tercero:

```
>>b=[[1],[2],[3]]; croutsol(r,b)
[[0.8186813186813187], [0.21428571428571425], [-0.08241758241758239]]
```

Es decir que: z_1=0.8186813186813187, z_2=0.21428571428571425 y z_3=-0.08241758 241758239.

Es posible, también, resolver los tres sistemas simultáneamente si se crea una matriz con las constantes de los tres sistemas:

```
>>b=[[21,11,1],[12,2,2],[15,4,3]]; croutsol(r,b)
[[7.203296703296704, 2.857142857142857, 0.8186813186813187],
[3.2142857142857144, 1.7142857142857144, 0.21428571428571425],
[2.456043956043956, 1.5714285714285716, -0.08241758241758239]]
```

Donde las soluciones del primer sistema se encuentran en la primera columna, las del segundo en la segunda y las del tercero en la tercera.

Los métodos de eliminación permiten resolver también (simultáneamente) sistemas de ecuaciones que sólo difieren en las constantes. Para ello se crea la matriz aumentada con las columnas de constantes de todos los sistemas. Así para resolver los sistemas (6.36) con el método de Gauss-Jordán se escribe:

```
>>a=[[1,2,3,21,11,1],[5,-9,2,12,2,2],[3,1,-4,15,4,3]]; gaussj(a)
[[7.2032967032967035, 2.857142857142857, 0.8186813186813187],
[3.2142857142857144, 1.7142857142857142, 0.21428571428571427],
[2.456043956043956, 1.5714285714285714, -0.08241758241758236]]
```

Y como se puede ver se obtienen los mismos resultados (despreciando los errores de redondeo característicos de los cálculos numéricos). Sin embargo, si se emplean los métodos de eliminación para resolver cada sistema por separado, se repiten las mismas operaciones para cada sistema, lo que representa una pérdida de tiempo. Esto se puede evitar si se calcula la inversa de la matriz de coeficientes:

```
>>a=[[1,2,3],[5,-9,2],[3,1,-4]]; c=inv(a)
[[0.18681318681318682, 0.06043956043956045, 0.17032967032967034],
[0.14285714285714288, -0.07142857142857144, 0.07142857142857142],
[0.17582417582417584, 0.027472527472527472, -0.1043956043956044]]
```

Y luego, se emplea esta matriz para resolver los tres sistemas (uno por uno) multiplicándola por la columna de constantes:

```
>>b=[[21],[12],[15]]; mul(c,b)
[[7.203296703296704], [3.2142857142857144], [2.456043956043956]]
>>b=[[1],[2],[3]]; mul(c,b)
[[0.8186813186813187], [0.21428571428571427], [-0.08241758241758241]]
>>b=[[21],[12],[15]]; mul(c,b)
[[7.203296703296704], [3.2142857142857144], [2.456043956043956]]
```

Por supuesto, con la matriz inversa, es posible también resolver los tres sistemas a la vez:

```
>>b=[[21,11,1],[12,2,2],[15,4,3]]; mul(c,b)
[[7.203296703296704, 2.8571428571428568, 0.8186813186813187],
[3.2142857142857144, 1.7142857142857144, 0.21428571428571427],
[2.456043956043956, 1.5714285714285716, -0.08241758241758241]]
```

6.4. MÉTODO DE DOOLITTLE

Como ya se dijo, cuando en la factorización L-U, la matriz triangular inferior queda con unos en la diagonal principal, el proceso se conoce como el método de Doolittle (ecuación (6.15)).

Las ecuaciones para el cálculo de las matrices "L" y "U" son:

$$\left.\begin{array}{l} u_{pj} = a_{pj} - \sum_{k=1}^{p-1} l_{pk} \cdot u_{kj} \quad \{ j = p \rightarrow n \\ l_{pp} = 1 \\ l_{ip} = \dfrac{a_{ip} - \sum_{k=1}^{p-1} l_{ik} \cdot u_{kp}}{u_{pp}} \quad \{ i = p+1 \rightarrow n \end{array}\right\} \; p = 1 \rightarrow n \qquad (6.38)$$

Y se deducen de manera similar al método de Crout, sólo que ahora los elementos de la matriz original "A" se calculan en el orden: fila 1, columna 1, fila 2, columna 2, fila 3, columna 3 y así sucesivamente.

Sin embargo, y al igual que en el método de Crout, no es necesario guardar los ceros ni los unos de las matrices, por lo que las dos matrices ("L"

y "U") pueden ser guardadas en una:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21} & u_{22} & u_{23} & u_{24} \\ l_{31} & l_{32} & u_{33} & u_{34} \\ l_{41} & l_{42} & l_{43} & u_{44} \end{bmatrix} \tag{6.39}$$

Si esta matriz se guarda en la matriz "A", las ecuaciones (6.38) se transforman en:

$$\left.\begin{array}{ll} a_{pj} = a_{pj} - \sum_{k=1}^{p-1} a_{pk} \cdot a_{kj} & \{ j = p \rightarrow n \\ a_{ip} = \dfrac{a_{ip} - \sum_{k=1}^{p-1} a_{ik} \cdot a_{kp}}{a_{pp}} & \{ i = p+1 \rightarrow n \end{array}\right\} \; p = 1 \rightarrow n \tag{6.40}$$

Las ecuaciones para el cálculo de las soluciones, tomando en cuenta que ahora "l" tiene la diagonal con unos y guardando las soluciones en la matriz de constantes "b", son:

$$b_{ij} = b_{ij} - \sum_{k=1}^{i-1} a_{ik} \cdot b_{kj} \quad \begin{cases} i = 1 \rightarrow n \\ j = 1 \rightarrow m \end{cases} \tag{6.41}$$

$$b_{ij} = \frac{b_{ij} - \sum_{k=i+1}^{n} a_{ik} \cdot b_{kj}}{a_{ii}} \quad \begin{cases} i = n \rightarrow 1 \\ j = 1 \rightarrow m \end{cases} \tag{6.42}$$

Para comprender mejor el proceso se aplicará manualmente el método de Doolittle al sistema (6.35), los datos son:

```
>>a=[[2,8,2],[1,6,-1],[2,-1,2]]; b=[[14],[13],[5]]; n=a.length; p=0
0
```

Ahora se aplican las ecuaciones (6.40) para calcular la primera fila y columna de "U" y "L":

```
>>for(j=p;j<n;j++){s=0; for(k=0;k<p;k++) s+=a[p][k]*a[k][j]; a[p][j]-=s;}
for(i=p+1;i<n;i++){s=0; for(k=0;k<p;k++) s+=a[i][k]*a[k][p]; a[i][p]=
(a[i][p]-s)/a[p][p];} a
[[2, 8, 2], [0.5, 6, -1], [1, -1, 2]]
```

Ahora se procede con la segunda fila y columna de "U" y "L" (incrementando "p"):

```
>>p++; for(j=p;j<n;j++){s=0; for(k=0;k<p;k++) s+=a[p][k]*a[k][j]; a[p]
[j]-=s;} for(i=p+1;i<n;i++){s=0; for(k=0;k<p;k++) s+=a[i][k]*a[k][p];
a[i][p]=(a[i][p]-s)/a[p][p];} a
[[2, 8, 2], [0.5, 2, -2], [1, -4.5, 2]]
```

Finalmente se calcula la última fila de "U" (y la última columna de "L"):

```
>>p++; for(j=p;j<n;j++){s=0; for(k=0;k<p;k++) s+=a[p][k]*a[k][j]; a[p]
[j]-=s;} for(i=p+1;i<n;i++){s=0; for(k=0;k<p;k++) s+=a[i][k]*a[k][p];
a[i][p]=(a[i][p]-s)/a[p][p];} a
[[2, 8, 2], [0.5, 2, -2], [1, -4.5, -9]]
```

Entonces se calculan las soluciones aplicando primero la ecuación (6.41):

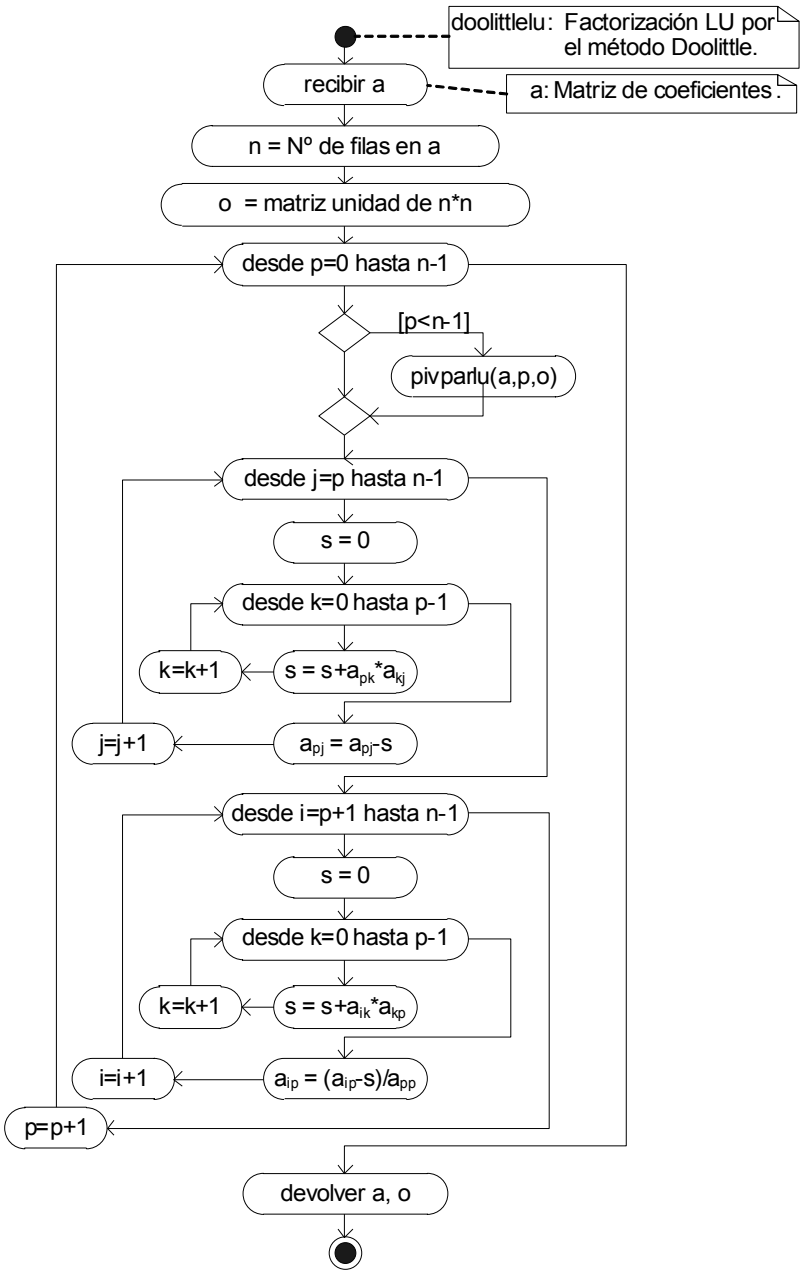
```
>>j=0; for(i=0;i<n;i++){s=0; for(k=0;k<i;k++) s+=a[i][k]*b[k][j]; b[i]
[j]-=s;} b
[[14], [6], [18]]
```

Y luego la ecuación (6.42):

```
>>for(i=n-1;i>=0;i--){s=0; for(k=i+1;k<n;k++) s+=a[i][k]*b[k][j]; b[i]
[j]=(b[i][j]-s)/a[i][i];} b
[[5], [1], [-2]]
```

6.4.1. Fatorización LU

El algoritmo para calcular las matrices "L" y "U" por el método de Doolittle es el siguiente:



Y el código, añadido a la librería "mat205.js", es:

```
function doolittlelu(b) {
  var i,j,k,p,s,a=mCopym(b),n=a.length,o=new Array(n);
  for (i=0;i<n;i++) {
    o[i]=new Array(n);
    for (j=0;j<n;j++) o[i][j]= i==j ? 1 : 0;}
  for (p=0;p<n;p++) {
    if (p<n-1) pivparlu(a,p,o);
    for (j=p;j<n;j++) {
      for (s=0,k=0;k<p;k++) s+=a[p][k]*a[k][j];
      a[p][j]-=s;}
    for (i=p+1;i<n;i++) {
      for (s=0,k=0;k<p;k++) s+=a[i][k]*a[k][p];
      a[i][p]=(a[i][p]-s)/a[p][p];}
  }
  return [a,o];
}
```

Haciendo correr el programa con la matriz del ejemplo manual se obtiene:

```
>>a=[[2,8,2],[1,6,-1],[2,-1,2]]; doolittlelu(a)
[[[2, 8, 2], [0.5, 2, -2], [1, -4.5, -9]], [[1, 0, 0], [0, 1, 0], [0, 0,
1]]]
```

Que son los resultados obtenidos en dicho ejemplo (además de la matriz de permutaciones).

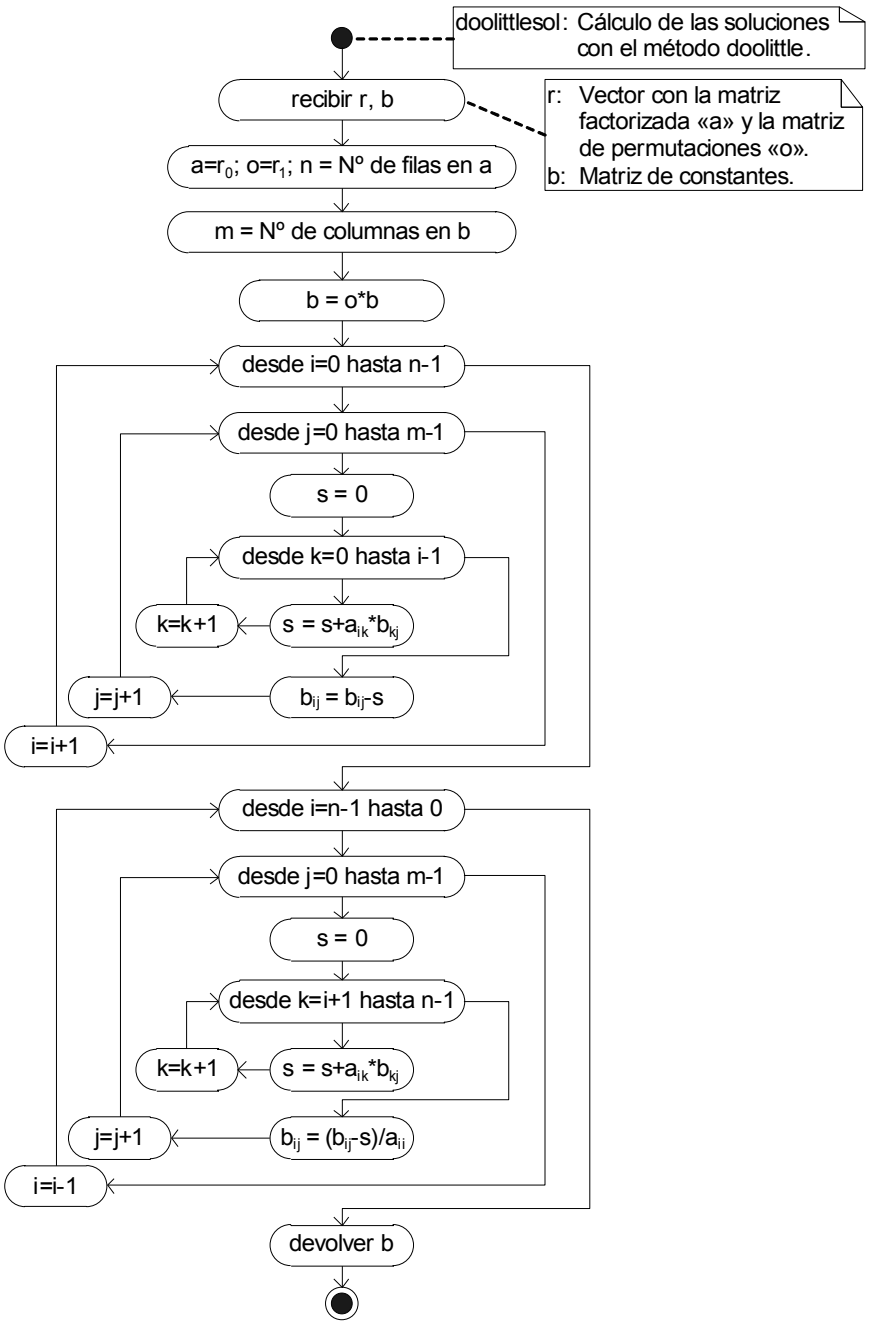
6.4.2. Cálculo de las soluciones

Con las matrices triangulares inferior, superior y la matriz de permutaciones (en una matriz aumentada), se pueden calcular las soluciones del sistema aplicando las ecuaciones (6.41) y (6.42). El algoritmo se presenta en la siguiente página, siendo el código (añadido a "mat205.js"):

```
function doolittlesol(r,c) {
  var i,j,k,s,a=r[0],o=r[1],n=a.length,m=c[0].length,b;
  b=mul(o,c);
  for (i=0;i<n;i++)
    for (j=0;j<m;j++) {
      for (s=0,k=0;k<i;k++) s+=a[i][k]*b[k][j];
      b[i][j]-=s;}
  for (i=n-1;i>=0;i--)
    for (j=0;j<m;j++) {
      for (s=0,k=i+1;k<n;k++) s+=a[i][k]*b[k][j];
      b[i][j]=(b[i][j]-s)/a[i][i];}
  return b;
}
```

Haciendo correr el programa con los datos del ejemplo manual se obtiene:

```
>>a=[[2,8,2],[1,6,-1],[2,-1,2]]; b=[[14],[13],[5]]; r=doolittlelu(a);
doolittlesol(r,b)
[[5], [1], [-2]]
```



Al igual que en el método de Crout, las dos funciones se pueden agrupar en una. Dicha función, añadida a la librería "mat205.js", es:

```
function doolittle(a,b){
  return doolittlesol(doolittlelu(a),b);
}
```

Con la cual, como era de esperar, se obtienen los resultados correctos:

```
>>a=[[2,8,2],[1,6,-1],[2,-1,2]]; b=[[14],[13],[5]]; doolittle(a,b)
[[5], [1], [-2]]
```

El método de Doolittle se emplea de la misma forma y en las mismas condiciones que el método de Crout.

6.5. MÉTODO DE CHOLESKY

Como ya se vio, cuando la matriz es simétrica, la descomposición "L" "U" puede ser simplificada y al proceso se conoce como el método de Cholesky (ecuación (6.16)). Puesto que la matriz a factorizar debe ser simétrica, el método de Cholesky no admite pivotaje.

A pesar de estas limitaciones, la forma de Cholesky resulta de utilidad práctica, pues los sistemas con matrices simétricas aparecen en la resolución de varios problemas prácticos en el campo de la ingeniería.

Las ecuaciones que permiten calcular los valores de "L" (y en consecuencia los de "U"), se deducen obteniendo los términos de la matriz triangular inferior de la matriz original "A" (columna por columna) y son:

$$\left.\begin{aligned} l_{pp} &= \sqrt{a_{pp} - \sum_{k=1}^{p-1} l_{pk}^{\,2}} \\ l_{ip} &= \frac{a_{ip} - \sum_{k=1}^{p-1} l_{ik} \cdot l_{pk}}{l_{pp}} = l_{pi} \quad \{i = p+1 \rightarrow n \end{aligned}\right\} \; p = 1 \rightarrow n \qquad (6.43)$$

Como se puede ver, el cálculo de los elementos de la diagonal principal involucra el cálculo de la raíz cuadrada, por lo que dicho valor debe ser positivo.

Al igual que en los métodos de Crout y Doolittle, no es necesario almacenar los ceros y dado que en este caso la matriz transpuesta y la original tienen los mismos elementos en la diagonal principal, los resultados de la factorización se pueden almacenar en una matriz:

$$\begin{bmatrix} l_{11} & l_{21} & l_{31} & l_{41} \\ l_{21} & l_{22} & l_{32} & l_{42} \\ l_{31} & l_{32} & l_{33} & l_{43} \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \qquad (6.44)$$

Si la matriz donde se guardan estos valores es la matriz "A", las ecuaciones (6.43) pueden ser reescritas como:

$$\left.\begin{aligned} a_{pp} &= \sqrt{a_{pp} - \sum_{k=1}^{p-1} a_{pk}^{\,2}} \\ a_{ip} &= \frac{a_{ip} - \sum_{k=1}^{p-1} a_{ik} \cdot a_{pk}}{a_{pp}} = a_{pi} \quad \{i = p+1 \rightarrow n \end{aligned}\right\} \; p = 1 \rightarrow n \qquad (6.45)$$

Una vez factorizada la matriz de los coeficientes, los resultados pueden ser calculados con las ecuaciones (6.33) y (6.42).

Para comprender mejor el proceso se resolverá el siguiente sistema de ecuaciones con este método:

$$\begin{aligned} 4x_1 + x_2 + 2x_3 &= 5 \\ x_1 + 7x_2 + 3x_3 &= 8 \\ 2x_1 + 3x_2 + 9x_3 &= 3 \end{aligned} \qquad (6.46)$$

Los datos son:

```
>>a=[[4,1,2],[1,7,3],[2,3,9]]; b=[[5],[8],[3]]; n=a.length; p=0;
0
```

Ahora se calcula el primer elemento de la diagonal principal y la primera columna (y por consiguiente la primera fila):

```
>>s=0; for (k=0;k<p;k++) s+=sqr(a[p][k]); a[p][p]=sqrt(a[p][p]-s);
for(i=p+1;i<n;i++){s=0; for(k=0;k<p;k++) s+=a[i][k]*a[p][k]; a[i][p]=
(a[i][p]-s)/a[p][p]; a[p][i]=a[i][p];} a
[[2, 0.5, 1], [0.5, 7, 3], [1, 3, 9]]
```

Se incrementa "p" y se repite el proceso para la segunda columna y fila:

```
>>p++; s=0; for (k=0;k<p;k++) s+=sqr(a[p][k]); a[p][p]=sqrt(a[p][p]-s);
for(i=p+1;i<n;i++){s=0; for(k=0;k<p;k++) s+=a[i][k]*a[p][k]; a[i][p]=
(a[i][p]-s)/a[p][p]; a[p][i]=a[i][p];} a
[[2, 0.5, 1], [0.5, 2.598076211353316, 0.9622504486493763], [1,
0.9622504486493763, 9]]
```

Finalmente se calcula el último valor de la diagonal principal:

```
>>p++; s=0; for (k=0;k<p;k++) s+=sqr(a[p][k]); a[p][p]=sqrt(a[p][p]-s);
for(i=p+1;i<n;i++){s=0; for(k=0;k<p;k++) s+=a[i][k]*a[p][k]; a[i][p]=
(a[i][p]-s)/a[p][p]; a[p][i]=a[i][p];} a
[[2, 0.5, 1], [0.5, 2.598076211353316, 0.9622504486493763], [1,
0.9622504486493763, 2.6597131563524052]]
```

Ahora, con la matriz factorizada, se calculan las soluciones aplicando primero la ecuación (633):

```
>>j=0; for(i=0;i<n;i++){s=0; for(k=0;k<i;k++) s+=a[i][k]*b[k][j]; b[i]
[j]=(b[i][j]-s)/a[i][i];} b
[[2.5], [2.598076211353316], [-0.7519607876598423]]
```

Y luego la ecuación (642):

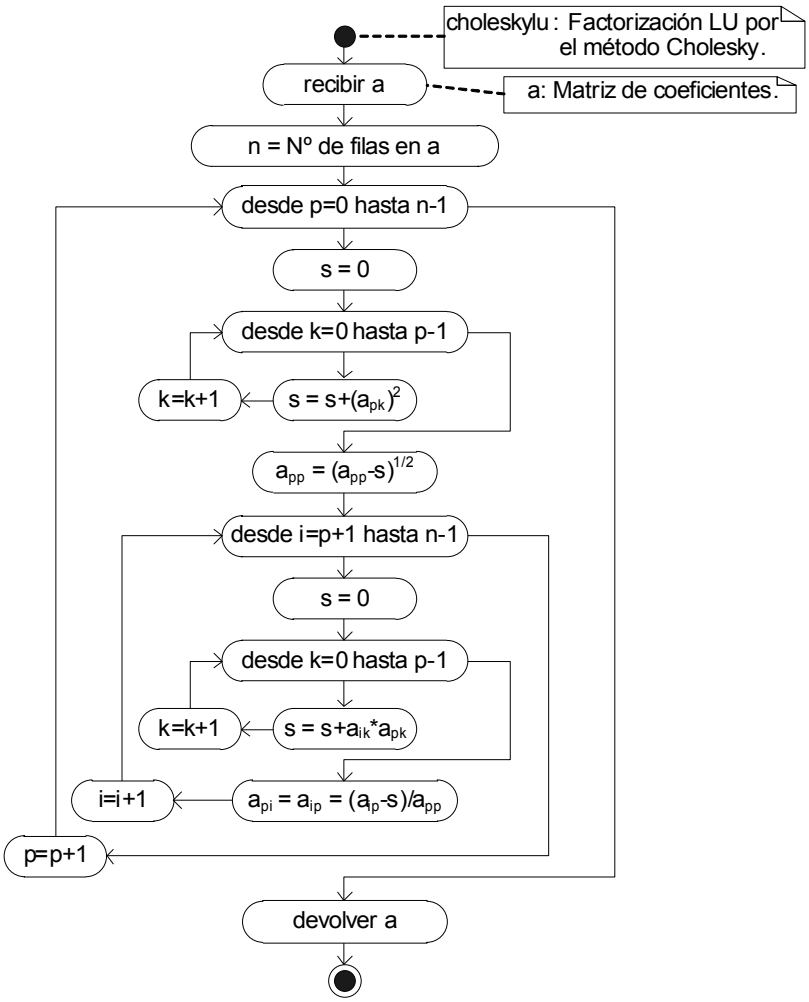
```
>>for(i=n-1;i>=0;i--){s=0; for(k=i+1;k<n;k++) s+=a[i][k]*b[k][j]; b[i]
[j]=(b[i][j]-s)/a[i][i];} b
[[1.1151832460732984], [1.1047120418848166], [-0.28272251308900526]]
```

Que son las soluciones del sistema: x_1=1.1151832460732984, x2=1.1047120418848166, x_3=-0.28272251308900526.

6.5.1. Factorización LU

El algoritmo que automatiza el proceso de factorización se presenta en la siguiente página, siendo el código respectivo (añadido a "mat205.js"):

```
function choleskylu(b) {
  var i,j,k,p,s,a=mCopym(b),n=a.length;
  for (p=0;p<n;p++) {
    for (s=0,k=0;k<p;k++) s+=sqr(a[p][k]);
    a[p][p]=sqrt(a[p][p]-s);
    for (i=p+1;i<n;i++) {
      for (s=0,k=0;k<p;k++) s+=a[i][k]*a[p][k];
      a[i][p]=(a[i][p]-s)/a[p][p];a[p][i]=a[i][p];}
  }
```




```
  return a;
}
```

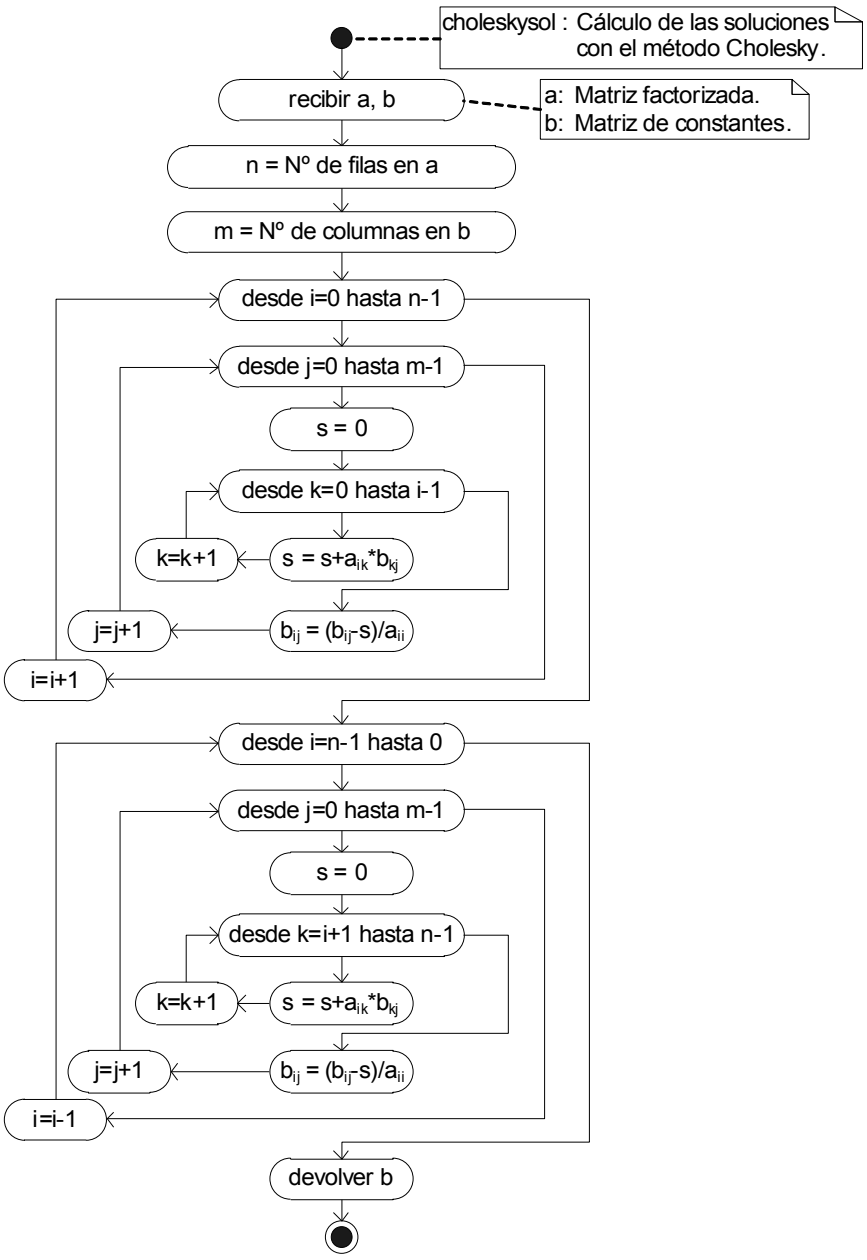
Haciendo correr el programa con la matriz del ejemplo manual se obtienen los mismos resultados que en dicho ejemplo:

```
>>a=[[4,1,2],[1,7,3],[2,3,9]]; choleskylu(a)
[[2, 0.5, 1], [0.5, 2.598076211353316, 0.9622504486493763], [1,
0.9622504486493763, 2.6597131563524052]]
```

6.5.2. Cálculo de las soluciones

Una vez factorizada la matriz de coeficientes se pueden calcular las soluciones. El algoritmo que automatiza el proceso se presenta en la siguiente página, siendo el código respectivo (añadido a "mat205.js"):

```
function choleskysol(a,c) {
  var i,j,k,s,n=a.length,b=mCopym(c),m=b[0].length;
  for (i=0;i<n;i++)
    for (j=0;j<m;j++) {
      for (s=0,k=0;k<i;k++) s+=a[i][k]*b[k][j];
      b[i][j]=(b[i][j]-s)/a[i][i];}
  for (i=n-1;i>=0;i--)
```




```
    for (j=0;j<m;j++) {
      for (s=0,k=i+1;k<n;k++) s+=a[i][k]*b[k][j];
      b[i][j]=(b[i][j]-s)/a[i][i];}
  return b;
}
```

Haciendo correr el programa con los datos del ejemplo manual se obtienen los mismos resultados:

```
>>a=[[4,1,2],[1,7,3],[2,3,9]]; b=[[5],[8],[3]]; r=choleskylu(a);
choleskysol(r,b)
[[1.1151832460732984], [1.1047120418848166], [-0.28272251308900526]]
```

Al igual que en los anteriores métodos se pueden agrupar las dos funciones en una. Dicha función, añadida a la librería "mat205.js", es:

```
function cholesky(a,b){
  return choleskysol(choleskylu(a),b);
}
```

Haciendo correr el programa con los datos del ejemplo, se obtiene, como era de esperar, los mismos resultados:

```
>>a=[[4,1,2],[1,7,3],[2,3,9]]; b=[[5],[8],[3]]; cholesky(a,b)
[[1.1151832460732984], [1.1047120418848166], [-0.28272251308900526]]
```

6.6. MÉTODO DE JACOBI

Cuando el número de ecuaciones en un sistema es elevado (con cientos o miles de ecuaciones) los métodos directos, estudiados en temas anteriores, no son de utilidad práctica y ello se debe a que en dichos métodos los resultados intermedios se emplean para calcular otros y estos a su vez para calcular otros y así sucesivamente hasta llegar a los resultados finales. Esto da origen a la propagación del error, donde los errores de redondeo se van acumulando en cada nuevo cálculo y cuando el número de ecuaciones es muy elevado el error acumulado es tan grande que los resultados son de hecho erróneos.

En estos casos se debe recurrir a métodos iterativos, en los cuales, partiendo de valores iniciales asumidos se calculan, en iteraciones consecutivas, nuevos valores hasta que las igualdades del sistema se cumplen con cierto margen de error. De esta manera se evita la propagación del error y el error permitido se fija de antemano.

Uno de los métodos iterativos más sencillos es el de ***Jacobi*** (que es equivalente al método de sustitución directa pero para sistemas de ecuaciones lineales).

En este método se asumen valores iniciales para todas las incógnitas (generalmente ceros), luego, con estos valores, se calcula la primera incógnita con la primera ecuación, la segunda incógnita con la segunda ecuación y así sucesivamente. Entonces, con los valores calculados, se verifica si se cumplen las igualdades del sistema, de ser así el proceso concluye y se tienen las soluciones buscadas, caso contrario los valores calculados se convierten en valores asumidos y se repite el proceso.

Para comprender mejor el método se resolverá el siguiente sistema de ecuaciones lineales:

$$\begin{aligned} 10x_1 &+ x_2 &+ 2x_3 &= 44 \\ 2x_1 &+ 10x_2 &+ x_3 &= 51 \\ x_1 &+ 2x_2 &+ 10x_3 &= 61 \end{aligned} \qquad (6.47)$$

Se asumen valores iniciales iguales a cero. Con estos ceros, se calculan los nuevos valores, a los cuales se denominarán "y", empleando la primera ecuación para calcular el primer valor (y_1), la segunda para el segundo y la tercera para el tercero:

$$y_1 = \frac{44 - x_2 - 2x_3}{10} = \frac{44 - 0 - 2*0}{10} = 4.4$$

$$y_2 = \frac{51 - 2x_1 - x_3}{10} = \frac{51 - 2*0 - 0}{10} = 5.1$$

$$y_3 = \frac{61 - x_1 - 2x_2}{10} = \frac{61 - 0 - 2*0}{10} = 6.1$$

Como los valores iniciales son diferentes a los valores asumidos (ceros) el proceso se repite, empleando los valores calculados como nuevos valores de prueba, es decir: $x_1=y_1=4.4$, $x_2=y_2=5.1$ y $x_3=y_3=6.1$:

$$y_1=\frac{44-x_2-2x_3}{10}=\frac{44-5.1-2*6.1}{10}=2.67$$
$$y_2=\frac{51-2x_1-x_3}{10}=\frac{51-2*4.4-6.1}{10}=3.61$$
$$y_3=\frac{61-x_1-2x_2}{10}=\frac{61-4.4-2*5.1}{10}=4.64$$

Una vez más los valores asumidos y calculados son diferentes, por lo que es necesario repetir el proceso (empleando nuevamente los valores calculados como nuevos valores de prueba):

$$y_1=\frac{44-x_2-2x_3}{10}=\frac{44-3.61-2*4.64}{10}=3.111$$
$$y_2=\frac{51-2x_1-x_3}{10}=\frac{51-2*2.67-4.64}{10}=4.102$$
$$y_3=\frac{61-x_1-2x_2}{10}=\frac{61-2.67-2*3.61}{10}=5.111$$

Siendo necesario volver a repetir el proceso:

$$y_1=\frac{44-x_2-2x_3}{10}=\frac{44-4.102-2*5.111}{10}=2.9676$$
$$y_2=\frac{51-2x_1-x_3}{10}=\frac{51-2*3.111-5.111}{10}=3.9667$$
$$y_3=\frac{61-x_1-2x_2}{10}=\frac{61-3.111-2*4.102}{10}=4.9685$$

Ahora los valores asumidos y calculados comienzan ha acercarse. Repitiendo el proceso dos veces más se obtiene:

$$y_1=\frac{44-x_2-2x_3}{10}=\frac{44-3.9667-2*4.9685}{10}=3.00963$$
$$y_2=\frac{51-2x_1-x_3}{10}=\frac{51-2*2.9676-4.9685}{10}=4.00963$$
$$y_3=\frac{61-x_1-2x_2}{10}=\frac{61-2.9676-2*3.9667}{10}=5.0099$$

$$y_1=\frac{44-x_2-2x_3}{10}=\frac{44-4.00963-2*5.0099}{10}=2.997057$$
$$y_2=\frac{51-2x_1-x_3}{10}=\frac{51-2*3.00963-5.0099}{10}=3.997084$$
$$y_3=\frac{61-x_1-2x_2}{10}=\frac{61-3.00963-2*4.00963}{10}=4.997111$$

Que ya son iguales en los 3 primeros dígitos, por lo que, con esa precisión, los resultados son: $x_1=3.00$, $x_2=4.00$ y $x_3=5.00$.

6.6.1. Algoritmo y código

Como se hace evidente en el ejemplo manual, la principal desventaja del método de Jacobi es el elevado número de iteraciones que requiere, por lo que sólo resulta de utilidad práctica si el proceso se automatiza mediante la elaboración de un programa.

Como ocurre con los métodos iterativos, la convergencia no está garantizada, por lo que es necesario detener el proceso si no se logra convergencia en un determinado número de iteraciones. En este método en particular, **la convergencia está asegurada si el sistema tienen una *diagonal dominante*,** es decir si los valores absolutos de los elementos de la diagonal principal son mayores a la suma de los valores absolutos de los otros coeficientes de la misma fila (tal como ocurre en el ejemplo manual).

En ocasiones es necesario reordenar el sistema (intercambiando filas) de manera que quede con una diagonal dominante, pero en otras, el sistema simplemente no tiene una diagonal dominante y entonces la convergencia no está asegurada. Afortunadamente, varios de los sistemas que aparecen en la resolución de problemas prácticos tienen una diagonal dominante.

La ecuación general para el cálculo de los nuevos valores de "x", deducida analizando las ecuaciones del ejemplo manual, es:

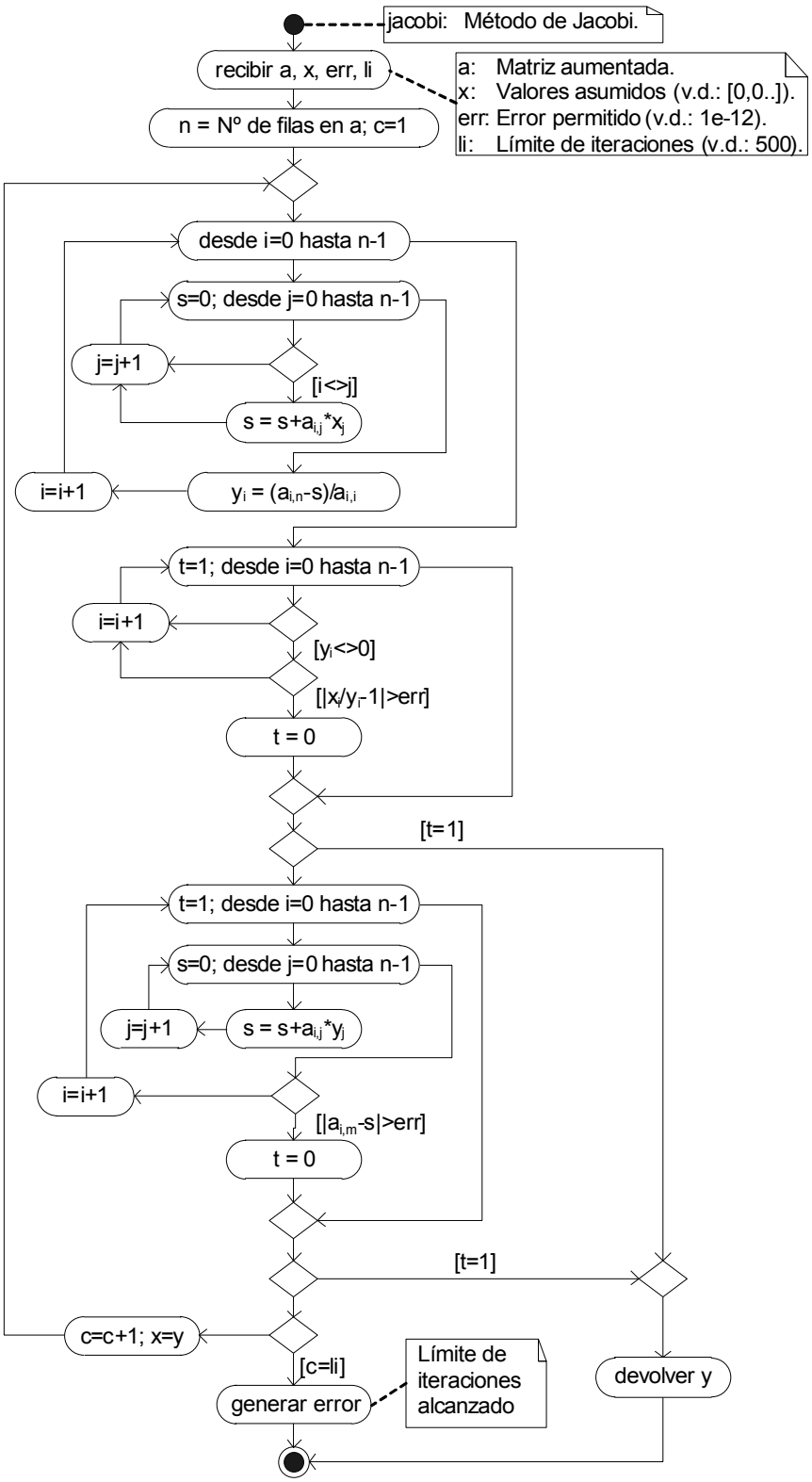
$$x_i = \frac{a_{i,m} - \sum_{j=1}^{n} a_{i,j} \cdot y_j \{ j \neq i}{} \quad \{ i = 1 \rightarrow n \qquad (6.48)$$

Donde "a" es la matriz aumentada del sistema de ecuaciones lineales, "n" es el número de filas y "m" el número de columnas.

Con esta ecuación se calculan los nuevos valores de "x" hasta que son aproximadamente iguales a los asumidos ("y") o hasta que con estos valores las ecuaciones del sistema son aproximadamente iguales a cero.

El algoritmo que automatiza el proceso (tomando en cuenta, como de costumbre, que en Javascript el primer índice es 0) se presenta en la siguiente página y el código respectivo, añadido a la librería "mat205.js", es:

```
function jacobi(a,x,err,li) {
  var i,j,aux,t,s,n=a.length,c=1,y=new Array(n);
  if (x==null) {x=new Array(n); for(i=0;i<n;i++) x[i]=0;}
  if (err==null) err=1e-12;
  if (li==null) li=500;
  while (true) {
    for (i=0;i<n;i++) {
      for (s=0,j=0;j<n;j++)
        s+= i!=j ? a[i][j]*x[j] : 0;
      y[i]=(a[i][n]-s)/a[i][i];}
    for (t=1,i=0;i<n;i++)
      if (y[i]) {if (abs(x[i]/y[i]-1)>err) {t=0; break;}}
    if (t) return y;
    for (t=1,i=0;i<n;i++) {
      for (s=0,j=0;j<n;j++)
        s+=a[i][j]*y[j];
      if (abs(a[i][n]-s)>err) {t=0; break;}};
    if (t) return y;
```




```
  if (c++ == li)
    throw new Error("Convergencia no lograda (jacobi).");
  aux=x;x=y;y=aux;
}
```

```
}
```

Haciendo correr el programa con el sistema del ejemplo manual (con los valores iniciales, error y límite de iteraciones por defecto) se obtiene:

```
>>a=[[10,1,2,44],[2,10,1,51],[1,2,10,61]]; jacobi(a)
[3.000000000000339, 4.000000000000339, 5.000000000000339]
```

Que con los 12 dígitos (que es la precisión por defecto), es:

```
>>precision(jacobi(a),12)
[3, 4, 5]
```

6.7. MÉTODO DE GAUSS-SEIDEL

El método de Gauss – Seidel es muy similar al método de Jacobi, prácticamente la única diferencia es que los nuevos valores se calculan siempre empleando los últimos valores disponibles, es decir si existen nuevos valores calculados se emplean los mismos, caso contrario se emplean los asumidos. Así la primera incógnita se calcula con los valores asumidos, pero la segunda se calcula con los valores asumidos y el valor calculado, la tercera con los valores asumidos y los dos valores calculados y así sucesivamente.

Para comprender mejor el método se resolverá manualmente el sistema de ecuaciones (6.47) y para ello se comienza guardando los datos (siendo los valores iniciales asumidos ceros)

```
>>a=[[10,1,2,44],[2,10,1,51],[1,2,10,61]]; x=[0,0,0]; y=new Array(3);
n=a.length; m=a[0].length
4
```

Y con estos valores se calculan los tres nuevos valores (en el vector y):

```
>>y[0]=(a[0][n]-(a[0][1]*x[1]+a[0][2]*x[2]))/a[0][0]
4.4
>>y[1]=(a[1][n]-(a[1][0]*y[0]+a[1][2]*x[2]))/a[0][0]
4.220000000000001
>>y[2]=(a[2][n]-(a[2][0]*y[0]+a[2][1]*y[1]))/a[0][0]
4.816
```

Como los valores asumidos (ceros) no son iguales a los calculados, se repite el proceso, realizando previamente un cambio de variables para que los valores calculados "y" sean ahora los valores asumidos "x":

```
>>aux=x; x=y; y=aux; y[0]=(a[0][n]-(a[0][1]*x[1]+a[0][2]*x[2]))/a[0][0]
3.0148
>>y[1]=(a[1][n]-(a[1][0]*y[0]+a[1][2]*x[2]))/a[0][0]
4.01544
>>y[2]=(a[2][n]-(a[2][0]*y[0]+a[2][1]*y[1]))/a[0][0]
4.995431999999999
```

Como se pude ver, aún en esta primera iteración, los valores asumidos y calculados se aproximan, lo que demuestra que el método de Gauss-Seidel converge más rápidamente que el método de Jacobi.

Continuando de esta manera hasta conseguir la igualdad se obtiene:

```
>>aux=x; x=y; y=aux; y[0]=(a[0][n]-(a[0][1]*x[1]+a[0][2]*x[2]))/a[0][0]
2.9993696
>>y[1]=(a[1][n]-(a[1][0]*y[0]+a[1][2]*x[2]))/a[0][0]
4.0005828800000005
```

```
>>y[2]=(a[2][n]-(a[2][0]*y[0]+a[2][1]*y[1]))/a[0][0]
4.999946464

>>aux=x; x=y; y=aux; y[0]=(a[0][n]-(a[0][1]*x[1]+a[0][2]*x[2]))/a[0][0]
2.9999524192
>>y[1]=(a[1][n]-(a[1][0]*y[0]+a[1][2]*x[2]))/a[0][0]
4.000014869759999
>>y[2]=(a[2][n]-(a[2][0]*y[0]+a[2][1]*y[1]))/a[0][0]
5.000001784128

>>aux=x; x=y; y=aux; y[0]=(a[0][n]-(a[0][1]*x[1]+a[0][2]*x[2]))/a[0][0]
2.9999981561984
>>y[1]=(a[1][n]-(a[1][0]*y[0]+a[1][2]*x[2]))/a[0][0]
4.00000019034752
>>y[2]=(a[2][n]-(a[2][0]*y[0]+a[2][1]*y[1]))/a[0][0]
5.000000146310656

>>aux=x; x=y; y=aux; y[0]=(a[0][n]-(a[0][1]*x[1]+a[0][2]*x[2]))/a[0][0]
2.9999999517031166
>>y[1]=(a[1][n]-(a[1][0]*y[0]+a[1][2]*x[2]))/a[0][0]
3.999999995028311
>>y[2]=(a[2][n]-(a[2][0]*y[0]+a[2][1]*y[1]))/a[0][0]
5.000000005824026

>>aux=x; x=y; y=aux; y[0]=(a[0][n]-(a[0][1]*x[1]+a[0][2]*x[2]))/a[0][0]
2.999999999332364
>>y[1]=(a[1][n]-(a[1][0]*y[0]+a[1][2]*x[2]))/a[0][0]
3.9999999995511244
>>y[2]=(a[2][n]-(a[2][0]*y[0]+a[2][1]*y[1]))/a[0][0]
5.000000000156539

>>aux=x; x=y; y=aux; y[0]=(a[0][n]-(a[0][1]*x[1]+a[0][2]*x[2]))/a[0][0]
3.00000000001358
>>y[1]=(a[1][n]-(a[1][0]*y[0]+a[1][2]*x[2]))/a[0][0]
3.9999999999816302
>>y[2]=(a[2][n]-(a[2][0]*y[0]+a[2][1]*y[1]))/a[0][0]
5.000000000002316

>>aux=x; x=y; y=aux; y[0]=(a[0][n]-(a[0][1]*x[1]+a[0][2]*x[2]))/a[0][0]
3.0000000000013736
>>y[1]=(a[1][n]-(a[1][0]*y[0]+a[1][2]*x[2]))/a[0][0]
3.999999999999494
>>y[2]=(a[2][n]-(a[2][0]*y[0]+a[2][1]*y[1]))/a[0][0]
4.999999999999964

>>aux=x; x=y; y=aux; y[0]=(a[0][n]-(a[0][1]*x[1]+a[0][2]*x[2]))/a[0][0]
3
>>y[1]=(a[1][n]-(a[1][0]*y[0]+a[1][2]*x[2]))/a[0][0]
4
>>y[2]=(a[2][n]-(a[2][0]*y[0]+a[2][1]*y[1]))/a[0][0]
5
```

Que son los resultados exactos. Si bien el método de Gauss-Seidel requiere menos iteraciones que el de Jacobi, este último puede ser programado en paralelo, por lo que en computadoras con dos o más núcleos (como las actuales) puede llegar a ser más eficiente que el de Gauss-Seidel.

6.7.1. Algoritmo y código

La ecuación general para calcular los nuevos valores con el método de *Gauss-Seidel*, puede ser deducida fácilmente del ejemplo manual, y es:

$$x_i = \frac{a_{i,m} - \sum_{j=1}^{i-1} a_{i,j} \cdot x_j - \sum_{j=i+1}^{n} a_{i,j} \cdot y_j}{a_{i,i}} \quad \{i = 1 \rightarrow n \tag{6.49}$$

Donde "a" es la matriz aumentada, "n" el número de filas, "m" el de columnas, "y" los valores asumidos y "x" los valores calculados (como de costumbre, al momento de programar, los límites deben ser cambiados para tomar en cuenta que en Javascript el primer índice es 0 y no 1).

Puesto que en el método de Gauss-Seidel, siempre se emplean los últimos valores calculados, es posible trabajar directamente con el vector "y" (los nuevos valores calculados), conservando el vector "x" simplemente para efectos de comparación. Procediendo así la ecuación (6.49) se convierte en:

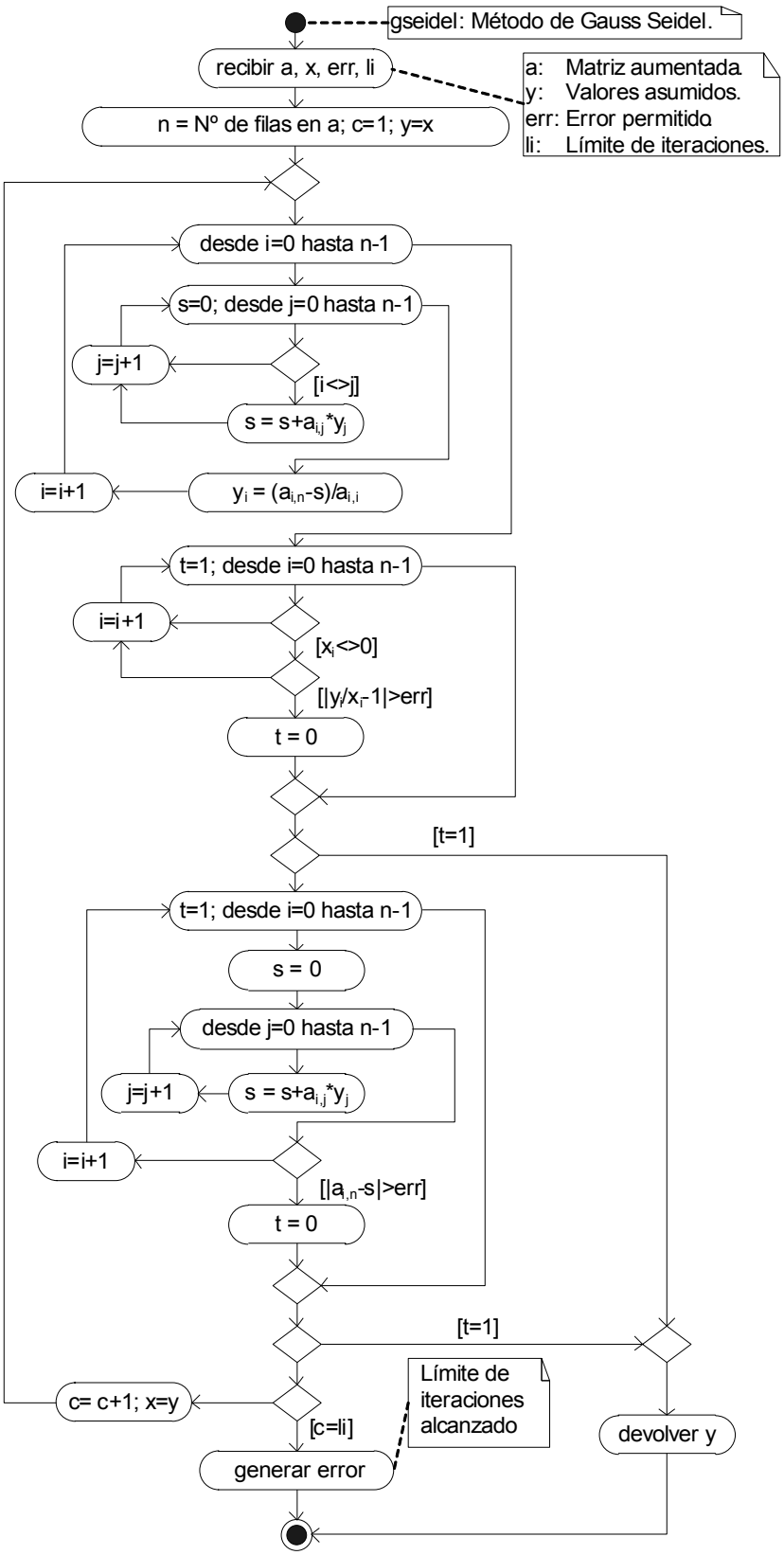
$$y_i = \frac{a_{i,m} - \sum_{j=1}^{n} a_{i,j} \cdot y_j \quad \{j \neq i}{a_{i,i}} \quad \{i = 1 \rightarrow n \tag{6.50}$$

El algoritmo que automatiza el cálculo de esta ecuación se presenta en la siguiente página y el código respectivo (añadido a "mat205.js") es:

```
function gseidel(a,x,err,li) {
  var i,j,aux,t,s,r,n=a.length;
  if (x==null) {x=new Array(n); for(i=0;i<n;i++) x[i]=0;}
  if (err==null) err=1e-12;
  if (li==null) li=500;
  var c=1,y=x.slice();
  while (true) {
    for (i=0;i<n;i++) {
      for (s=0,j=0;j<n;j++)
        s+= i!=j ? a[i][j]*y[j] : 0;
      y[i]=(a[i][n]-s)/a[i][i];}
    for (t=1,i=0;i<n;i++)
      if (y[i]) {if (abs(x[i]/y[i]-1)>err) {t=0; break;}}
    if (t) return y;
    for (t=1,i=0;i<n;i++) {
      for (s=0,j=0;j<n;j++) s+=a[i][j]*y[j];
      if (abs(a[i][n]-s)>err) {t=0; break;}};
    if (t) return y;
    if (c++ == li)
      throw new Error("Convergencia no lograda (gaussSeidel).");
    for (i=0;i<n;i++) x[i]=y[i];
  }
}
```

Haciendo correr el programa con los datos del ejemplo manual (valores iniciales, error y límite de iteraciones por defecto) se obtiene:

```
>>a=[[10,1,2,44],[2,10,1,51],[1,2,10,61]];gseidel(a)
[3.0000000000000577, 3.999999999999992, 4.999999999999996]
```



Que con los 12 dígitos de precisión (que es el error por defecto) son:

```
>>precision(ans,12)
```

```
[3, 4, 5]
```

6.8. MÉTODO DE GAUSS PARA SISTEMAS TRIDIAGONALES

Con frecuencia en la resolución de problemas en el campo de la ingeniería se forman sistemas de ecuaciones lineales que sólo tienen valores en la diagonal principal y a ambos lados de la misma, tales sistemas se conocen como *sistemas tridiagonales* y tienen la forma general:

$$\begin{array}{rcrcrcrcrcrcr}
a_{1,2}x_1 & + & a_{1,3}x_2 & + & 0x_3 & + & 0x_4 & +\cdots+ & 0x_{n-1} & + & 0x_n & = & a_{1,m} \\
a_{2,1}x_1 & + & a_{2,2}x_2 & + & a_{2,3}x_3 & + & 0x_4 & +\cdots+ & 0x_{n-1} & + & 0x_n & = & a_{2,m} \\
0x_1 & + & a_{3,2}x_2 & + & a_{3,3}x_3 & + & a_{3,4}x_4 & +\cdots+ & 0x_{n-1} & + & 0x_n & = & a_{3,m} \\
0x_1 & + & 0x_2 & + & a_{4,3}x_3 & + & a_{4,4}x_4 & +\cdots+ & 0x_{n-1} & + & 0x_n & = & a_{4,m} \\
0x_1 & + & 0x_2 & + & 0x_3 & + & a_{5,4}x_4 & +\cdots+ & 0x_{n-1} & + & 0x_n & = & a_{5,m} \\
\cdots & + & \cdots & + & \cdots & + & \cdots & +\cdots+ & \cdots & + & \cdots & = & \cdots \\
0x_1 & + & 0x_2 & + & 0x_3 & + & 0x_4 & +\cdots+ & a_{n,n-1}x_{n-1} & + & a_{n,n}x_n & = & a_{n,m}
\end{array} \qquad (6.51)$$

Donde *"n"* es el número de filas y *"m"* es el número de columnas.

Si sólo se guardan los valores diferentes de cero, el sistema tridiagonal puede ser reescrito como:

$$\begin{array}{rcrcrcr}
0 & + & a_{1,2}x_2 & + & a_{1,3}x_3 & = & a_{1,4} \\
a_{2,1}x_1 & + & a_{2,2}x_2 & + & a_{2,3}x_3 & = & a_{2,4} \\
a_{3,1}x_2 & + & a_{3,2}x_3 & + & a_{3,3}x_4 & = & a_{3,4} \\
a_{4,1}x_3 & + & a_{4,2}x_4 & + & a_{4,3}x_5 & = & a_{4,4} \\
a_{5,1}x_4 & + & a_{5,2}x_5 & + & a_{5,3}x_6 & = & a_{5,4} \\
\cdots & + & \cdots & + & \cdots & = & \cdots \\
a_{n,1}x_{n-1} & + & a_{n,2}x_n & + & 0 & = & a_{5,4}
\end{array} \qquad (6.52)$$

Con lo que se consigue un gran ahorro de memoria, porque en lugar de reservar memoria para n*m elementos sólo se requiere memoria para n*4 elementos. La mayoría de los métodos estudiados hasta ahora pueden ser adaptados de manera que trabajen con sistemas tridiagonales (ecuación 6.51).

En la deducción de las ecuaciones para el método de *Gauss*, se considera un sistema tridiagonal de 5 ecuaciones con 5 incógnitas y con el fin de facilitar la deducción de las ecuaciones se escriben también los ceros:

$$\begin{array}{rcrcrcrcrcr}
a_{1,2}x_1 & + & a_{1,3}x_2 & + & 0x_3 & + & 0x_4 & + & 0x_5 & = & a_{1,4} \\
a_{2,1}x_1 & + & a_{2,2}x_2 & + & a_{2,3}x_3 & + & 0x_4 & + & 0x_5 & = & a_{2,4} \\
0x_1 & + & a_{3,1}x_2 & + & a_{3,2}x_3 & + & a_{3,3}x_4 & + & 0x_5 & = & a_{3,4} \\
0x_1 & + & 0x_2 & + & a_{4,1}x_3 & + & a_{4,2}x_4 & + & a_{4,3}x_5 & = & a_{4,4} \\
0x_1 & + & 0x_2 & + & 0x_3 & + & a_{5,1}x_4 & + & a_{5,2}x_5 & = & a_{5,4}
\end{array} \qquad (6.53)$$

Que al aplicar el método de Gauss se convierte en:

$$\begin{array}{rcrcrcrcrcr}
a_{1,2}x_1 & + & a_{1,3}x_2 & + & 0x_3 & + & 0x_4 & + & 0x_5 & = & a_{1,4} \\
0x_1 & + & a_{2,2}x_2 & + & a_{2,3}x_3 & + & 0x_4 & + & 0x_5 & = & a_{2,4} \\
0x_1 & + & 0x_2 & + & a_{3,2}x_3 & + & a_{3,3}x_4 & + & 0x_5 & = & a_{3,4} \\
0x_1 & + & 0x_2 & + & 0x_3 & + & a_{4,2}x_4 & + & a_{4,3}x_5 & = & a_{4,4} \\
0x_1 & + & 0x_2 & + & 0x_3 & + & 0x_4 & + & a_{5,2}x_5 & = & a_{5,4}
\end{array} \qquad (6.54)$$

Por lo tanto sólo es necesario convertir en ceros los elementos que se encuentran debajo de la diagonal principal. Así para convertir en cero el elemento $a_{2,1}$, se resta a la segunda fila la primera multiplicada por $a_{2,1}/a_{1,2}$. Como no es necesario guardar ni calcular los ceros, se puede aprovechar el elemento $a_{2,1}$ para guardar el valor de esta división y con el mismo calcular los nuevos valores de los elementos $a_{2,2}$ y $a_{2,4}$ (el valor de $a_{2,3}$ no cambia):

$$a_{2,1}=\frac{a_{2,1}}{a_{1,2}}$$
$$a_{2,2}=a_{2,2}-a_{2,1}\cdot a_{1,3}$$
$$a_{2,4}=a_{2,4}-a_{2,1}\cdot a_{1,4}$$

De la misma forma, para convertir el elemento $a_{3,1}$ en cero se resta a la tercera fila la segunda multiplicada por $a_{3,1}/a_{2,2}$, al igual que antes almacenando este valor en el elemento $a_{3,1}$:

$$a_{3,1}=\frac{a_{3,1}}{a_{2,2}}$$
$$a_{3,2}=a_{3,2}-a_{3,1}\cdot a_{2,3}$$
$$a_{3,4}=a_{3,4}-a_{3,1}\cdot a_{2,4}$$

De manera similar, las operaciones para la cuarta y quinta fila son:

$$a_{4,1}=\frac{a_{4,1}}{a_{3,2}}$$
$$a_{4,2}=a_{4,2}-a_{4,1}\cdot a_{3,3}$$
$$a_{4,4}=a_{4,4}-a_{4,1}\cdot a_{3,4}$$

$$a_{5,1}=\frac{a_{5,1}}{a_{4,2}}$$
$$a_{5,2}=a_{5,2}-a_{5,1}\cdot a_{4,3}$$
$$a_{5,4}=a_{5,4}-a_{5,1}\cdot a_{4,4}$$

Analizando estas ecuaciones se pueden deducir las ecuaciones generales para reducir a cero los elementos que se encuentran debajo de la diagonal principal:

$$\left.\begin{aligned} a_{i,1}&=\frac{a_{i,1}}{a_{i-1,2}} \\ a_{i,2}&=a_{i,2}-a_{i,1}\cdot a_{i-1,3} \\ a_{i,4}&=a_{i,4}-a_{i,1}\cdot a_{i-1,4} \end{aligned}\right\} \; i=2\rightarrow n \qquad (6.55)$$

Una vez que el sistema ha quedado reducido a la forma (654), los resultados se calculan por sustitución inversa y pueden ser almacenados en la última columna (la columna 4) del sistema tridiagonal. Así el valor de la última variable (x_5) se calcula con:

$$x_5=a_{5,4}=\frac{a_{5,4}}{a_{5,2}}$$

De manera similar se calculan los valores de las otras variables:

$$x_4=a_{4,4}=\frac{a_{4,4}-a_{4,3}\cdot x_5}{a_{4,2}}=\frac{a_{4,4}-a_{4,3}\cdot a_{5,4}}{a_{4,2}}$$

$$x_3 = a_{3,4} = \frac{a_{3,4} - a_{3,3} \cdot x_4}{a_{3,2}} = \frac{a_{3,4} - a_{3,3} \cdot a_{4,4}}{a_{3,2}}$$
$$x_2 = a_{2,4} = \frac{a_{2,4} - a_{2,3} \cdot x_3}{a_{2,2}} = \frac{a_{2,4} - a_{2,3} \cdot a_{3,4}}{a_{2,2}}$$
$$x_1 = a_{1,4} = \frac{a_{1,4} - a_{1,3} \cdot x_2}{a_{1,2}} = \frac{a_{1,4} - a_{1,3} \cdot a_{2,4}}{a_{1,2}}$$

De donde se deducen las ecuaciones generales para el cálculo de las soluciones por sustitución inversa:

$$x_n = a_{n,4} = \frac{a_{n,4}}{a_{n,2}}$$
$$x_i = a_{i,4} = \frac{a_{i,4} - a_{i,3} \cdot x_{i+1}}{a_{i,2}} = \frac{a_{i,4} - a_{i,3} \cdot a_{i+1,4}}{a_{i,2}} \quad \left\{ i = n-1 \rightarrow 1 \right. \tag{6.56}$$

Como de costumbre, al momento de programar estas ecuaciones se deben cambiar los límites para tomar en cuenta que en Javascript el primer índice es 0 y no 1.

Para comprender mejor como se aplican las ecuaciones (6.55) y (6.56) se resuelve manualmente el siguiente sistema de ecuaciones:

$$\begin{array}{rcrcrcrcrcr} 2x_1 & + & x_2 & + & 0x_3 & + & 0x_4 & + & 0x_5 & = & 6 \\ x_1 & + & 4x_2 & + & x_3 & + & 0x_4 & + & 0x_5 & = & 36 \\ 0x_1 & + & x_2 & + & 4x_3 & + & x_4 & + & 0x_5 & = & 72 \\ 0x_1 & + & 0x_2 & + & x_3 & + & 4x_4 & + & x_5 & = & 108 \\ 0x_1 & + & 0x_2 & + & 0x_3 & + & x_4 & + & 2x_5 & = & 66 \end{array} \tag{6.57}$$

Primero se guardan los datos (la matriz y el número de filas):

```
>>a=[[0,2,1,6],[1,4,1,36],[1,4,1,72],[1,4,1,108],[1,2,0,66]]; n=a.length;
5
```

Luego, se aplican las ecuaciones (6.55) para reducir el sistema:

```
>>for (i=1;i<n;i++){ a[i][0]=a[i][0]/a[i-1][1]; a[i][1]=a[i][1]-a[i]
[0]*a[i-1][2]; a[i][3]=a[i][3]-a[i][0]*a[i-1][3];}; a
[[0, 2, 1, 6], [0.5, 3.5, 1, 33], [0.2857142857142857,
3.7142857142857144, 1, 62.57142857142857], [0.2692307692307692,
3.730769230769231, 1, 91.15384615384616], [0.26804123711340205,
1.731958762886598, 0, 41.567010309278345]]
```

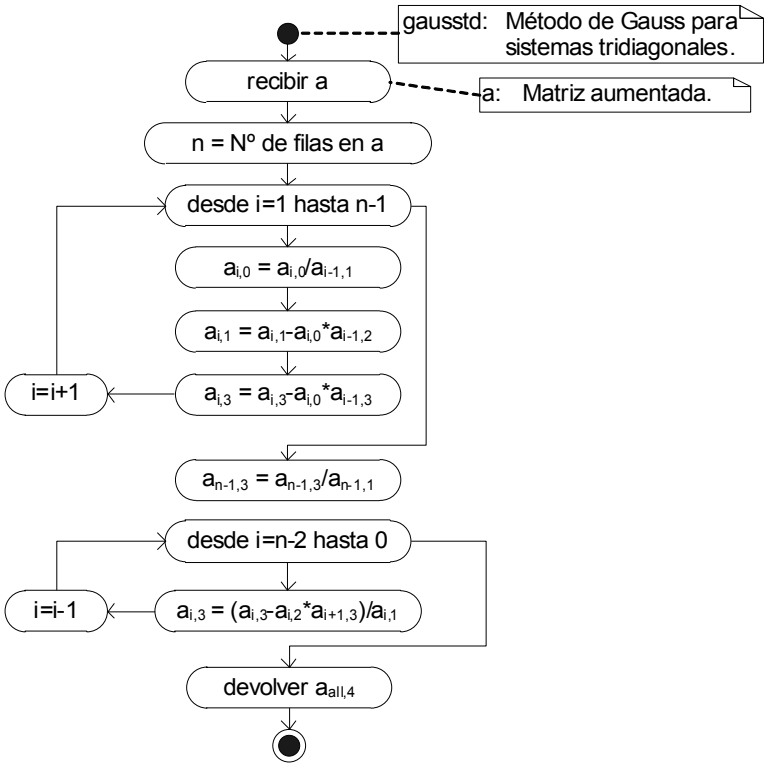
Una vez reducido el sistema se aplican las ecuaciones (6.56) para calcular las soluciones por sustitución inversa:

```
>>a[n-1][3]=a[n-1][3]/a[n-1][1]; for(i=n-2;i>=0;i--) a[i][3]=(a[i][3]-
a[i][2]*a[i+1][3])/a[i][1]; a
[[0, 2, 1, 0], [0.5, 3.5, 1, 6], [0.2857142857142857, 3.7142857142857144,
1, 11.999999999999998], [0.2692307692307692, 3.730769230769231, 1, 18],
[0.26804123711340205, 1.731958762886598, 0, 23.999999999999996]]
```

Por lo tanto, las soluciones redondeadas (que se encuentran en la última columna) son: $x_1=0$; $x_2=6$; $x_3=12$; $x_4=18$; $x_5=24$.

6.8.1. Algoritmo y código

El algoritmo que automatiza el proceso de cálculo es:



Siendo el código respectivo, añadido a la librería "mat205.js":

```
function gausstd(b) {
  var i,a=b.slice();n=a.length,x=new Array(n);
  for (i=1;i<n;i++) {
    a[i][0]/=a[i-1][1];
    a[i][1]-=a[i][0]*a[i-1][2];
    a[i][3]-=a[i][0]*a[i-1][3];}
  a[n-1][3]/=a[n-1][1];
  for (i=n-2;i>=0;i--)
    a[i][3]=(a[i][3]-a[i][2]*a[i+1][3])/a[i][1];
  for (i=0;i<n;i++) x[i]=a[i][3];
  return x;
}
```

Haciendo correr el programa con los datos del ejemplo manual se obtiene:

```
>>a=[[0,2,1,6],[1,4,1,36],[1,4,1,72],[1,4,1,108],[1,2,0,66]]; gausstd(a)
[0, 6, 11.999999999999998, 18, 23.999999999999996]
```

Que con 15 dígitos de precisión es:

```
>>precision(ans,15)
[0, 6, 12, 18, 24]
```

Que son los resultados exactos.

6.9. MÉTODO DE GAUSS SEIDEL PARA SISTEMAS TRIDIAGONALES

Generalmente los sistemas tridiagonales tienen además una diagonal dominante. Por ello, los métodos iterativos como *Jacobi* y *Gauss-Seidel* resultan particularmente útiles para resolver estos sistemas (como ya se dijo, en estos métodos, cuando la diagonal es dominante la convergencia está asegurada).

Al igual que en el método de *Gauss* se empleará el sistema tridiagonal de 5 ecuaciones (ecuación 657) para deducir las ecuaciones generales del método. Si se denomina "x" al vector con los valores iniciales asumidos, entonces, los nuevos valores se calculan con:

$$x_1=\frac{a_{1,4}-a_{1,3}\cdot x_2}{a_{1,2}}$$
$$x_2=\frac{a_{2,4}-a_{2,1}\cdot x_1-a_{2,3}\cdot x_3}{a_{2,2}}$$
$$x_3=\frac{a_{3,4}-a_{3,1}\cdot x_2-a_{3,3}\cdot x_4}{a_{3,2}}$$
$$x_4=\frac{a_{4,4}-a_{4,1}\cdot x_3-a_{4,3}\cdot x_5}{a_{4,2}}$$
$$x_5=\frac{a_{5,4}-a_{5,1}\cdot x_4}{a_{5,2}}$$

De donde se deducen las ecuaciones generales del método:

$$x_1=\frac{a_{1,4}-a_{1,3}\cdot x_2}{a_{1,2}}$$
$$x_i=\frac{a_{i,4}-a_{i,1}\cdot x_{i-1}-a_{i,3}\cdot x_{i+1}}{a_{i,2}} \quad \left\{ i=2\rightarrow n-1 \right. \qquad (6.58)$$
$$x_n=\frac{a_{n,4}-a_{n,1}\cdot x_{n-1}}{a_{n,2}}$$

Como de costumbre, en Javascript se deben cambiar los límites para tomar en cuenta el índice inicial 0. Para comprender mejor la aplicación de estas ecuaciones se resolverá manualmente el sistema de ecuaciones (6.57).

Primero se crea la matriz tridiagonal y se inicializan variables (siendo los valores asumidos ceros):

```
>>a=[[0,2,1,6],[1,4,1,36],[1,4,1,72],[1,4,1,108],[1,2,0,66]]; y=
[0,0,0,0,0]; n=a.length;
5
```

Entonces se calculan los nuevos valores de x (en la variable "y") aplicando las ecuaciones (6.58) y redondeando los resultados en el tercer dígito después del punto:

```
>>y[0]=(a[0][3]-a[0][2]*y[1])/a[0][1]; for (i=1;i<n-1;i++) y[i]=(a[i][3]-
a[i][0]*y[i-1]-a[i][2]*y[i+1])/a[i][1]; y[n-1]=(a[n-1][3]-a[n-1][0]*y[n-
2])/a[n-1][1]; round(y,3)
[3, 8.25, 15.938, 23.016, 21.492]
```

Ahora se vuelve a repetir el proceso hasta que los valores se repiten (redondeados en los 3 dígitos después del punto):

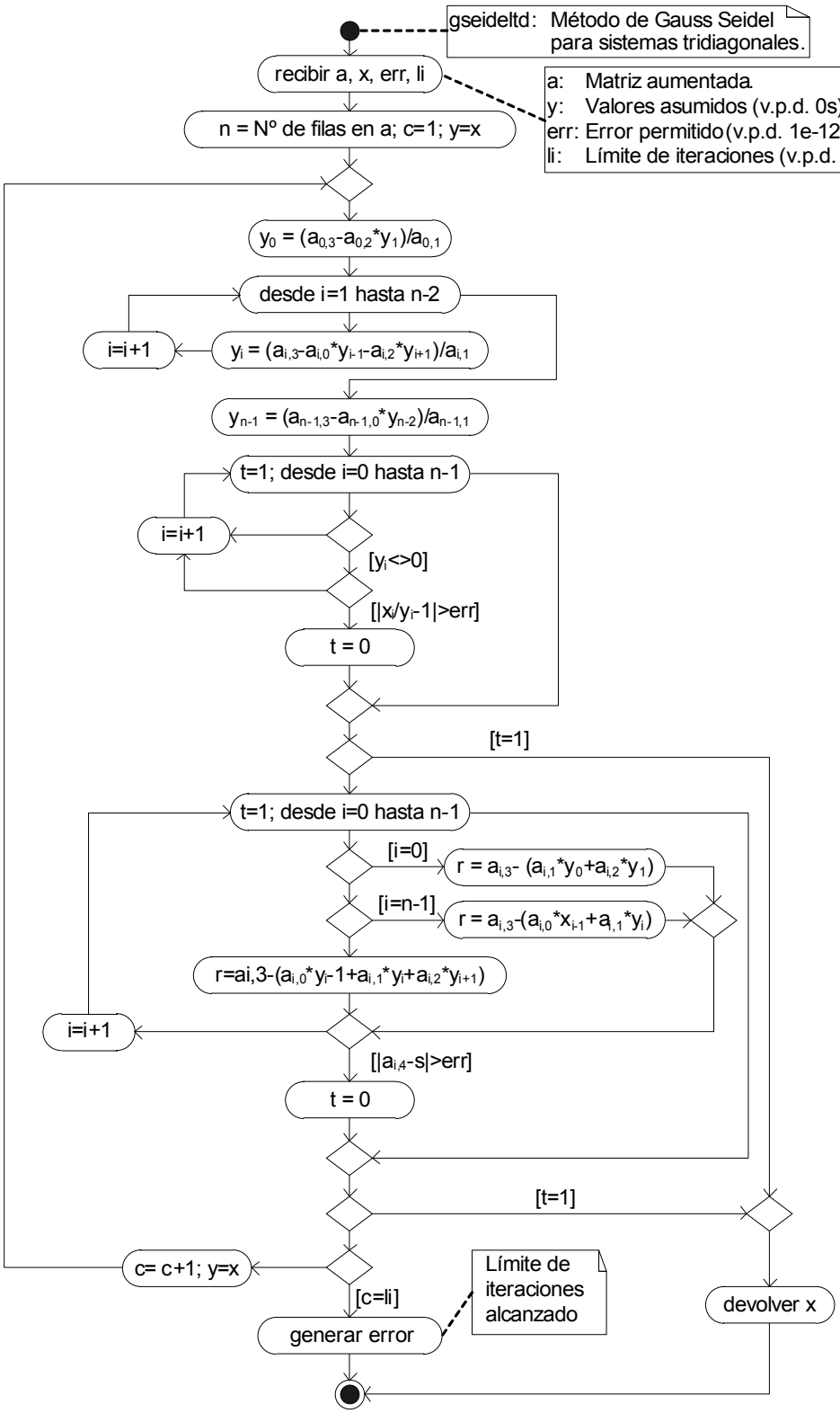
```
>>y[0]=(a[0][3]-a[0][2]*y[1])/a[0][1]; for (i=1;i<n-1;i++) y[i]=(a[i][3]-
a[i][0]*y[i-1]-a[i][2]*y[i+1])/a[i][1]; y[n-1]=(a[n-1][3]-a[n-1][0]*y[n-
2])/a[n-1][1]; round(y,3)
[-1.125, 5.297, 10.922, 18.896, 23.552]

>>y[0]=(a[0][3]-a[0][2]*y[1])/a[0][1]; for (i=1;i<n-1;i++) y[i]=(a[i][3]-
a[i][0]*y[i-1]-a[i][2]*y[i+1])/a[i][1]; y[n-1]=(a[n-1][3]-a[n-1][0]*y[n-
2])/a[n-1][1]; round(y,3)
[0.352, 6.182, 11.73, 18.179, 23.91]

>>y[0]=(a[0][3]-a[0][2]*y[1])/a[0][1]; for (i=1;i<n-1;i++) y[i]=(a[i][3]-
a[i][0]*y[i-1]-a[i][2]*y[i+1])/a[i][1]; y[n-1]=(a[n-1][3]-a[n-1][0]*y[n-
2])/a[n-1][1]; round(y,3)
[-0.091, 6.09, 11.933, 18.039, 23.98]

>>y[0]=(a[0][3]-a[0][2]*y[1])/a[0][1]; for (i=1;i<n-1;i++) y[i]=(a[i][3]-
a[i][0]*y[i-1]-a[i][2]*y[i+1])/a[i][1]; y[n-1]=(a[n-1][3]-a[n-1][0]*y[n-
2])/a[n-1][1]; round(y,3)
[-0.045, 6.028, 11.983, 18.009, 23.995]

>>y[0]=(a[0][3]-a[0][2]*y[1])/a[0][1]; for (i=1;i<n-1;i++) y[i]=(a[i][3]-
a[i][0]*y[i-1]-a[i][2]*y[i+1])/a[i][1]; y[n-1]=(a[n-1][3]-a[n-1][0]*y[n-
2])/a[n-1][1]; round(y,3)
[-0.014, 6.008, 11.996, 18.002, 23.999]

>>y[0]=(a[0][3]-a[0][2]*y[1])/a[0][1]; for (i=1;i<n-1;i++) y[i]=(a[i][3]-
a[i][0]*y[i-1]-a[i][2]*y[i+1])/a[i][1]; y[n-1]=(a[n-1][3]-a[n-1][0]*y[n-
2])/a[n-1][1]; round(y,3)
[-0.004, 6.002, 11.999, 18.001, 24]

>>y[0]=(a[0][3]-a[0][2]*y[1])/a[0][1]; for (i=1;i<n-1;i++) y[i]=(a[i][3]-
a[i][0]*y[i-1]-a[i][2]*y[i+1])/a[i][1]; y[n-1]=(a[n-1][3]-a[n-1][0]*y[n-
2])/a[n-1][1]; round(y,3)
[-0.001, 6.001, 12, 18, 24]

>>y[0]=(a[0][3]-a[0][2]*y[1])/a[0][1]; for (i=1;i<n-1;i++) y[i]=(a[i][3]-
a[i][0]*y[i-1]-a[i][2]*y[i+1])/a[i][1]; y[n-1]=(a[n-1][3]-a[n-1][0]*y[n-
2])/a[n-1][1]; round(y,3)
[0, 6, 12, 18, 24]
```

A partir de esta iteración los valores se repiten, por lo que el proceso concluye, siendo las soluciones: $x_1 \approx 0$; $x_2=6$; $x_3=12$, $x_4=18$; $x_5=24$.

6.9.1. Algoritmo y código

El algoritmo que automatiza el proceso de cálculo se presenta en la siguiente página, siendo el código respectivo:

```
function gseideltd(a,x,err,li) {
  var aux,i,j,t,r,n=a.length,c=1,y;
  if (x==null) {x=new Array(n); for(i=0;i<n;i++) x[i]=0;}
  if (err==null) err=1e-12;
  if (li==null) li=500;
  y=x.slice();
  while (true) {
    y[0]=(a[0][3]-a[0][2]*y[1])/a[0][1];
```




```
for (i=1;i<n-1;i++)
  y[i]=(a[i][3]-a[i][0]*y[i-1]-a[i][2]*y[i+1])/a[i][1];
y[n-1]=(a[n-1][3]-a[n-1][0]*y[n-2])/a[n-1][1];
for (t=1,i=0;i<n;i++)
  if (y[i])
```

```
        {if (abs(x[i]/y[i]-1)>err) {t=0; break;}}
      if (t) return y;
      for (t=1,i=0;i<n;i++) {
        if (i==0) r=a[i][3]-(a[i][1]*y[0]+a[i][2]*y[1]);
        else if (i==n-1) r=a[i][3]-(a[i][0]*y[i-1]+a[i][1]*y[i]);
        else r=a[i][3]-(a[i][0]*y[i-1]+a[i][1]*y[i]+a[i][2]*y[i+1]);
        if (abs(r)>err) {t=0; break;}};
      if (t) return y;
      if (c++ == li)
        throw new Error("Convergencia no lograda (gseideltd).");
      for (i=0;i<n;i++) x[i]=y[i];
    }
  }
```

Haciendo correr el programa para los datos del ejemplo manual (con los valores iniciales, error y límite de iteraciones por defecto) se obtiene:

```
>>a=[[0,2,1,6],[1,4,1,36],[1,4,1,72],[1,4,1,108],[1,2,0,66]];
gseideltd(a)
[-2.4513724383723456e-13, 6.000000000000121, 11.999999999999941,
18.00000000000003, 23.999999999999986]
```

Que redondeado a 12 dígitos después del punto (que es el error por defecto) es:

```
>>round(ans,12)
[0, 6, 12, 18, 24]
```

6.10. EJEMPLOS

1, Encuentre las soluciones del siguiente sistema de ecuaciones lineales aplicando los métodos de Gauss con pivotaje parcial, Gauss-Jordán con pivotaje total, el métoto de Crout con factorización, el método de Doolittle directo, la matriz inversa. Finalmente calcule el determinante de la matriz de coeficientes (muestre todos los resultados redondeados a 2 dígitos después del punto).

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &= -1 \\ 4x_1 + 5x_2 + 6x_3 + 7x_4 &= 0 \\ 6x_1 + 10x_2 + 15x_3 + 21x_4 &= 10 \\ 12x_1 + 30x_2 + 60x_3 + 105x_4 &= 10 \end{aligned}$$

En primer lugar se crea la matriz aumentada:

```
>>a=[[1,1,1,1,-1],[4,5,6,7,0],[6,10,15,21,10],[12,30,60,105,10]]
[[1, 1, 1, 1, -1], [4, 5, 6, 7, 0], [6, 10, 15, 21, 10], [12, 30,
60, 105, 10]]
```

Se resuelve el sistema con el método de Gauss con pivotaje parcial (mostrando los resultados redondeados a 2 dígitos después del punto):

```
>>mshow(round(gauss(a),2))
[[11.67]
 [-46]
 [50]
 [-16.67]]
```

Por lo tanto las cuatro soluciones del sistema son: x1= 11.67; x2=-46; x3=50 y x4=-16.67.

Con la misma matriz se vuelven a calcular los resultados con el método de gauss jordán con pivotaje total:

```
>>mshow(round(gaussjpt(a),2))
[[11.67]
 [-46]
 [50]
 [-16.67]]
```

Para encontrar las soluciones con los métodos de Crout y Doolitle, se necesitan las matrices de coeficientes y constantes. Dichas matrices pueden volver a ser introducidas, o mejor aún, obtenidas de la matriz aumentada, con "getcol(matriz, número_de_columna, número_de_columnas)":

```
>>b=getcol(a,0,4)
[[1, 1, 1, 1], [4, 5, 6, 7], [6, 10, 15, 21], [12, 30, 60, 105]]
>>c=getcol(a,4)
[[-1], [0], [10], [10]]
```

Con el método de Crout con factorización, se obtiene:

```
>>r=croutlu(b); mshow(round(croutsol(r,c),2))
[[11.67]
 [-46]
 [50]
 [-16.67]]
```

Y con el método de Doolittle directo:

```
>>mshow(round(doolittle(b,c),2))
[[11.67]
 [-46]
 [50]
 [-16.67]]
```

Finalmente, con la matriz inversa se obtiene:

```
>>mi=inv(b); mshow(round(mul(mi,c),2))
[[11.67]
 [-46]
 [50]
 [-16.67]]
```

2. Encuentre las soluciones del siguiente sistema de ecuaciones lineales, que se muestra en la siguiente página, aplicando: a) El método de Jacobi; b) El método de Gauss Seidel; c) El método de Cholesky directo; d) El método de Doolittle con factorización d) La matriz inversa.

$$\begin{aligned} 7x_1 + x_2 + 2x_3 - x_4 &= 3 \\ x_1 + 9x_2 + 3x_3 + 2x_4 &= 2 \\ 2x_1 + 3x_2 + 12x_3 + 3x_4 &= 7 \\ -x_1 + 2x_2 + 3x_3 + 11x_4 &= 5 \end{aligned}$$

Simplemente para mostrar otra alternativa, en este caso se crean primero las matrices de coeficientes y constantes:

```
>>b=[[7,1,2,-1],[1,9,3,2],[2,3,12,3],[-1,2,3,11]];c=[[3],[2],[7],[5]]
```

```
[[3], [2], [7], [5]]
```

Con estas dos matrices se crea la matriz aumentada empleando las funciones "copy(objeto_a_copiar)" (para copiar la maptriz) y "appendcol(matriz,columna)" (para añadir a la matriz copiada la columna).

```
>>a=copy(b); appendcol(a,c)
[[7, 1, 2, -1, 3], [1, 9, 3, 2, 2], [2, 3, 12, 3, 7], [-1, 2, 3, 11, 5]]
```

Entonces se resuelve el sistema de ecuaciones con los métodos de Jacobi y Gauss Seidel, empleando los valores iniciales y error por defecto:

```
>>precision(jacobi(a),11)
[0.36329352608, -0.0487115022, 0.44091766185, 0.37617850409]
>>precision(gseidel(a),11)
[0.36329352608, -0.0487115022, 0.44091766185, 0.37617850409]
```

Y con las matrices de coeficientes y constantes, se resuelve el sistema con los métodos de Crout, Doolittle y la matriz inversa:

```
>>precision(crout(b,c),11)
[[0.36329352608], [-0.0487115022], [0.44091766185], [0.37617850409]]
>>r=doolittlelu(b); precision(doolittlesol(r,c),11)
[[0.36329352608], [-0.0487115022], [0.44091766185], [0.37617850409]]
>>precision(mul(inv(b),c),11)
[[0.36329352608], [-0.0487115022], [0.44091766185], [0.37617850409]]
```

3. Encuentre las soluciones de los siguientes sistemas de ecuaciones aplicando: a) El método de Jacobi; b) El método de Gauss Seidel para sistemas tridiagonales; c) El método de Gauss para sistemas tridiagonales; d) El método de Doolittle con factorización; e) El método de Crout directo.

$$\begin{aligned} 19x_1 + 2x_2 &= 0 \\ 2x_1 + 15x_2 + 3x_3 &= 0 \\ 3x_2 + 20x_3 + 2x_4 &= 1 \\ 2x_3 + 34x_4 &= 0 \end{aligned}$$

$$\begin{aligned} 19y_1 + 2y_2 &= -2 \\ 2y_1 + 15y_2 + 3y_3 &= 2 \\ 3y_2 + 20y_3 + 2y_4 &= 3 \\ 2y_3 + 34y_4 &= 4 \end{aligned}$$

$$\begin{aligned} 19z_1 + 2z_2 &= 2 \\ 2z_1 + 15z_2 + 3z_3 &= 2 \\ 3z_2 + 20z_3 + 2z_4 &= 1 \\ 2z_3 + 34z_4 &= 0 \end{aligned}$$

Si bien existen funciones para manipular matrices tridiagonales (tales como diagonal y getdiagonal), en este ejemplo se crean dichas matrices en forma manual. Para el método de Jacobi se requiere la matriz aumentada de cada sistema, entonces, para no introducir dos o más veces los mismos datos, se crea la matriz de coeficientes y los vectores de constantes:

```
>>b=[[19,2,0,0],[2,15,3,0],[0,3,20,2],[0,0,2,34]]
[[19, 2, 0, 0], [2, 15, 3, 0], [0, 3, 20, 2], [0, 0, 2, 34]]
```

```
>>c1=[0,0,1,0]; c2=[-2,2,3,4]; c3=[2,2,1,0]
[2, 2, 1, 0]
```

Con estas matrices se crea la matriz aumentada para el primer sistema y se resuelve el mismo con el método de Jacobi:

```
>>a=copy(b); appendcol(a,c1); precision(jacobi(a),11)
[0.0011078406882, -0.010524486538, 0.051883872229, -0.0030519924841]
```

Ahora se reemplaza la columna de constantes (con "replacecol") por los vectores del segundo y tercer sistema y se resuelven los mismos con Jacobi:

```
>>replacecol(a,c2,4); precision(jacobi(a),11)
[-0.1184303418, 0.12508824711, 0.1201789923, 0.11057770634]
>>replacecol(a,c3,4); precision(jacobi(a),11)
[0.093232396737, 0.114292231, 0.03305058053, -0.0019441517959]
```

Para el método de Gauss-Seidel (para sistemas tridiagonales) se requiere la matriz conformada por las tres diagonales:

```
>>d=[[0,19,2],[2,15,3],[3,20,2],[2,34,0]]
[[0, 19, 2], [2, 15, 3], [3, 20, 2], [2, 34, 0]]
```

Al igual que Jacobi, Gauss-Seidel no puede resolver los tres sistemas a la vez, por lo que se resuelve sistema por sistema, cambiando en cada caso la columna de las constantes:

```
>>a=copy(d); appendcol(a,c1,3); precision(gseideltd(a),11)
[0.0011078406881, -0.010524486538, 0.051883872229, -0.0030519924841]
>>replacecol(a,c2,3); precision(gseideltd(a),11)
[-0.1184303418, 0.12508824711, 0.1201789923, 0.11057770634]
>>replacecol(a,c3,3); precision(gseideltd(a),11)
[0.093232396737, 0.114292231, 0.03305058053, -0.0019441517959]
```

El método de Gauss para sistemas tridagonales requiere los mismos datos que el método de Gauss-Seidel:

```
>>replacecol(a,c1,3); precision(gseideltd(a),11)
[0.0011078406881, -0.010524486538, 0.051883872229, -0.0030519924841]
>>replacecol(a,c2,3); precision(gseideltd(a),11)
[-0.1184303418, 0.12508824711, 0.1201789923, 0.11057770634]
>>replacecol(a,c3,3); precision(gseideltd(a),11)
[0.093232396737, 0.114292231, 0.03305058053, -0.0019441517959]
```

El método de Doolittle requiere la matriz de coeficientes y la matriz de constantes. La matriz de constantes se crea con los vectores de constantes, transponiendo la primera columna con "transpose(matriz)" (para convertirla en una matriz de una columna) y añadiendo a esta matriz (con "appendCol") los otros dos vectores:

```
>>c=transpose(copy(c1)); appendcol(c,c2,1); appendcol(c,c3,2); mshow(c)
[[0, -2, 2]
 [0, 2, 2]
 [1, 3, 1]
 [0, 4, 0]]
```

Con esta matriz y la matriz de coeficientes se resuelven, simultáneamente, los tres sistemas:

```
>>r=doolittlelu(b); mshow(precision(doolittlesol(r,c),11))
```

```
[[0.0011078406882, -0.1184303418, 0.093232396737]
 [-0.010524486538, 0.12508824711, 0.114292231]
 [0.051883872229, 0.1201789923, 0.03305058053]
 [-0.0030519924841, 0.11057770634, -0.0019441517959]]
```

Finalmente, con las mismas matrices, se encuentran las soluciones con el método de Crout:

```
>>mshow(precision(crout(b,c),11))
[[0.0011078406882, -0.1184303418, 0.093232396737]
 [-0.010524486538, 0.12508824711, 0.114292231]
 [0.051883872229, 0.1201789923, 0.03305058053]
 [-0.0030519924841, 0.11057770634, -0.0019441517959]]
```

4. Encuentre la solución (el valor de "x") de la siguiente función, resolviendo el sistema de ecuaciones lineales con el método de Gauss con pivotaje parcial, la ecuación polinomial con "roots" y la ecuación no lineal con el método de Regula Falsi. Los valores iniciales para resolver la ecuación no lineal deben ser calculados con el método incremental (se sabe que la solución es menor a 7).

$$f(x)=z_3 x^{3.2}+ z_2 \ln(x)-z_0 * x - z_1 = 0$$

$$\begin{aligned} y_1 + 3y_2 - 2y_3 + y_4 &= 335.137 \\ 2y_2 + 5y_3 + 4y_4 &= -122.815 \\ -4y_1 + 4y_2 + y_3 - 2y_4 &= 70.6096 \\ 3y_1 - 6y_3 - 7Y_4 &= 129.13 \end{aligned}$$

$$z^4 + y_1 z^3 + y_2 z^2 + y_3 z + y_4 = 0$$

Primero se resuelve el sistema de ecuaciones lineales con el método de Gauss con pivotaje parcial:

```
>>y=gauss([[1,3,-2,1,335.137],[0,2,5,4,-122.815],[-4,4,1,-2,70.6096],
[3,0,-6,-7,129.13]]); precision(y,7)
[[-11.79989], [47.93002], [-79.32787], [44.49108]]
```

Con estas soluciones se resuelve la ecuación de cuarto grado con "roots", siendo el vector de coeficientes:

```
>>a=[1,y[0][0],y[1][0],y[2][0],y[3][0]]; precision(a,7)
[1, -11.79989, 47.93002, -79.32787, 44.49108]
```

Observe que los elementos se extraen empleando dos índices (fila y columna). Se procede así porque los resultados devueltos tanto por Gauss como por Gauss Jordán son matrices, en este caso de una sola columna porque sólo se ha resuelto un sistema de ecuaciones. Con este vector, las soluciones de la ecuación de cuarto grado son:

```
>>z=roots(a); precision(z,7)
[5.199316, 3.101179, 2.299364, 1.20003]
```

Con estas soluciones se crea la función para la ecuación no lineal:

```
>>f=function(x){return z[3]*pow(x,3.2)+z[2]*ln(x)-z[0]*x-z[1];}
function()
```

Y se encuentra la solución pedida con el método de Regula Falsi, ubicando

el segmento de solución con "incr1", comenzando la búsqueda en 7 y siendo el incremento -0.1 (pues se sabe que la solución es menor a 7):

```
>>z1=incr1(f,7,-0.1); z2=z1-0.1; x=refa(f,z1,z2); precision(x,7)
2.06191
```

5. Encuentre la solución (el valor de "x") de la siguiente función no lineal empleando el método de Newton Raphson, donde z_0 es la solución real de la ecuación polinomial y y_1 a y_4 son las soluciones del sistema de ecuaciones lineales. El sistema de ecuaciones lineales debe ser resuelto con el método de Doolittle y la ecuación cúbica con "cub". El valor inicial para el método de Newton debe ser encontrado graficando la función.

$$f(x)=y_1+y_2-y_3\,y_4+z_0^{1.0135}-z_0=0$$

$$\begin{aligned} y_1+2\,y_2-y_3+3\,y_4&=3.75\,x\\ 2\,y_1-y_2+5\,y_3-6\,y_4&=1.46\,x\\ 4\,y_1+y_2-7\,y_3+y_4&=-0.49\,x\\ y_1+y_2+y_3+y_4&=1.97\,x^{1.8} \end{aligned}$$

$$z^3-6.78\,z^2+14.678\,z-5\,x=0$$

En este caso se trata de un sistema que depende de una variable, la variable "x": los valores de "y" y "z" dependen de esta variable.

Como el sistema de ecuaciones lineales se resuelve con el método de Doolitle y como en dicho método la factorización sólo depende de la matriz de coeficientes, que en este caso es constante, dicha factorización puede ser hecha antes de crear la función (para evitar cálculos innecesarios):

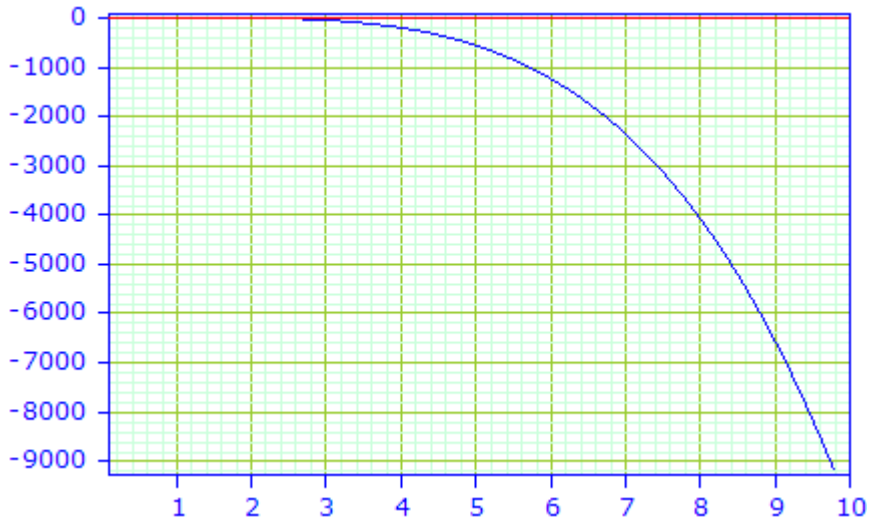
```
>>b=[[1,2,-1,3],[2,-1,5,-6],[4,1,-7,1],[1,1,1,1]]
[[1, 2, -1, 3], [2, -1, 5, -6], [4, 1, -7, 1], [1, 1, 1, 1]]
>>r=doolittlelu(b)
[[[4, 1, -7, 1], [0.25, 1.75, 0.75, 2.75], [0.5, -0.8571428571428571,
9.142857142857142, -4.142857142857142], [0.25, 0.42857142857142855,
0.26562500000000006, 0.671875]], [[0, 0, 1, 0], [1, 0, 0, 0], [0, 1, 0,
0], [0, 0, 0, 1]]]
```

Ahora, con la matriz factorizada y la matriz de permutaciones (guardadas en la variable "r"), se crea la función:

```
>>f=function(x){var c,y,a,z;  c=[[3.75*x],[1.46*x],[-0.49*x],
[1.97*pow(x,1.8)]];  y=doolittlesol(r,c); a=[1,-6.78,14.678,-5*x];
z=cub(a);  return y[0][0]+y[1][0]-y[2][0]*y[3][0]+pow(z[0],1.0135)-
z[0]; }
function (x){var c,y,a,z;
 c=[[3.75*x],[1.46*x],[-0.49*x],[1.97*pow(x,1.8)]];
 y=doolittlesol(r,c); a=[1,-6.78,14.678,-5*x]; z=cub(a);
 return y[0][0]+y[1][0]-y[2][0]*y[3][0]+pow(z[0],1.0135)-z[0];
}
```

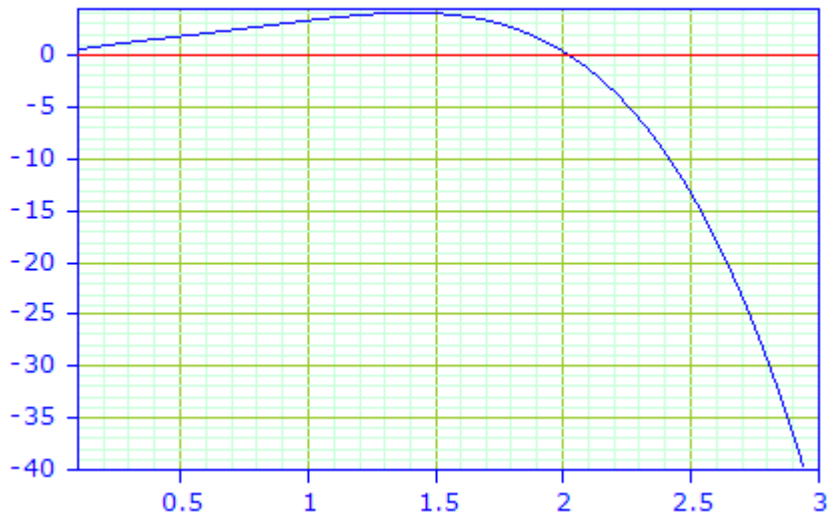
Se sabe que la solución está en el lado positivo, porque existe un término elevado a una potencia real y como se sabe una potencia real sólo tiene resultados reales si su base es positiva. Por lo tanto, para obtener el valor inicial se grafica la función en el lado positivo, en este caso se prueba buscando la solución entre 0.1 y 10:

```
>>plot([f,function(x){return 0;}],0.1,10)
```



Con esta gráfica se puede puede afirmar que existe por lo menos una solución entre 0.1 y 3, pero la misma no puede ser vista con claridad, por lo que se vuelve a graficar entre estos límites:

```
>>plot([f,function(x){return 0;}],0.1,3)
```



Ahora se ve claramente que existe una solución cercana a 2, por que se puede encontrar una solución más exacta empleando dicho valor como valor inicial para el método de Newton:

```
>>precision(newton(f,2),14)
2.0176982172789
```

Siendo esta la solución pedida.

6.11. EJERCICIOS

Como en los anteriores temas para presentar los ejercicios que resuelva en la calculadora, copie las instrucciones en un editor de texto y luego, al momento de presentar el trabajo, vuelva a copiarlos y ejecutarlos en la calculadora. Alternativamente, puede guardar el ejercicio resuelto como una página HTML, haciendo clic derecho en página y eligiendo la opción "guardar como" (o su equivalente) y guardando la página como página web completa, o fichero único, asignándole un nombre adecuado como ejercicio1.html.

1. Encuentre las soluciones del siguiente sistema de ecuaciones lineales aplicando el método de Gauss Jordán con pivotaje total, Gauss con pivotaje parcial y la matriz inversa calculada con el método de Gauss Jordán con pivotaje parcial, corrobore dicho resultado con "inv". Finalmente calcule el determinante de la matriz de coeficientes (muestre todos los resultados redondeados a 1 dígitos después del punto).

$$\begin{aligned} 3y_1 + 2y_2 - y_3 + 2y_4 &= -2 \\ y_1 + 4y_2 + y_3 + 2y_4 &= 2 \\ 2y_1 + y_2 + 2y_3 - y_4 &= 3 \\ y_1 + y_2 - y_3 + 3y_4 &= 4 \end{aligned}$$

2. Encuentre las soluciones del siguiente sistema de ecuaciones lineales aplicando: a) choleskylu, choleskysol y cholesky; b) doolittlelu, doolittlesol y doolittle; c) gausspt y gaussjpt; d) inv. Muestre los resultados en forma matricial y redondeados a dos dígitos después del punto.

$$\begin{aligned} 7x_1 + x_2 + 2x_3 - x_4 &= 3 \\ x_1 + 9x_2 + 3x_3 + 2x_4 &= 2 \\ 2x_1 + 3x_2 + 12x_3 + 3x_4 &= 7 \\ -x_1 + 2x_2 + 3x_3 + 11x_4 &= 5 \end{aligned}$$

3. Encuentre las soluciones del siguiente sistema de ecuaciones lineales empleando: a) El método de Jacobi; b) El método de Gauss Seidel; c) El método de Crout (con croutlu y croutsol); d) El método de Doolittle directo; e) El método de Gauss Jordán con pivotaje total; f) El método de la matriz inversa. Muestre los resulados en forma matricial y redondeados al primer dígito después del punto.

$$\begin{aligned} 22x_1 - 2x_2 + 5x_3 + 2x_4 &= 13 \\ x_1 + 31x_2 + 3x_3 + 3x_4 &= 20 \\ 3x_1 - x_2 + 43x_3 + 5x_4 &= 12 \\ 4x_1 + 3x_2 + x_3 + 21x_4 &= 15 \end{aligned}$$

4. Encuentre las soluciones del siguiente sistema de ecuaciones lineales empleando: a) El método de gauss para sistemas tridiagonales; b) El método de Gauss Seidel para sistemas tridiagonales; c) El método de Gauss Seidel estándar; d) El método de Crout directo; e) El método de Doolittle con factorización y e) El método de Gauss Jordán con pivotaje parcial. Muestre los resultados en forma matricial y redondeados al segundo dígito después del punto.

$$\begin{aligned} 4x_1 + 2x_2 + 0x_3 + 0x_4 + 0x_5 &= 8 \\ 3x_1 + 8x_2 + x_3 + 0x_4 + 0x_5 &= 14 \\ 0x_1 + 2x_2 + 12x_3 + 3x_4 + 0x_5 &= 21 \\ 0x_1 + 0x_2 + x_3 + 9x_4 + 2x_5 &= 34 \\ 0x_1 + 0x_2 + 0x_3 + 3x_4 + 14x_5 &= 37 \end{aligned}$$

5. Encuentre las soluciones de los siguientes sistemas de ecuaciones lineales: a) Resolviendo sistema por sistema con croutlu y croutsol; b) Resolviendo los sistemas simultáneamente con doolittlelu y doolittlesol;

c) Resolviendo los sistemas simultáneamente con gaussj; d) Resolviendo sistema por sistema con inv; e) Resolviendo los sistemas simultáneamente con crout. Muestre los resultados en forma matricial y redondeados al cuarto dígito después del punto.

$$\begin{aligned} 3x_1 + 2x_2 - x_3 + 2x_4 &= 0 \\ x_1 + 4x_2 + + 2x_4 &= 0 \\ 2x_1 + x_2 + 2x_3 - x_4 &= 1 \\ x_1 + x_2 - x_3 + 3x_4 &= 0 \end{aligned}$$

$$\begin{aligned} 3y_1 + 2y_2 - y_3 + 2y_4 &= -2 \\ y_1 + 4y_2 + + 2y_4 &= 2 \\ 2y_1 + y_2 + 2y_3 - y_4 &= 3 \\ y_1 + y_2 - y_3 + 3y_4 &= 4 \end{aligned}$$

$$\begin{aligned} 3z_1 + 2z_2 - z_3 + 2z_4 &= 2 \\ z_1 + 4z_2 + + 2z_4 &= 2 \\ 2z_1 + z_2 + 2z_3 - z_4 &= 1 \\ z_1 + z_2 - z_3 + 3z_4 &= 0 \end{aligned}$$

6. Encuentre la solución (el valor de "x") de la siguiente función, resolviendo el sistema de ecuaciones lineales con el método de la matriz inversa, la ecuación polinomial con "roots" y la ecuación no lineal con el método de la bisección (con 7 dígitos de precisión). En esta ecuación y_1 a y_5 son las soluciones del sistema de ecuaciones lineales, y z_0 a z_4 son las soluciones de la ecuación de quinto grado. Los valores iniciales para resolver la ecuación no lineal deben ser calculados con el método incremental (se sabe que la solución es mayor a 0.5).

$$f(x) = y_1 y_2 x^{1.3} - z_4 z_3 \ln(x) + \frac{y_3 y_4}{y_5} e^{\frac{x}{z_2 + z_1}} - z_0 y_5 = 0$$

$$\begin{aligned} 2y_1 + y_3 - 2y_4 + y_5 &= 2.3 \\ y_1 - y_2 + 2y_3 + y_4 - 3y_5 &= -1.9 \\ 3y_1 + y_2 - 2y_3 + 5y_4 - 4y_5 &= 2.5 \\ y_1 + y_2 + y_3 - y_4 - y_5 &= -0.2 \\ 2y_1 + 3y_2 + 4y_3 - 5y_4 - y_5 &= 0.5 \end{aligned}$$

$$z^5 - 11.8z^4 + 54.8z^3 - 125.042z^2 + 140.02z - 61.4916 = 0$$

7. Encuentre las 3 soluciones (los valores de "x") de la siguiente función no lineal empleando el método de la secante, donde z_0 es la solución real de la ecuación polinomial y y_1 a y_4 son las soluciones del sistema de ecuaciones lineales. El sistema de ecuaciones lineales debe ser resuelto con el método de Crout directo, la solución real de la ecuación polinomial debe ser calculada con el método de Newton (se sabe que la solución real está comprendida entre 6 y 10). Los valores iniciales para el método de la secante deben ser obtenidos de la gráfica de la función,

las soluciones deben ser mostradas redondeadas al tercer dígito después del punto.

$$f(x)=\sqrt{2\,y_1+3\,y_2+4\,y_3-y_4}-\sqrt[3]{2\,z_0+7}-1.87=0$$

$$\begin{gathered}3\,y_1-2\,y_2+4\,y_3-y_4=1.26*x\\ y_1+y_2-3\,y_3+2\,y_4=e^{0.131\,x}\\ 5\,y_1+y_2-y_3-3\,y_4=x/8.11\\ 2\,y_1+3\,y_2+4\,y_3-5\,y_4=\sqrt{6.15\,x}\end{gathered}$$

$$z^5-16.32\,z^4+116.84\,z^3-464.8\,z^2+1054.88\,z-(x/1.79156)^5=0$$

7. ECUACIONES NO LINEALES

Para la solución de sistemas de ecuaciones no lineales (con excepción de los polinomios) no existen métodos analíticos, por lo que dichos sistemas sólo pueden ser resueltos empleando métodos numéricos.

Como sucede con la mayoría de los métodos numéricos, para comenzar el proceso, se requieren valores iniciales (valores de prueba) y en este caso es aún más importante que dichos valores sean lo más cercanos posible a las soluciones debido a que los sistemas de ecuaciones no lineales tienen conjuntos de soluciones, de manera que se puede encontrar un conjunto de soluciones diferente al buscado (o a ninguno si no se logra convergencia).

Cuando el sistema de ecuaciones no lineales puede ser expresado en función de una sola variable, la forma más eficiente y segura de resolverlo es empleando los métodos estudiados en temas previos (tales como los métodos de la secante y Newton).

7.1. MÉTODO SIMPLEX

En realidad, el método Simplex es un método de optimización, es decir un método que permite encontrar un máximo o un mínimo y en consecuencia puede ser empleado para resolver una gran variedad de problemas, tales como optimizar ganancias (es decir maximizar los beneficios), optimizar gastos (minimizar costos), ajustar datos (minimizar residuos), optimizar el aprovechamiento de recursos (maximizar el tiempo de uso), etc.

En este sentido puede ser empleado también para resolver sistemas de ecuaciones no lineales, porque los sistemas de ecuaciones no lineales pueden ser expresados como un problema de optimización, donde el objetivo es minimizar las diferencias entre los lados izquierdo y derecho de las ecuaciones (es decir minimizar los residuos). Por ejemplo, si la función a resolver es:

$$f(x_1, x_2, x_3, \cdots, x_n) = \alpha \tag{7.1}$$

La solución se encuentra cuando al reemplazar los valores de las variables independientes (x_1 a x_n), la igualdad se cumple. Una forma conveniente de conseguirlo es colocando la función en forma de un problema de optimización, más específicamente de minimización:

$$g(x_1, x_2, x_3, \cdots, x_n) = f(x_1, x_2, x_3, \cdots, x_n) - \alpha = 0 \tag{7.2}$$

Sin embargo, esta forma tiene un inconveniente pues la diferencia puede ser positiva o negativa y si es negativa, es menor a cero, entonces se podría concluir erróneamente que se ha encontrado el mínimo. Este problema puede ser evitado si la diferencia se eleva al cuadrado:

$$\left[g(x_1, x_2, x_3, \cdots, x_n)\right]^2 = \left[f(x_1, x_2, x_3, \cdots, x_n) - \alpha\right]^2 = 0 \tag{7.3}$$

De esa manera nunca se tienen valores negativos. Cuando la función se coloca en esta forma, se conoce como la ***función de error***, y es la forma en que deben ser colocadas las expresiones para que puedan ser resueltas por el método Simplex:

$$e(x_1, x_2, x_3, \cdots, x_n) = \left[f(x_1, x_2, x_3, \cdots, x_n) - \alpha\right]^2 = 0 \tag{7.4}$$

Es importante recordar que el objetivo de esta transformación es simplemente la de asegurar que la función devuelva siempre valores positivos por lo que en ese sentido se puede recurrir a otras alternativas.

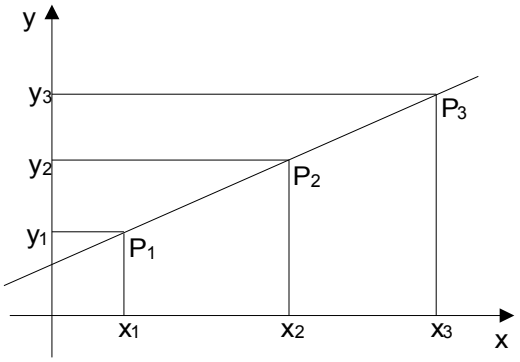
Ahora bien, partiendo de un conjunto de valores iniciales asumidos (plan inicial Simplex), el método encuentra nuevos valores proyectando los valores asumidos en dirección al mínimo. Dicha proyección se la realiza mediante interpolación o extrapolación lineal de puntos "n-dimensionales".

7.1.1. Proyección lineal n-dimensional

En un sistema de ecuaciones no lineales, cada valor asumido constituye un vector, o, lo que es lo mismo un punto "n-dimensional", que tiene tantas dimensiones como variables tiene el sistema.

La deducción de las ecuaciones que permiten la proyección de un punto a través de otros dos, se hará empleando puntos en el plano (vectores con dos dimensiones), sin embargo, las ecuaciones resultantes tendrán aplicación general.

Tomando en cuenta la siguiente figura:



Donde se conocen los puntos P_1 y P_2 y el objetivo calcular el punto P_3. Si se conocen las distancias que existen entre los puntos P_1, P_2 y P_1, P_3, es decir si se conoce la relación:

$$k = \frac{\overline{P_1 P_3}}{\overline{P_1 P_2}} \tag{7.5}$$

Los elementos del punto P_3 (x_3, y_3), se calculan con:

$$k = \frac{x_3 - x_1}{x_2 - x_1}$$
$$x_3 = k \cdot x_2 + (1-k) \cdot x_1 \tag{7.6}$$

$$k = \frac{y_3 - y_1}{y_2 - y_1}$$
$$y_3 = k \cdot y_2 + (1-k) \cdot y_1 \tag{7.7}$$

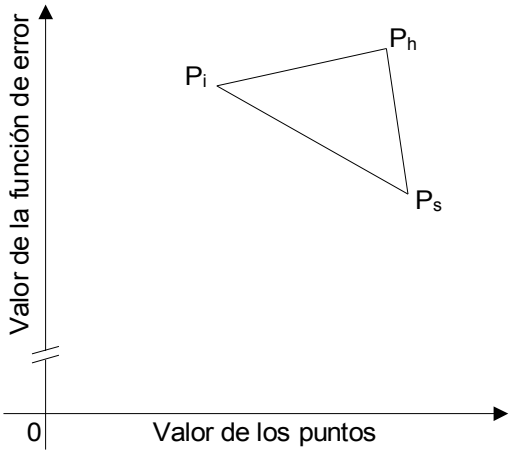
O generalizando, para cualquier punto P (un vector con "n" dimensiones o elementos), se tiene:

$$P_3 = k \cdot P_2 + (1-k) \cdot P_1 \tag{7.8}$$

7.1.2. Plan inicial Simplex

Como ya se dijo, para comenzar el método se requiere un conjunto de valores iniciales asumidos, dicho conjunto debe estar conformado por n+1 valores (puntos) siendo "n" el número de dimensiones (variables) del sistema.

Por ejemplo, para dos dimensiones se requieren 3 puntos y si los puntos son los que se muestran en la figura:



Al punto que se encuentra más cerca del mínimo (más cerca de 0) se denomina P_s ("s" de smallest) mientras que al que se encuentra más alejado se denomina P_h ("h" de highest). Los otros puntos se identifican con un número.

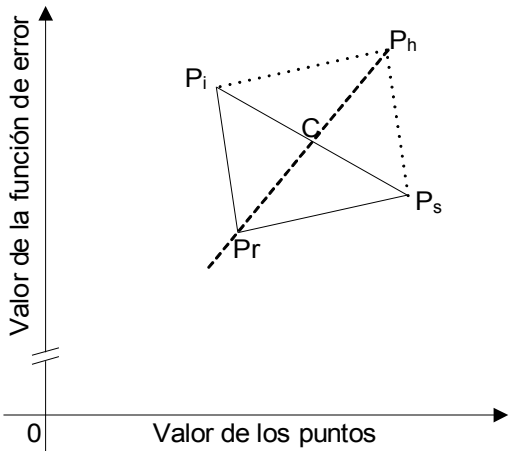
7.1.3. Centroide

En el método Simplex se busca reemplazar el peor valor asumido (o el peor valor del plan) por uno más cercano al mínimo. Con este propósito se calcula el promedio de todos los puntos del plan Simplex, sin incluir el punto P_h. Al punto resultante se denomina centroide "C" y se calcula con:

$$C=\frac{1}{n}\sum_{i=1}^{n+1} P_i \quad \{i\neq h \tag{7.9}$$

7.1.4. Reflexión

Para reemplazar el punto que se encuentra más alejado del mínimo (P_h), se realiza una proyección lineal del punto P_h a través del centroide C:



De manera que el punto proyectado sea el reflejo del punto P_h (siendo P_i-P_s el plano de proyección). Si se denomina "a" a la relación entre las distancias $(P_h-P_r)/(P_h-C)$ (el valor de "k" en las ecuación (7.8)), la ecuación de proyección para el cálculo del punto reflejado P_r es:

$$P_r=a\cdot C+(1-a)\cdot P_h \tag{7.10}$$

Donde el valor de "a" es 2. En la práctica, sin embargo, se emplea un valor un tanto diferente (por lo general 1.95), porque un valor entero puede dar lugar a un comportamiento oscilatorio.

Si se denomina e_r al valor de la función de error en el punto P_r ($e(P_r)$), e_h al valor de la función de error en el punto P_h ($e(P_h)$) y e_s al valor de la función de error en el punto P_s ($e(P_s)$), al realizar la reflexión se pueden presentar los siguientes casos:

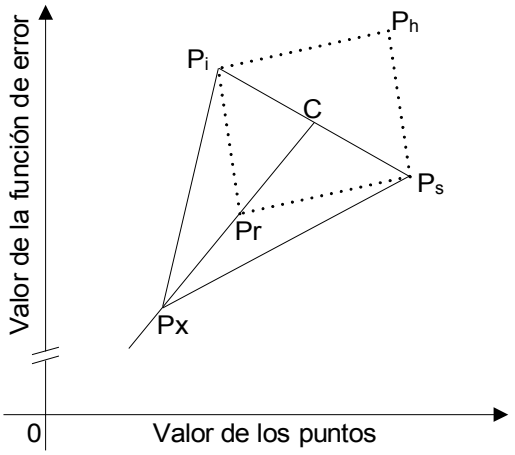
a) Si e_r es menor e_s, entonces la reflexión ha sido muy buena, pues se ha conseguido un punto más cercano al mínimo. Entonces se procede con la expansión (que se estudiará más adelante).

b) Si e_r es menor e_h pero no menor a e_s, entonces la reflexión es buena. En este caso se reemplaza el punto P_h por el punto P_r.

c) Si e_r es mayor a e_h, entonces la reflexión es mala. En este caso se procede con la contracción (que se estudiará más adelante).

7.1.5. Expansión

Como se dijo, cuando la reflexión es muy buena ($e_r<e_s$), se procede con la expansión, que simplemente consiste en proyectar el punto P_r al doble de la distancia que existe entre P_r y C:



Si se denomina "b" a la relación entre las distancias $(P_x-C)/(P_r-C)$, la ecuación de proyección (ecuación (7.8)) se transforma en:

$$P_x = b \cdot P_r + (1-b) \cdot C \qquad (7.11)$$

Donde "b" es igual a 2, sin embargo y como sucede en la reflexión, se emplea un valor cercano a 2 (generalmente 1.95).

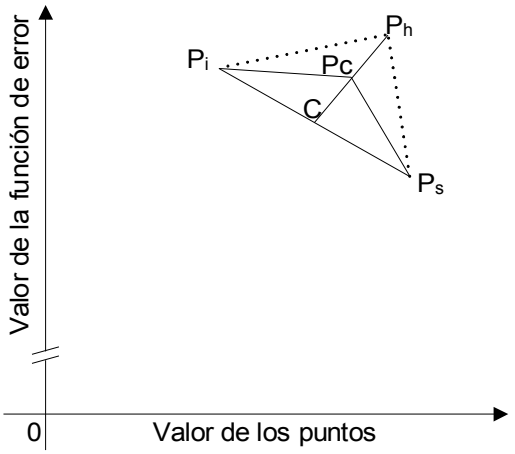
Si se denomina e_x al valor de la función de error en P_x ($e(P_x)$), entonces se pueden presentar los siguientes casos:

a) Si e_x es menor a e_r, el nuevo punto se aproxima más aún al mínimo, por lo tanto P_x reemplaza a P_h.

b) Si e_x es mayor a e_r, la expansión no ha sido buena, por lo tanto se mantiene el punto de reflexión, es decir que P_r reemplaza a P_h.

7.1.6. Contracción

Como se dijo, cuando la reflexión es mala, se procede con la contracción, que consiste en recorrer el punto reflejado a la mitad de la distancia que existe entre C y P_h:



Si se denomina "d" a la relación entre las distancias $(P_c-C)/(P_h-C)$, la ecuación de proyección (ecuación (78)) se transforma en:

$$P_c = d \cdot C + (1-d) \cdot P_h \qquad (7.12)$$

Donde "d" es 0.5, pero al igual que en los otros casos (para evitar un comportamiento oscilatorio) se toma un valor cercano a 0.5 (generalmente 0.45).

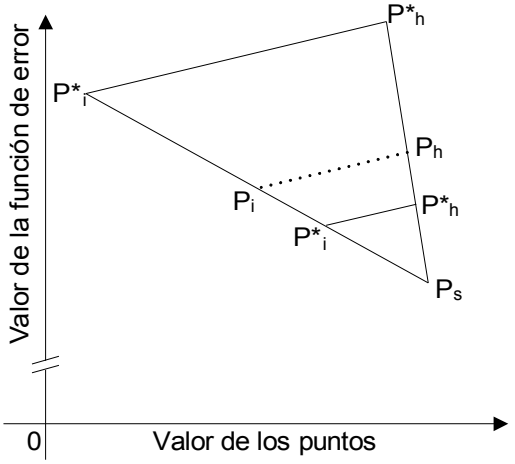
Si se denomina e_c al valor de la función de error en el punto P_c ($e(P_c)$), entonces se pueden presentar los siguientes casos:

a) Si e_c es menor a e_h, la contracción ha sido buena, entonces se reemplaza P_h por P_c.

b) Si e_c es mayor a e_h, la contracción ha sido mala, se procede con el escalamiento.

7.1.7. Escalamiento

Como se dijo, cuando la contracción es mala se procede con el escalamiento. En el escalamiento se proyectan los puntos del plan Simplex (con excepción de P_s) a través del punto P_s:



El escalamiento puede hacerse hasta la mitad de la distancia que existe entre los puntos P_i y P_s (como se muestra en la figura) o al doble de esta distancia. Si se denomina P* a los puntos resultantes de la proyección, la ecuación para el cálculo de estos puntos es:

$$P^*_i = k \cdot P_s + (1-k) \cdot P_i \quad \{i = 1 \rightarrow n+1;\ i \neq s \tag{7.13}$$

Donde el valor de k puede ser 2 (en la práctica 1.95) o 0.5 (en la práctica 0.45).

Como el escalamiento cambia todos los puntos del plan (con excepción del punto P_s), es necesario recalcular los valores de la función de error para los nuevos puntos del plan.

En el método Simplex, las operaciones de reflexión y las consiguientes operaciones de expansión, contracción y/o escalamiento se repiten hasta que se encuentra un valor de la función de error aproximadamente igual a cero o hasta que los valores de P_s y P_h son casi iguales. Dado que la convergencia no está asegurada, se debe emplear también un contador de iteraciones para terminar el proceso cuando se alcance un límite determinado.

7.1.8. Plan inicial Simplex

Como ya se señaló previamente un plan Simplex está conformado por n+1 puntos, siendo "n" el número de dimensiones (variables del sistema). Si por ejemplo se tiene un sistema con 5 dimensiones (o variables), se requieren 6 puntos para el plan inicial Simplex, ello implica asumir los valores de 30 variables (5 variables para cada punto), una labor tediosa que puede desalentar inclusive el empleo del método.

Por esta razón se han desarrollado algunos procedimientos para generar los puntos del plan inicial Simplex en base a un solo punto asumido.

Uno de dichos métodos es el siguiente: Sea P_1 el punto asumido, entonces el segundo punto (P_2) se genera multiplicando el primer elemento del punto asumido por 1.1, el tercer punto (P_3) multiplicando el segundo elemento del punto asumido por 1.1, el cuarto punto (P_4) multiplicando el tercer elemento del punto asumido por 1.1 y así sucesivamente hasta generar los n+1 puntos del plan.

Por ejemplo, si el sistema tiene 3 dimensiones (o variables) y el punto asumido es P1=(15,15,150), los otros puntos del plan Simplex se obtienen de la siguiente manera:

P2 = (15*1.1, 15, 150) = (16.5, 15, 150)

P3 = (15, 15*1.1, 150) = (15, 16.5, 150)

P4 = (15, 15, 150*1.1) = (15, 15, 165)

7.1.9. Ejemplo manual

Como este método, y los otros que se estudiarán en este tema, requiere muchas iteraciones para encontrar un resultado, en la práctica no se emplea nunca de forma manual, sin embargo, para comprender mejor el proceso y así poder automatizar el mismo, se realizarán algunas iteraciones manuales para el siguiente sistema (que por cierto puede ser resuelto con los métodos estudiados en temas previos):

$$\begin{aligned} x^2 + 2y^2 &= 22 \\ -2x^2 + xy - 3y &= -11 \end{aligned} \tag{7.14}$$

Para resolver este sistema con el método Simplex, primero se programa la función de error y dado que la misma debe recibir un vector con "n" elementos (siendo "n" el número de variables del sistema, en este caso 2), se reescribe el sistema de ecuaciones en forma vectorial:

$$\begin{aligned} x_0^2+2x_1^2 &= 22 \\ -2x_0^2+x_0x_1-3x_1 &= -11 \end{aligned} \qquad (7.15)$$

Para crear la función de error (un problema de minimización) las ecuaciones se igualan a cero, se elevan al cuadrado y se suman:

$$e(x)=\left[x_0^2+2x_1^2-22\right]^2+\left[-2x_0^2+x_0x_1-3x_1+11\right]^2=0$$

Siendo esta la función de error "fe" que se programa para el método (recuerde que en Javascript el primer índice es 0).

```
>>fe=function(x){return
sqr(x[0]*x[0]+2*x[1]*x[1]-22)+sqr(-2*x[0]*x[0]+x[0]*x[1]-3*x[1]+11);}
function (x){return
sqr(x[0]*x[0]+2*x[1]*x[1]-22)+sqr(-2*x[0]*x[0]+x[0]*x[1]-3*x[1]+11);}
```

Se guardan en variables las constantes del método, se asume los valores iniciales en un vector (x) y del mismo se extrae la dimensión del problema (n):

```
>>a=1.95; b=1.95; d=0.45; k=1.95; x=[1,2]; n=x.length;
2
```

Entonces, con el vector asumido, se genera la matriz "p" con el plan inicial simplex (siguiendo el algoritmo descrito en el acápite respectivo):

```
>>p=new Array(n+1); for(i=0;i<=n;i++) p[i]=x.slice(); for(i=1;i<=n;i++)
p[i][i-1]*=1.1; p
[[1, 2], [1.1, 2], [1, 2.2]]
```

Donde cada fila corresponde a cada uno de los puntos del plan (en este caso 3 puntos, pues se tienen 2 dimensiones).

Para cada uno de los puntos del plan se calcula el valor de la función de error:

```
>>e=new Array(n+1); for(i=0;i<=n;i++) e[i]=fe(p[i]); e
[194, 186.43249999999998, 149.30239999999995]
```

Con estos valores se fijan los índices del menor ("s") y mayor ("h") valores del plan, que en este caso corresponden a 2 y 0 respectivamente (recuerde que en Javascript el primer índice es siempre 0).

```
>>s=2;h=0;
0
```

Entonces se calcula el promedio de los puntos (el centroide "c") aplicando la ecuación (7.9):

```
>>c=new Array(n); for(j=0;j<n;j++){c[j]=0; for(i=0;i<=n;i++) if(i!=h)
c[j]+=p[i][j]; c[j]/=n;} c
[1.05, 2.1]
```

Y se procede a la reflexión, aplicando la ecuación (7.10) (tomando en cuenta que los puntos de estas expresiones son en realidad vectores):

```
>>pr=new Array(n); for(j=0;j<n;j++) pr[j]=a*c[j]+(1-a)*p[h][j]; pr;
[1.0975, 2.195]
```

Para el cual el valor de la función de error es:

```
>>er=fe(pr)
144.0254098094141
```

Y dado que este valor es menor a todos los valores de la función de error, se concluye que la reflexión ha sido muy buena y en consecuencia se procede con la expansión (ecuación (7.11)):

```
>>px=new Array(n); for(j=0;j<n;j++) px[j]=b*pr[j]+(1-b)*c[j]; px
[1.1426249999999998, 2.2852499999999996]
```

Siendo el valor de la función de error para este punto:

```
>>ex=fe(px)
122.23060434912682
```

Y como este valor es menor al de la reflexión (er=144.0...), la expansión ha sido buena, por lo que se reemplaza el punto "p_h" del plan con el punto "px":

```
>>p[h]=px.slice(); p
[[1.1426249999999998, 2.2852499999999996], [1.1, 2], [1, 2.2]]
```

Siendo este el nuevo plan Simplex.

Igualmente se reemplaza el valor de la función de error respectivo:

```
>>e[h]=ex; e
[122.23060434912682, 186.43249999999998, 149.30239999999995]
```

Como el menor valor de la función de error (122.23...) esta aún bastante alejado de cero (que es el mínimo buscado), se repite el proceso y para ello, con los nuevos valores de la función de error, se determinan los nuevos índices de plan: "s" (menor) y "h" (mayor), que en este caso son 0 y 1 respectivamente:

```
>>s=0; h=1;
1
```

Entonces se calcula el nuevo centroide ("c"), se procede con la reflexión y se calcula el valor de la función de error ("er") para este punto:

```
>>for(j=0;j<n;j++){c[j]=0; for(i=0;i<=n;i++) if(i!=h) c[j]+=p[i][j];
c[j]/=n;}; for(j=0;j<n;j++) pr[j]=a*c[j]+(1-a)*p[h][j]; er=fe(pr)
91.15681770076533
```

Una vez más la reflexión es muy buena (es menor a al menor valor de la función de error "e_s=122.2..."), por lo que se procede con la expansión:

```
>>for(j=0;j<n;j++) px[j]=b*pr[j]+(1-b)*c[j]; ex=fe(px)
54.74170263257704
```

Y una vez más la expansión es buena, por lo tanto se reemplaza el punto "p_h" del plan y el valor respectivo de la función de error, por el punto "px" y el valor ex:

```
>>p[h]=px.slice(); e[h]=ex; e
[122.23060434912682, 54.74170263257704, 149.30239999999995]
```

Como el menor valor de la función de error (54.742) no es aún cercano a cero, se repite el proceso.

Los nuevos valores de "s" y "h" son:

```
>>s=1; h=2;
2
```

Procediendo con la reflexión se obtiene:

```
>>for(j=0;j<n;j++){c[j]=0; for(i=0;i<=n;i++) if(i!=h) c[j]+=p[i][j];
c[j]/=n;}; for(j=0;j<n;j++) pr[j]=a*c[j]+(1-a)*p[h][j]; er=fe(pr)
39.541576474495265
```

Una vez más la reflexión es muy buena ("ex" es menor a "e_s"), por lo que se procede con la expansión:

```
>>for(j=0;j<n;j++) px[j]=b*pr[j]+(1-b)*c[j]; ex=fe(px)
11.754040405376578
```

Como la expansión es buena, "px" reemplaza a "p_h":

```
>>p[h]=px.slice(); e[h]=ex; e
[122.23060434912682, 54.74170263257704, 11.754040405376578]
```

Entonces se repite el proceso:

```
>>s=2; h=0;
0
>>for(j=0;j<n;j++){c[j]=0; for(i=0;i<=n;i++) if(i!=h) c[j]+=p[i][j];
c[j]/=n;}; for(j=0;j<n;j++) pr[j]=a*c[j]+(1-a)*p[h][j]; er=fe(pr)
10.050293149723583
```

Como la reflexión es muy buena, se procede con la expansión:

```
>>for(j=0;j<n;j++) px[j]=b*pr[j]+(1-b)*c[j]; ex=fe(px)
99.15313218617443
```

Ahora, sin embargo, la expansión no es buena ("ex" es mayor a "er"), por lo que "pr" reemplaza a "p_h":

```
>>p[h]=pr.slice(); e[h]=er; e
[10.050293149723583, 54.74170263257704, 11.754040405376578]
```

Y como aún el menor valor no es cercano a cero, se repite el proceso:

```
>>s=0; h=1;
1
>>for(j=0;j<n;j++){c[j]=0; for(i=0;i<=n;i++) if(i!=h) c[j]+=p[i][j];
c[j]/=n;}; for(j=0;j<n;j++) pr[j]=a*c[j]+(1-a)*p[h][j]; er=fe(pr)
53.509699294384355
```

Como la reflexión es buena ("er" es ligeramente menor a "e_h") "pr" reemplaza a "p_h":

```
>>p[h]=pr.slice(); e[h]=er; e
[10.050293149723583, 53.509699294384355, 11.754040405376578]
```

Continuando de esta forma, unas cuantas iteraciones más, se obtiene:

```
>>s=0; h=1;
1
>>for(j=0;j<n;j++){c[j]=0; for(i=0;i<=n;i++) if(i!=h) c[j]+=p[i][j];
c[j]/=n;}; for(j=0;j<n;j++) pr[j]=a*c[j]+(1-a)*p[h][j]; er=fe(pr)
46.70733814239368
>>p[h]=pr.slice(); e[h]=er; e
[10.050293149723583, 46.70733814239368, 11.754040405376578]

>>s=0; h=1;
1
>>for(j=0;j<n;j++){c[j]=0; for(i=0;i<=n;i++) if(i!=h) c[j]+=p[i][j];
```

```
c[j]/=n;}; for(j=0;j<n;j++) pr[j]=a*c[j]+(1-a)*p[h][j]; er=fe(pr)
43.60343922407818
>>p[h]=pr.slice(); e[h]=er; e
[10.050293149723583, 43.60343922407818, 11.754040405376578]

>>s=0; h=1;
1
>>for(j=0;j<n;j++){c[j]=0; for(i=0;i<=n;i++) if(i!=h) c[j]+=p[i][j];
c[j]/=n;}; for(j=0;j<n;j++) pr[j]=a*c[j]+(1-a)*p[h][j]; er=fe(pr)
39.966670379285766
>>p[h]=pr.slice(); e[h]=er; e
[10.050293149723583, 39.966670379285766, 11.754040405376578]

>>s=0; h=1;
1
>>for(j=0;j<n;j++){c[j]=0; for(i=0;i<=n;i++) if(i!=h) c[j]+=p[i][j];
c[j]/=n;}; for(j=0;j<n;j++) pr[j]=a*c[j]+(1-a)*p[h][j]; er=fe(pr)
35.73485389682637
>>p[h]=pr.slice(); e[h]=er; e
[10.050293149723583, 35.73485389682637, 11.754040405376578]

>>s=0; h=1;
1
>>for(j=0;j<n;j++){c[j]=0; for(i=0;i<=n;i++) if(i!=h) c[j]+=p[i][j];
c[j]/=n;}; for(j=0;j<n;j++) pr[j]=a*c[j]+(1-a)*p[h][j]; er=fe(pr)
34.3227001316411
>>p[h]=pr.slice(); e[h]=er; e
[10.050293149723583, 34.3227001316411, 11.754040405376578]
```

Como se puede ver el método va convergiendo lentamente, después de 10 iteraciones aún está bastante alejado de cero. Esa es una característica del método y la razón por la cual no se aplica manualmente, se requieren cientos o miles de iteraciones para lograr resultados aceptables.

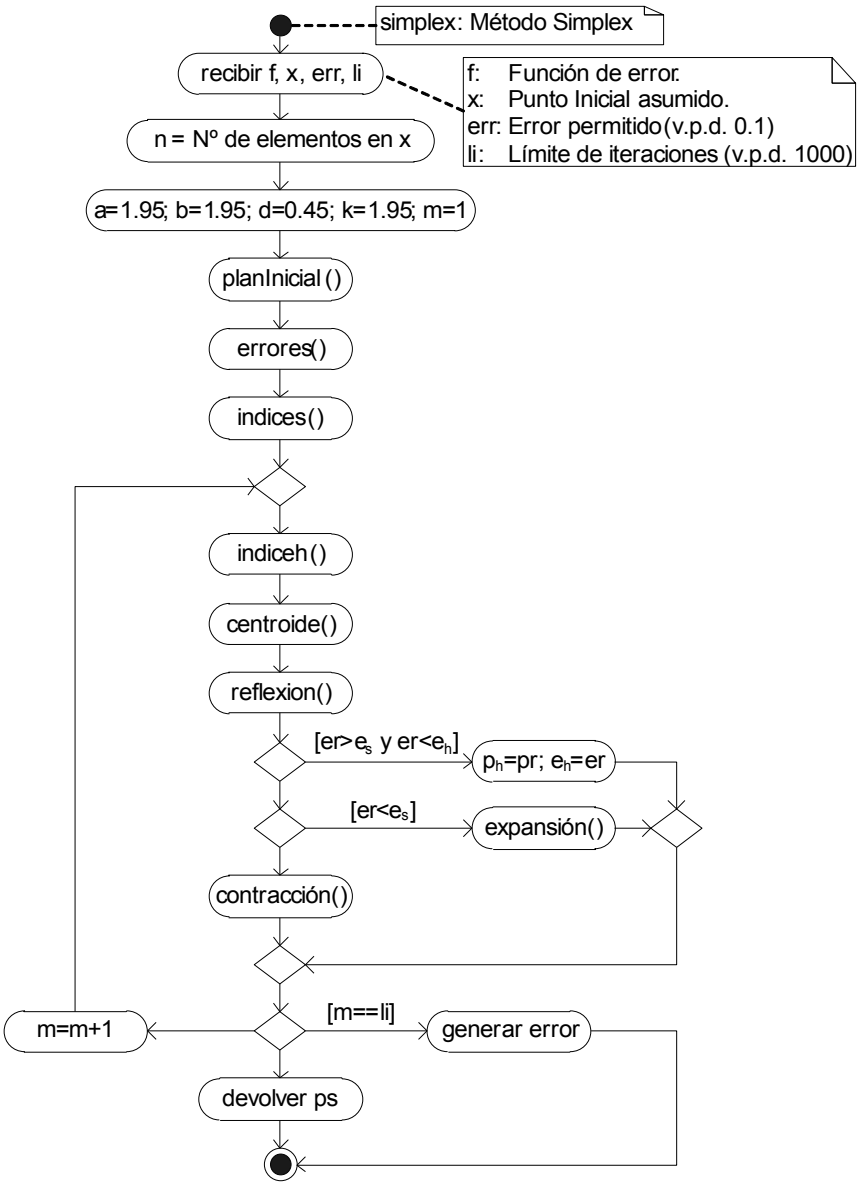
Hasta el punto donde se ha iterado la mejor aproximación (la correspondiente al índice "s") es:

```
>>p[s]
[1.1058199346630857, 3.401640470888672]
```

Es decir que hasta el momento las soluciones serían: x_1=x=1.1058... y x_2=y=3.4016..., valores que todavía están lejos de las soluciones correctas (x=2, y=3).

7.1.10. Algoritmo y código

El algoritmo que automatiza la aplicación del método se presenta en la siguiente página y en la misma "planInicial" es la función donde se calculan los puntos del plan inicial ("p"), "errores" es la función donde se calculan los valores de la función de error para los puntos del plan simplex ("e"), "indices" es la función donde se determina el índice del menor valor de la función de error ("s"), "indiceh" es la función donde se determina el índice del mayor valor de la función de error ("h"), "centroide" es la función donde se calcula el centroide del plan simplex ("c"), "reflexión" es la función donde se realiza la reflexión ("pr" y "er"), "expansión" es la función donde se realiza la reflexión ("px" y "ex") y "contraccion" es la función donde se realiza la contracción y desde la cual se llama a "escalamiento" en caso de que la contracción sea mala.



El código respectivo, añadido a la librería "mat205.js", es:

```
function simplex(f,x,err,li) {
  if (x==null)
    throw new Error("Valores iniciales no asumidos (simplex).");
  if (err==null) err=0.1;
  if (li==null) li=1000;
  var i,j,h,s,s1,er,ex,ec,a=1.95,b=1.95,d=0.45,k=1.95,m=1;
  var n=x.length,p=new Array(n+1),e=new Array(n+1);
  var c=new Array(n),pr=new Array(n),px=new Array(n);
  var pc=new Array(n);err=sqr(err);
  function planInicial() {
    for (i=0;i<=n;i++)
      p[i]=x.slice(0);
    for (i=0;i<n;i++)
      p[i+1][i]*=1.1;
```

```
}
function errores() {
  for (i=0;i<=n;i++)
    e[i]=f(p[i]);
}
function indiceh() {
  h=0;
  for (i=1;i<=n;i++)
    if (e[i]>e[h]) h=i;
}
function indices() {
  s=0;
  for (i=1;i<=n;i++)
    if (e[i]<e[s]) s=i;
}
function centroide() {
  for (j=0;j<n;j++) {
    c[j]=0;
    for (i=0;i<=n;i++)
      if (i!=h) c[j]+=p[i][j];
    c[j]/=n;
  }
}
function reflexion() {
  for (j=0;j<n;j++)
    pr[j]=a*c[j]+(1-a)*p[h][j];
  er=f(pr);
}
function expansion() {
  for (j=0;j<n;j++)
    px[j]=b*pr[j]+(1-b)*c[j];
  ex=f(px);
  if (ex<er)
    {p[h]=px.slice(); e[h]=ex;}
  else
    {p[h]=pr.slice(); e[h]=er;}
  s=h;
}
function escalamiento(pla) {
  for (i=0;i<=n;i++)
    if (i!=s)
      for (j=0;j<n;j++)
        p[i][j]=k*p[s][j]+(1-k)*p[i][j];
  errores();
  indices();
}
function contraccion() {
  for (j=0;j<n;j++)
    pc[j]=d*c[j]+(1-d)*p[h][j];
```

```
      ec=f(pc);
      if (ec<e[h])
        {p[h]=pc.slice(0); e[h]=ec;}
      else
      else
        escalamiento();
    }
    planInicial();
    errores()
    indices();
    while (true) {
      indiceh();
      centroide();
      reflexion();
      if (er>e[s] && er<e[h])
        {p[h]=pr.slice(0); e[h]=er;}
      else if (er<e[s])
        expansion();
      else
        contraccion();
      if (e[s]<err){ alert("Número de iteraciones: "+m);
        return p[s];}
      if (m++ == li)
        throw new Error("Convergencia no lograda (simplex).");
    }
  }
```

Haciendo correr el programa con el sistema del ejemplo manual, con el error y límite de iteraciones por defecto, se obtiene:

```
>>fe=function(x){return
sqr(x[0]*x[0]+2*x[1]*x[1]-22)+sqr(-2*x[0]*x[0]+x[0]*x[1]-3*x[1]+11);};
simplex(fe,[1,2])
[2.00113988101627, 2.9921815440261015]
```

Que son soluciones cercanas a las correctas. Haciendo correr el programa con un error igual a 1e-9 (y mostrando el resultado con 9 dígitos de precisión), se obtienen:

```
>>precision(simplex(fe,[1,2],1e-9),9)
[2, 3]
```

Que, con la precisión empleada, son los resultados exactos.

7.2. MÉTODO DEL DESCENSO ACELERADO

Al igual que Simplex, el método del descenso acelerado es un método de optimización, por lo tanto se trabaja con la función de error:

$$e(x_1, x_2, x_3, \cdots, x_n) = \left[f(x_1, x_2, x_3, \cdots, x_n) - \alpha \right]^2 = 0 \qquad (7.4)$$

Pero la dirección a seguir se determina no por extrapolación multidimensional, sino mediante el gradiente de la función de error, definida por la siguiente ecuación:

$$\vec{z}=\nabla fe(\vec{x})=\begin{bmatrix}\frac{\partial fe(\vec{x})}{\partial x_1}\\ \frac{\partial fe(\vec{x})}{\partial x_2}\\ \cdots\\ \frac{\partial fe(\vec{x})}{\partial x_n}\end{bmatrix} \tag{7.16}$$

Para acercarse al mínimo, se resta a los valores iniciales asumidos (el vector inicial) el gradiente de la función multiplicado por un factor α:

$$\vec{x}_n=\vec{x}-\alpha\cdot\vec{z} \tag{7.17}$$

El valor de α se determina de manera que, con dicho valor, se minimice el valor de la función en la dirección de la gradiente. En este tema α se calcula normalizando primero los valores del vector "z" (para que su suma sea 1) y probando luego valores de alfa que comienzan en 1 y que en iteraciones sucesivas disminuye a la mitad del valor anterior, hasta que la función de error es menor al valor de la función de error en el punto asumido. Entonces se toma un valor de α igual a la mitad del valor encontrado y con el mismo se calcula el valor de la función. Posteriormente, con el punto inicial y los dos puntos calculados con α y $\alpha/2$, se encuentra el mínimo de la curva formada por esos tres puntos, igualando a cero la derivada primera de la ecuación de segundo grado resultante de la interpolación polinomial.

Para el cálculo del gradiente, el principal problema radica en el cálculo de las derivadas parciales. Si bien con funciones sencillas es posible trabajar con las derivadas analíticas, en la mayoría de los casos esa alternativa no es práctica, por un lado porque consume mucho tiempo, y por otro, porque sería necesario repetir dicho procedimiento para cada nuevo sistema de ecuaciones.

Para elaborar un programa que sea de utilidad práctica, el cálculo de las derivadas parciales debe ser numérico y para ello se empleará la fórmula de diferencia central de segundo orden (empleada ya en el método de Newton) adaptada para el cálculo de la derivada parcial:

$$\frac{\partial fe(\vec{x})}{\partial x_i}=\frac{fe(x_1,x_2,\cdots,x_i+h,\cdots,x_{n-1},x_n)-fe(x_1,x_2,\cdots,x_i-h,\cdots,x_{n-1},x_n)}{2h} \tag{7.18}$$

Donde el valor de "h" se calcula con: $h=x_i\cdot 10^{-6}$.

El proceso comienza calculando el valor de la función de error para los valores iniciales asumidos (vector "x"):

$$fe_1=fe(\vec{x}) \tag{7.19}$$

Para determinar la dirección a seguir se calcula "z" y con el mismo el valor absoluto de "z":

$$z_0=|\vec{z}| \tag{7.20}$$

Si este valor es cero, el proceso concluye, siendo las soluciones los valores del vector "x", caso contrario, se normalizan los valores de "z":

$$\vec{z}=\frac{\vec{z}}{z_0} \tag{7.21}$$

Se fija a_3 (valor de α) en 1 y con el mismo (y el valor del vector "z") se calcula el nuevo vector de prueba y los respectivos valores de la función de error:

$$\vec{x}_3=\vec{x}-a_3\vec{z}$$
$$fe_3=fe(\vec{x}_3) \tag{7.22}$$

Si "fe_3" no es menor a "fe_1", "a_3" se divide entre dos ($a_3=a_3/2$) y se vuelven a calcular los valores de "x_3" y "fe_3", repitiendo si es necesario hasta que "fe_3" sea menor a "fe_1" o hasta que "a_3" sea menor al error permitido, en cuyo caso el proceso concluye, siendo las soluciones los valores actuales de "x".

Una vez que "fe_3" es menor a "fe_1", se calcula un nuevo valor de α extrapolado "a0" y para ello se calculan los siguientes parámetros:

$$\vec{x}_2=\vec{x}-a_2\vec{z}$$
$$fe_2=fe(\vec{x}_2) \tag{7.23}$$

$$h_1=\frac{fe_2-fe_0}{a_2} \tag{7.24}$$

$$h_2=\frac{fe_3-fe_2}{a_3-a_2} \tag{7.25}$$

$$h_3=\frac{h_2-h_1}{a_3} \tag{7.26}$$

Y con los mismos se calcula el valor del parámetro "a_0":

$$a_0=0.5\cdot\left(a_2-\frac{h_1}{h_3}\right) \tag{7.27}$$

Con este valor (α) se calcula un nuevo valor de prueba y el correspondiente valor de la función de error:

$$\vec{x}_0=\vec{x}-a_0\vec{z}$$
$$fe_0=fe(\vec{x}_0) \tag{7.28}$$

Si "fe_0" es menor a "fe_3", los nuevos valores de prueba son los valores del vector "x_0", caso contrario los nuevos valores de prueba son los valores del vector "x_3".

7.2.1. Ejemplo manual

Para comprender mejor el proceso se aplicará el método al sistema de ecuaciones (7.15), por lo tanto, la función de error es la misma que en el método Simplex:

```
>>fe=function(x){return
sqr(x[0]*x[0]+2*x[1]*x[1]-22)+sqr(-2*x[0]*x[0]+x[0]*x[1]-3*x[1]+11);};
function (x){return
sqr(x[0]*x[0]+2*x[1]*x[1]-22)+sqr(-2*x[0]*x[0]+x[0]*x[1]-3*x[1]+11);}
```

Puesto que en este método se deben calcular las derivadas parciales, se crea una función para ese fin:

```
>>dp1=function(f,x,i){h=x[i]*1e-6; x[i]+=h; f2=f(x); x[i]-=2*h; f1=f(x);
```

```
x[i]+=h; return (f2-f1)/(2*h);}
function (f,x,i){h=x[i]*1e-6; x[i]+=h; f2=f(x); x[i]-=2*h; f1=f(x);
x[i]+=h; return (f2-f1)/(2*h);}
```

Los valores iniciales asumidos, serán los mismos que en el método Simplex es decir $x=x_1=1$ y $y=x_2=2$ y con la dimensión del vector se crean los otros vectores necesarios para el método:

```
>>x=[1,2]; n=x.length; z=new Array(n); x0=new Array(n); x2=new Array(n);
x3=new Array(n);
```

Con los valores asumidos (el vector "x") se calcula el valor de la función de error:

```
>>fe1=fe(x)
194
```

Como este valor no es cercano a cero se determina la dirección a seguir con el gradiente de la función de error:

```
>>for(i=0;i<n;i++) z[i]=dp1(fe,x,i); z
[-72.00000001716944, -227.99999999989495]
```

Y se calcula su valor absoluto (que también puede ser calculado con "abs"):

```
>>z0=0; for(i=0;i<n;i++) z0+=z[i]*z[i]; z0=sqrt(z0)
239.09830614712538
```

Con este valor se normalizan los valores de "z":

```
>>for(i=0;i<n;i++) z[i]/=z0; z
[-0.301131368002521, -0.9535826651134816]
```

Ahora se inicia a3 en 1 y se calculan el vector x3 y el valor de la función de error para x3:

```
>>a3=1; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
14.91931784100977
```

Como "fe3" ya es menor a "fe1", se pasa al cálculo del parámetro "a0", el vector "x0" y el valor de la función de error para "x0" (fe0):

```
>>a2=a3/2; for(i=0;i<n;i++) x2[i]=x[i]-a2*z[i]; fe2=fe(x2);
84.90712615356169

>>h1=(fe2-fe1)/a2; h2=(fe3-fe2)/(a3-a2); h3=(h2-h1)/a3;
78.21013106777278

>>a0=(a2-h1/h3)/2; for(i=0;i<n;i++) x0[i]=x[i]-a0*z[i]; fe0=fe(x0)
33.88509476511371
```

Entonces se compara "fe3" con "fe0" y como "fe3" es menor, se toman como nuevos valores de prueba los valores de "x3":

```
>>for(i=0;i<n;i++) x[i]=x3[i]; x
[1.301131368002521, 2.9535826651134816]
>>fe1=fe3
14.91931784100977
```

Ahora (puesto que fe1 está todavía bastante lejos de cero) se repite el proceso, calculando el nuevo valor de "z0":

```
>>for(i=0;i<n;i++) z[i]=dp1(fe,x,i); z0=0; for(i=0;i<n;i++)
```

```
z0+=z[i]*z[i]; z0=sqrt(z0)
80.8833842126411
```

Y como este valor no es cero, se continúa con el proceso, calculando el nuevo valor de "fe3":

```
>>for(i=0;i<n;i++) z[i]/=z0; a3=1; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i];
fe3=fe(x3)
122.12999165223934
```

Ahora "fe3" no es menor a "fe1" por lo que se reduce "a3" hasta que el valor de "fe3" sea menor a "fe1":

```
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
15.200975830465419
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
4.133845613737937
```

Ahora que "fe3" es menor a "fe1", se continúa con el proceso calculando los nuevos valores de "a0" y "fe0":

```
>>a2=a3/2; for(i=0;i<n;i++) x2[i]=x[i]-a2*z[i]; fe2=fe(x2);h1=(fe2-
fe1)/a2; h2=(fe3-fe2)/(a3-a2); h3=(h2-h1)/a3; a0=(a2-h1/h3)/2;
for(i=0;i<n;i++) x0[i]=x[i]-a0*z[i]; fe0=fe(x0)
4.130488785735921
```

Como "fe0" es menor a "fe3" los nuevos valores de prueba son los correspondientes al vector "x0":

```
>>for(i=0;i<n;i++) x[i]=x0[i]; x
[1.3874032193290886, 3.201610541804361]
>>fe1=fe0
4.130488785735921
```

Entonces se repite el proceso (pues "fe1" está todavía lejos de cero):

```
>>for(i=0;i<n;i++) z[i]=dp1(fe,x,i); z0=0; for(i=0;i<n;i++)
z0+=z[i]*z[i]; z0=sqrt(z0)
8.291359302596135
```

Como "z0" tampoco es cero, el proceso continúa calculando los nuevos valores de "fe3" y "fe0":

```
>>for(i=0;i<n;i++) z[i]/=z0; a3=1; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i];
fe3=fe(x3)
9.271086618967068
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
3.347154257760634
>>a2=a3/2; for(i=0;i<n;i++) x2[i]=x[i]-a2*z[i]; fe2=fe(x2);h1=(fe2-
fe1)/a2; h2=(fe3-fe2)/(a3-a2); h3=(h2-h1)/a3; a0=(a2-h1/h3)/2;
for(i=0;i<n;i++) x0[i]=x[i]-a0*z[i]; fe0=fe(x0)
2.918543179344603
```

Y como "fe0" es menor a "fe3", "x0" son los nuevos valores de prueba:

```
>>for(i=0;i<n;i++) x[i]=x0[i]; x
[1.6503985926268574, 3.032255030345631]
>>fe1=fe0
2.918543179344603
```

Continuando con algunas iteraciones más, se obtiene:

```
>>for(i=0;i<n;i++) z[i]=dp1(fe,x,i); z0=0; for(i=0;i<n;i++)
z0+=z[i]*z[i]; z0=sqrt(z0)
30.218316090359753
>>for(i=0;i<n;i++) z[i]/=z0; a3=1; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i];
fe3=fe(x3)
167.32162934681529
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
30.688748570872825
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
5.396253147179007
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
1.565534839747958
>>a2=a3/2; for(i=0;i<n;i++) x2[i]=x[i]-a2*z[i]; fe2=fe(x2);h1=(fe2-
fe1)/a2; h2=(fe3-fe2)/(a3-a2); h3=(h2-h1)/a3; a0=(a2-h1/h3)/2;
for(i=0;i<n;i++) x0[i]=x[i]-a0*z[i]; fe0=fe(x0)
1.4360231975707145
>>for(i=0;i<n;i++) x[i]=x0[i]; x
[1.7025429290140204, 3.113805936551593]
>>fe1=fe0
1.4360231975707145

>>for(i=0;i<n;i++) z[i]=dp1(fe,x,i); z0=0; for(i=0;i<n;i++)
z0+=z[i]*z[i]; z0=sqrt(z0)
7.845688595651542
>>for(i=0;i<n;i++) z[i]/=z0; a3=1; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i];
fe3=fe(x3)
14.832861033302716
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
1.9452523896886578
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.5335082677647153
>>a2=a3/2; for(i=0;i<n;i++) x2[i]=x[i]-a2*z[i]; fe2=fe(x2);h1=(fe2-
fe1)/a2; h2=(fe3-fe2)/(a3-a2); h3=(h2-h1)/a3; a0=(a2-h1/h3)/2;
for(i=0;i<n;i++) x0[i]=x[i]-a0*z[i]; fe0=fe(x0)
0.5259460853653797
>>for(i=0;i<n;i++) x[i]=x0[i]; x
[1.8970843179392562, 2.98998559550296]
>>fe1=fe0
0.5259460853653797

>>for(i=0;i<n;i++) z[i]=dp1(fe,x,i); z0=0; for(i=0;i<n;i++)
z0+=z[i]*z[i]; z0=sqrt(z0)
16.06699416744283
>>for(i=0;i<n;i++) z[i]/=z0; a3=1; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i];
fe3=fe(x3)
186.67315395139562
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
37.19090200059564
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
6.986518585047353
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
1.051902735041507
```

```
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.14485189920230476
>>a2=a3/2; for(i=0;i<n;i++) x2[i]=x[i]-a2*z[i]; fe2=fe(x2);h1=(fe2-
fe1)/a2; h2=(fe3-fe2)/(a3-a2); h3=(h2-h1)/a3; a0=(a2-h1/h3)/2;
for(i=0;i<n;i++) x0[i]=x[i]-a0*z[i]; fe0=fe(x0)
0.12003414303472591
>>for(i=0;i<n;i++) x[i]=x0[i]; x
[1.9239349723160748, 3.03240651998964]
>>fe1=fe0
0.12003414303472591

>>for(i=0;i<n;i++) z[i]=dp1(fe,x,i); z0=0; for(i=0;i<n;i++)
z0+=z[i]*z[i]; z0=sqrt(z0)
2.8455927565816963
>>for(i=0;i<n;i++) z[i]/=z0; a3=1; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i];
fe3=fe(x3)
27.55895112936383
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
4.766815592421047
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.7990787799219695
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.09987318050188494
>>a2=a3/2; for(i=0;i<n;i++) x2[i]=x[i]-a2*z[i]; fe2=fe(x2);h1=(fe2-
fe1)/a2; h2=(fe3-fe2)/(a3-a2); h3=(h2-h1)/a3; a0=(a2-h1/h3)/2;
for(i=0;i<n;i++) x0[i]=x[i]-a0*z[i]; fe0=fe(x0)
0.024419055889811487
>>for(i=0;i<n;i++) x[i]=x0[i]; x
[1.9798172565265588, 2.996916919245346]
>>fe1=fe0
0.024419055889811487

>>for(i=0;i<n;i++) z[i]=dp1(fe,x,i); z0=0; for(i=0;i<n;i++)
z0+=z[i]*z[i]; z0=sqrt(z0)
3.5948481555389744
>>for(i=0;i<n;i++) z[i]/=z0; a3=1; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i];
fe3=fe(x3)
203.40415438988762
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
44.08914043813528
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
9.890408698264455
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
2.1807811755188533
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.4406226116636027
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.0709903331707672
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.007813884522188076
>>a2=a3/2; for(i=0;i<n;i++) x2[i]=x[i]-a2*z[i]; fe2=fe(x2);h1=(fe2-
fe1)/a2; h2=(fe3-fe2)/(a3-a2); h3=(h2-h1)/a3; a0=(a2-h1/h3)/2;
for(i=0;i<n;i++) x0[i]=x[i]-a0*z[i]; fe0=fe(x0)
```

```
0.004459255396000325
>>for(i=0;i<n;i++) x[i]=x0[i]; x
[1.98581572845245, 3.0062383067775453]
>>fe1=fe0
0.004459255396000325

>>for(i=0;i<n;i++) z[i]=dp1(fe,x,i); z0=0; for(i=0;i<n;i++)
z0+=z[i]*z[i]; z0=sqrt(z0)
0.5791231206685721
>>for(i=0;i<n;i++) z[i]/=z0; a3=1; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i];
fe3=fe(x3)
32.716740392718805
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
6.391736409749074
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
1.3837484981056918
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.29873215705611833
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.05837237173219989
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.008708844599962694
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.0009756917912747144
>>a2=a3/2; for(i=0;i<n;i++) x2[i]=x[i]-a2*z[i]; fe2=fe(x2);h1=(fe2-
fe1)/a2; h2=(fe3-fe2)/(a3-a2); h3=(h2-h1)/a3; a0=(a2-h1/h3)/2;
for(i=0;i<n;i++) x0[i]=x[i]-a0*z[i]; fe0=fe(x0)
0.0007784074370660529
>>for(i=0;i<n;i++) x[i]=x0[i]; x
[1.996504656996256, 2.9993920336808366]
>>fe1=fe0
0.0007784074370660529

>>for(i=0;i<n;i++) z[i]/=z0; a3=1; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i];
fe3=fe(x3)
207.7322646671023
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
45.870694906341875
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
10.68397511999231
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
2.545275097852366
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.606039883990474
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.1406396662776399
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.030510144792432758
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.005656008015171822
>>a3/=2; for(i=0;i<n;i++) x3[i]=x[i]-a3*z[i]; fe3=fe(x3)
0.0007278082028298814
>>a2=a3/2; for(i=0;i<n;i++) x2[i]=x[i]-a2*z[i]; fe2=fe(x2);h1=(fe2-
```

```
fe1)/a2; h2=(fe3-fe2)/(a3-a2); h3=(h2-h1)/a3; a0=(a2-h1/h3)/2;
for(i=0;i<n;i++) x0[i]=x[i]-a0*z[i]; fe0=fe(x0)
0.00013145147212259567
>>for(i=0;i<n;i++) x[i]=x0[i]; x
[1.9975793772718815, 3.0010702969439027]
>>fe1=fe0
0.00013145147212259567
```

Si se detiene el proceso en esta iteración las soluciones serían: "x=1.99757..." y "y=3.00107...", que son bastante cercanas a las soluciones exactas ("x=2", "y=3"). Estos resultados se consiguen en 10 iteraciones, mientras que con Simplex en 10 iteraciones los resultados apenas empiezan a acercarse a las soluciones correctas.

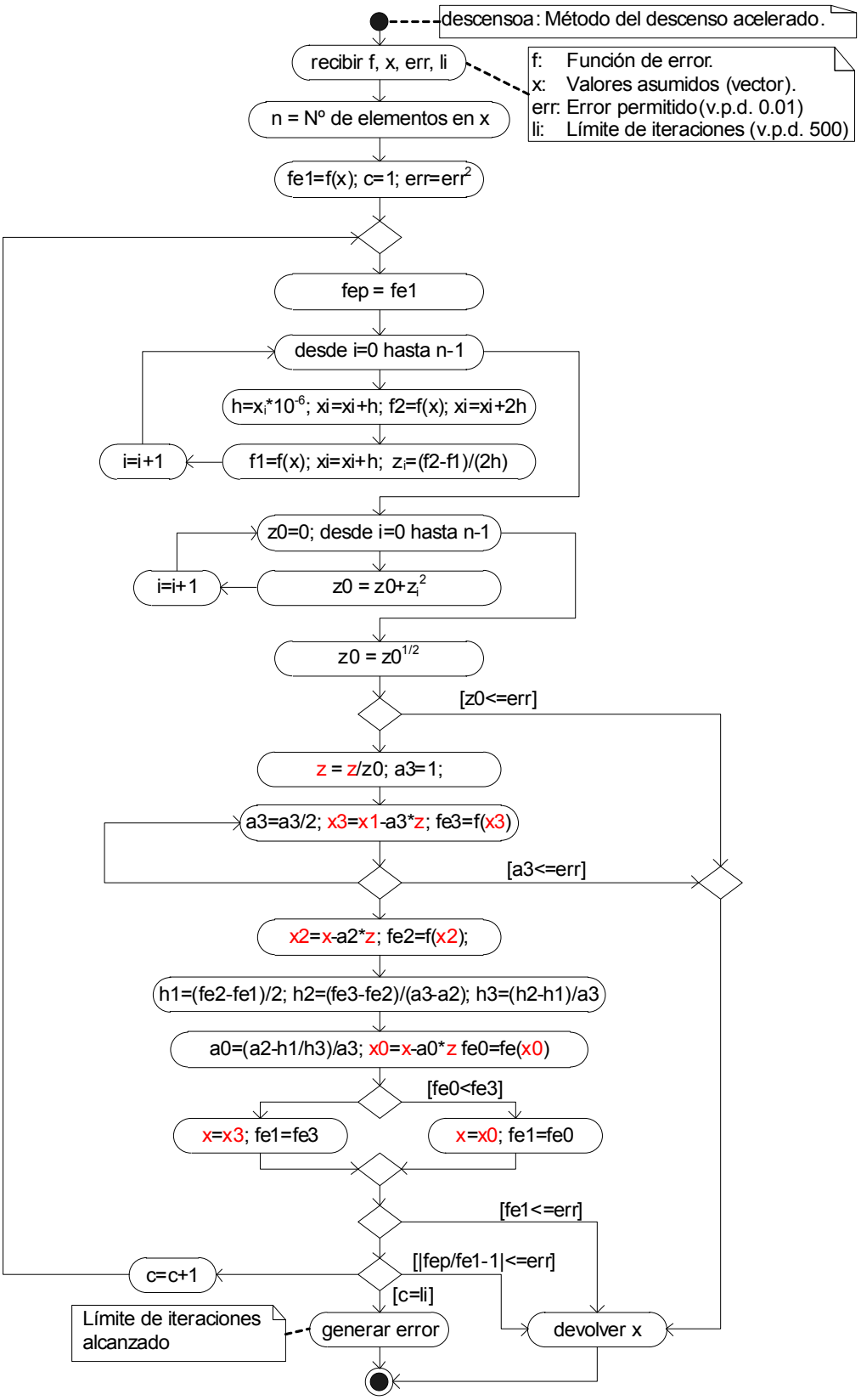
Sin embargo, el método del descenso acelerado no siempre requiere menos iteraciones que el método Simplex, en ocasiones suele requerir muchas más iteraciones, pero además suele ser menos estable (es decir que el método Simplex logra convergencia en la mayoría de los casos, mientras que el método del descenso acelerado no).

7.2.2. Algoritmo y código

El algoritmo que automatiza el proceso se presenta en la siguiente página, en el mismo se han empleado variables en rojo para diferenciar las operaciones con vectores de las operaciones con escalares.

El código respectivo, añadido a la librería "mat205.js", es:

```
function descensoa(f,x,err,li) {
  if (x==null)
    throw new Error("Valores iniciales no asumidos (descensoa).");
  if (err==null) err=0.01;
  if (li==null) li=500;
  var n=x.length,z=new Array(n),x0=new Array(n);
  var x2=new Array(n),x3=new Array(n);
  var a0,a2,a3,fe0,fe1,fe2,fe3,fep;
  var f1,f2,i,h1,h2,h3,z0;
  var fe1=f(x), c=1, err=sqr(err);
  do {
    //Penúltimo valor de la función de error
    fep=fe1;
    //Cálculo de las derivadas parciales (valores de "z")
    for (i=0;i<n;i++){
      h=x[i]!=0 ? x[i]*1e-6 : 1e-6;
      x[i]+=h; f2=f(x);
      x[i]-=2*h; f1=f(x);
      x[i]+=h;
      z[i]=(f2-f1)/(2*h);
    }
    //Valor absoluto de "z"
    z0=0;
    for (i=0;i<n;i++)
      z0+=z[i]*z[i];
    z0=sqrt(z0);
```




```
//Si z0 es menor al error, el proceso termina
if (z0<err) return x;
//Normalización de z
```

```
    for (i=0;i<n;i++)
      z[i]/=z0;
    //Cálculo de x3 y la función de error para x3
    a3=2;
    do {
      a3/=2;
      for (i=0;i<n;i++)
        x3[i]=x[i]-a3*z[i];
      fe3=f(x3);
    } while (fe3>fe1 && a3>err);
    //si a3 es menor al error, el proceso concluye
    if (a3<err) return x3;
    //Cálculo de x0 y la función de error para x0
    a2=a3/2;
    for (i=0;i<n;i++)
      x2[i]=x[i]-a2*z[i];
    fe2=f(x2);
    h1=(fe2-fe1)/a2;
    h2=(fe3-fe2)/(a3-a2);
    h3=(h2-h1)/a3;
    a0=(a2-h1/h3)/2;
    for (i=0;i<n;i++)
      x0[i]=x[i]-a0*z[i];
    fe0=f(x0);
    //Selección de los nuevos valores de prueba
    if (fe0<fe3)
      {for (i=0;i<n;i++) x[i]=x0[i]; fe1=fe0;}
    else
      {for (i=0;i<n;i++) x[i]=x3[i]; fe1=fe3;}
    //Si la función de error es menor al error, el proceso termina
    if (fe1<err) return x;
    //Si las funciones de error de las dos últimas iteraciones son
    //casi iguales, el proceso concluye
    if (abs(fep/fe1-1)<err) return x;
  } while(c++ != li);
  throw new Error("No se logra convergencia (descensoa).");
}
```

Haciendo correr el programa para el sistema del ejemplo manual (con el error y límite de iteraciones por defecto) se obtiene:

```
>>fe=function(x){return
sqr(x[0]*x[0]+2*x[1]*x[1]-22)+sqr(-2*x[0]*x[0]+x[0]*x[1]-3*x[1]+11);};
descensoa(fe,[1,2])
[1.999420703553801, 2.9998953704234]
```

Que mostrado con 3 dígitos de precisión es:

```
>>precision(ans,3)
[2, 3]
```

Que son los resultados exactos.

7.3. MÉTODO DE NEWTON PARA SISTEMAS DE ECUACIONES NO LINEALES

Los métodos Simplex y del descenso acelerado se emplean generalmente sólo para obtener valores iniciales (debido al elevado número de iteraciones que requieren).

Una vez que se cuenta con valores iniciales adecuados se puede proseguir con un método más eficiente. Uno de dichos métodos es el método de Newton para sistemas de ecuaciones no lineales.

En este método se expande el sistema de ecuaciones lineales empleando series de Taylor. Por ejemplo, si en el sistema se tienen 3 ecuaciones no lineales con 3 incógnitas:

$$\begin{aligned} f_1(x_1, x_2, x_3) &= 0 \\ f_3(x_1, x_2, x_3) &= 0 \\ f_2(x_1, x_2, x_3) &= 0 \end{aligned} \tag{7.29}$$

La expansión en series de Taylor resulta:

$$\begin{aligned} f_1(x_1+\Delta x_1, x_2+\Delta x_2, x_3+\Delta x_3) &= f_1+\frac{\partial f_1}{\partial x_1}\Delta x_1+\frac{\partial f_1}{\partial x_2}\Delta x_2+\frac{\partial f_1}{\partial x_3}\Delta x_3+\cdots\infty=0 \\ f_2(x_1+\Delta x_1, x_2+\Delta x_2, x_3+\Delta x_3) &= f_2+\frac{\partial f_2}{\partial x_1}\Delta x_1+\frac{\partial f_2}{\partial x_2}\Delta x_2+\frac{\partial f_2}{\partial x_3}\Delta x_3+\cdots\infty=0 \\ f_3(x_1+\Delta x_1, x_2+\Delta x_2, x_3+\Delta x_3) &= f_3+\frac{\partial f_3}{\partial x_1}\Delta x_1+\frac{\partial f_3}{\partial x_2}\Delta x_2+\frac{\partial f_3}{\partial x_3}\Delta x_3+\cdots\infty=0 \end{aligned} \tag{7.30}$$

Donde Δx_1, Δx_2 y Δx_3 son los valores que deberían añadirse a los valores asumidos (x_1, x_2 y x_3) para que las funciones (f_1, f_2 y f_3) se hagan cero. En otras palabras si fuera posible calcular los valores de Δx_1, Δx_2 y Δx_3, las soluciones del sistema serían $y_1 = x_1+\Delta x_1$, $y_2 = x_2+\Delta x_2$ y $y_3 = x_3+\Delta x_3$.

Por supuesto al tratarse de series infinitas, no es posible en la práctica calcular dichos valores, sin embargo, sí es posible obtener una aproximación de los mismos tomando en cuenta sólo los términos de primer orden:

$$\begin{aligned} f_1(x_1+\Delta x_1, x_2+\Delta x_2, x_3+\Delta x_3) &\approx f_1+\frac{\partial f_1}{\partial x_1}\Delta x_1+\frac{\partial f_1}{\partial x_2}\Delta x_2+\frac{\partial f_1}{\partial x_3}\Delta x_3=0 \\ f_2(x_1+\Delta x_1, x_2+\Delta x_2, x_3+\Delta x_3) &\approx f_2+\frac{\partial f_2}{\partial x_1}\Delta x_1+\frac{\partial f_2}{\partial x_2}\Delta x_2+\frac{\partial f_2}{\partial x_3}\Delta x_3=0 \\ f_3(x_1+\Delta x_1, x_2+\Delta x_2, x_3+\Delta x_3) &\approx f_3+\frac{\partial f_3}{\partial x_1}\Delta x_1+\frac{\partial f_3}{\partial x_2}\Delta x_2+\frac{\partial f_3}{\partial x_3}\Delta x_3=0 \end{aligned}$$

$$\begin{aligned} \frac{\partial f_1}{\partial x_1}\Delta x_1+\frac{\partial f_1}{\partial x_2}\Delta x_2+\frac{\partial f_1}{\partial x_3}\Delta x_3 &= -f_1 \\ \frac{\partial f_2}{\partial x_1}\Delta x_1+\frac{\partial f_2}{\partial x_2}\Delta x_2+\frac{\partial f_2}{\partial x_3}\Delta x_3 &= -f_2 \\ \frac{\partial f_3}{\partial x_1}\Delta x_1+\frac{\partial f_3}{\partial x_2}\Delta x_2+\frac{\partial f_3}{\partial x_3}\Delta x_3 &= -f_3 \end{aligned} \tag{7.31}$$

De esta manera se forma un sistema de 3 ecuaciones lineales con 3 incógnitas: Δx_1, Δx_2 y Δx_3. No obstante estos valores son sólo aproximados, por lo que el proceso de cálculo se torna iterativo: comenzando con los valores asumidos x_1 a x_n, se calculan los valores de Δx_1, Δx_2 y Δx_3 (resolviendo el

sistema de ecuaciones lineales). Si estos valores son diferentes de cero, se calculan nuevos valores de prueba: $x_1=x_1+\Delta x_1$, $x_2=x_2+\Delta x_2$; $x_3=x_3+\Delta x_3$ y el proceso se repite, empleando estos nuevos valores, hasta que los valores Δx_i son cero (o casi cero) o hasta que los valores de "x" de dos iteraciones sucesivas son iguales (o casi iguales).

El cálculo de las derivadas parciales se realiza con la ecuación (7.18).

7.3.1. Ejemplo manual

Para comprender mejor el método se resolverá el sistema de ecuaciones (7.15), con el método de la Newton, empleando como valores iniciales $x=x_1=1.5$, $y=x_2=2$:

Primero se programa el sistema de ecuaciones y para ello se crea una función, donde se reciben los valores de las variables y se calcula los valores de cada ecuación devolviéndolos en un vector:

```
>>fe=function(x){var f1=x[0]*x[0]+2*x[1]*x[1]-22; var
f2=-2*x[0]*x[0]+x[0]*x[1]-3*x[1]+11; return [f1,f2]}
function (x){var f1=x[0]*x[0]+2*x[1]*x[1]-22; var
f2=-2*x[0]*x[0]+x[0]*x[1]-3*x[1]+11; return [f1,f2]}
```

Se asumen los valores iniciales y con el número de elemento se crea la matriz que contendrá la matriz aumentada del sistema de ecuaciones lineales que se forma en el proceso:

```
>>x=[1.5,2]; n=x.length; a=new Array(n); for(i=0;i<n;i++) a[i]=new
Array(n+1);
[]
```

Entonces se calcula la matriz de coeficientes (la matriz de derivadas parciales de la ecuación (731)):

```
>>for(i=0;i<n;i++){h=x[i]*1e-6; x[i]+=h; f2=fe(x); x[i]-=2*h; f1=fe(x);
x[i]+=h; for(j=0;j<n;j++) a[j][i]=(f2[j]-f1[j])/(2*h);}; a
[[3.0000000001232743, 8.000000000230045], [-3.9999999999669926,
-1.4999999997655777]]
```

Y el vector de las constantes (el vector de funciones con signo cambiado de la misma ecuación):

```
>>f1=fe(x); for(i=0;i<n;i++) a[i][n]=-f1[i]; a
[[3.0000000001232743, 8.000000000230045, 11.75], [-3.9999999999669926,
-1.4999999997655777, -3.5]]
```

Ahora, con la matriz aumentada, se resuelve el sistema de ecuaciones con el método de Gauss Jordán (encontrando así los incrementos Δx de la ecuación (731)):

```
>>dx=gaussj(a)
[[0.3772727273860554], [1.327272727186249]]
```

Y con los mismos se calculan los nuevos valores de x:

```
>>for(i=0;i<n;i++) x[i]+=dx[i][0]; x
[1.8772727273860554, 3.327272727186249]
```

Y como ni los valores de Δx son cercanos a cero, ni los nuevos valores de "x" son cercanos a los anteriores, se repite el proceso:

```
>>for(i=0;i<n;i++){h=x[i]*1e-6; x[i]+=h; f2=fe(x); x[i]-=2*h; f1=fe(x);
x[i]+=h; for(j=0;j<n;j++) a[j][i]=(f2[j]-f1[j])/(2*h);}; f1=fe(x);
```

```
for(i=0;i<n;i++) a[i][n]=-f1[i]; dx=gaussj(a)
[[0.1359088293381877], [-0.313764208340408 25]]
>>for(i=0;i<n;i++) x[i]+=dx[i][0]; x
[2.0131815567242435, 3.0135085188458404]
```

Ahora los valores de "x" se van acercando a los de la iteración anterior y los de Δx a cero. Repitiendo el proceso unas cuantas veces más se obtiene:

```
>>for(i=0;i<n;i++){h=x[i]*1e-6; x[i]+=h; f2=fe(x); x[i]-=2*h; f1=fe(x);
x[i]+=h; for(j=0;j<n;j++) a[j][i]=(f2[j]-f1[j])/(2*h);}; f1=fe(x);
for(i=0;i<n;i++) a[i][n]=-f1[i]; dx=gaussj(a)
[[-0.013154942731710357], [-0.013472717054074589]]
>>for(i=0;i<n;i++) x[i]+=dx[i][0]; x
[2.000026613992533, 3.0000358017917654]

>>for(i=0;i<n;i++){h=x[i]*1e-6; x[i]+=h; f2=fe(x); x[i]-=2*h; f1=fe(x);
x[i]+=h; for(j=0;j<n;j++) a[j][i]=(f2[j]-f1[j])/(2*h);}; f1=fe(x);
for(i=0;i<n;i++) a[i][n]=-f1[i]; dx=gaussj(a)
[[-0.00002661395157279945], [-0.000035801532766671496]]
>>for(i=0;i<n;i++) x[i]+=dx[i][0]; x
[2.00000000004096, 3.000000000258999]

>>for(i=0;i<n;i++){h=x[i]*1e-6; x[i]+=h; f2=fe(x); x[i]-=2*h; f1=fe(x);
x[i]+=h; for(j=0;j<n;j++) a[j][i]=(f2[j]-f1[j])/(2*h);}; f1=fe(x);
for(i=0;i<n;i++) a[i][n]=-f1[i]; dx=gaussj(a)
[[-4.095923601833369e-11], [-2.589999325912955e-10]]
>>for(i=0;i<n;i++) x[i]+=dx[i][0]; x
[2.0000000000000004, 2.9999999999999996]

>>for(i=0;i<n;i++){h=x[i]*1e-6; x[i]+=h; f2=fe(x); x[i]-=2*h; f1=fe(x);
x[i]+=h; for(j=0;j<n;j++) a[j][i]=(f2[j]-f1[j])/(2*h);}; f1=fe(x);
for(i=0;i<n;i++) a[i][n]=-f1[i]; dx=gaussj(a)
[[-3.806478941730828e-16], [1.2688263137537556e-16]]
>>for(i=0;i<n;i++) x[i]+=dx[i][0]; x
[2, 2.9999999999999996]

>>for(i=0;i<n;i++){h=x[i]*1e-6; x[i]+=h; f2=fe(x); x[i]-=2*h; f1=fe(x);
x[i]+=h; for(j=0;j<n;j++) a[j][i]=(f2[j]-f1[j])/(2*h);}; f1=fe(x);
for(i=0;i<n;i++) a[i][n]=-f1[i]; dx=gaussj(a)
[[-6.344131569305419e-17], [3.1720657846527087e-16]]
>>for(i=0;i<n;i++) x[i]+=dx[i][0]; x
[2, 3]
```

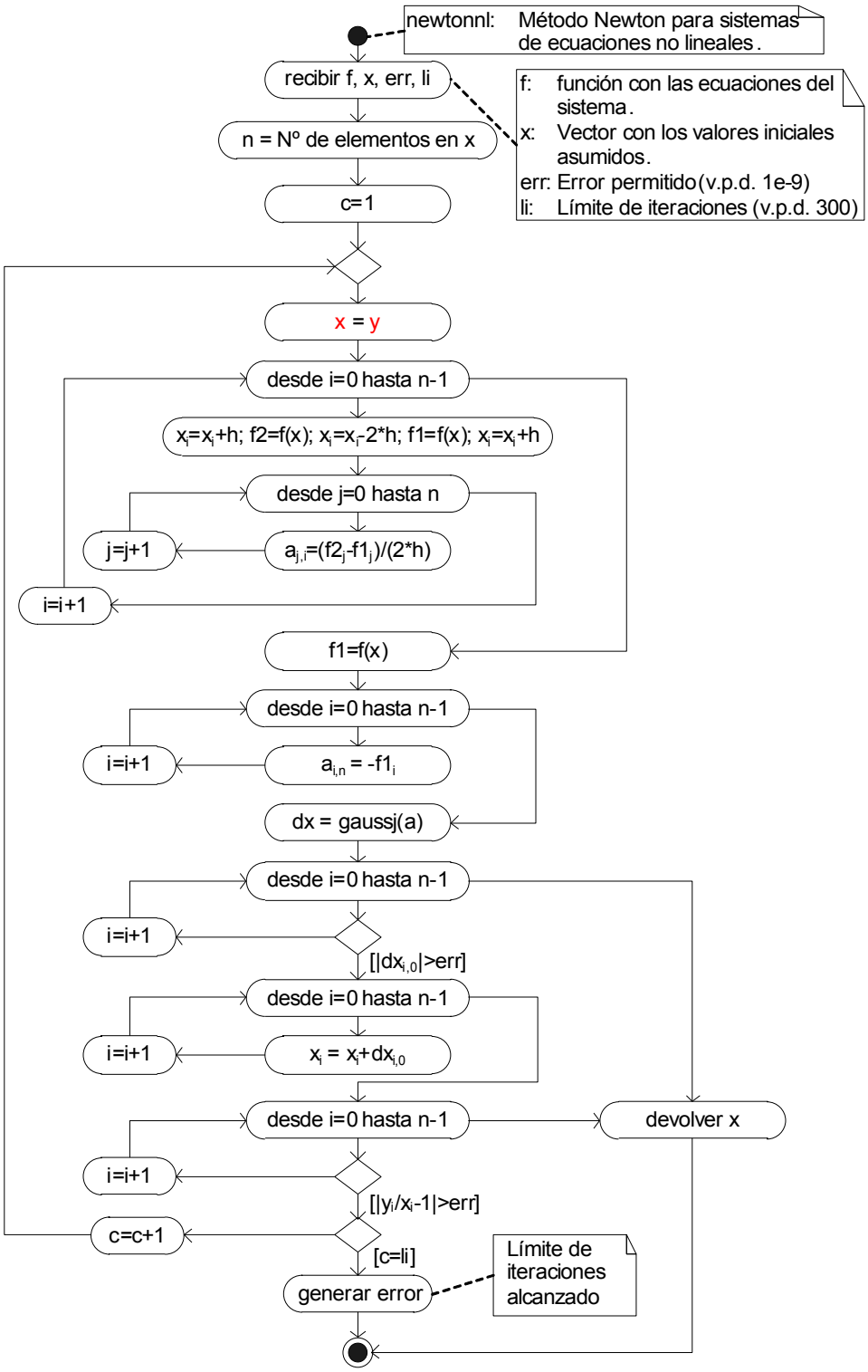
Que son las soluciones exactas y para llegar a las mismas sólo han sido necesarias 7 iteraciones (mucho menos que en los métodos Simplex y del descenso acelerado).

7.3.2. Algoritmo y código

El algoritmo que automatiza el proceso de cálculo se presenta en la siguiente página (las variables en rojo denotan operaciones con vectores).

El código respectivo, añadido a la librería "mat205.js", es:

```
function newtons(f,x,err,li) {
  if (x==null)
    throw new Error("Valores iniciales no asumidos (newtons).");
```




```
if (err==null) err=1e-9;
if (li==null) li=300;
var i,j,v,h,f1,f2,dx,c=1;
var n=x.length, y=new Array(n), a=new Array(n);
for(i=0;i<n;i++) a[i]=new Array(n+1);
while (true) {
```

```
//Guardado de los valores anteriores para efectos de comparación
for (i=0;i<n;i++)
  y[i]=x[i];
//Cálculo de los coeficientes de la matriz aumentada
for (i=0;i<n;i++){
  h= x[i]!=0 ? x[i]*1e-6: 1e-6;
  x[i]+=h; f2=f(x);
  x[i]-=2*h; f1=f(x);
  x[i]+=h;
  for(j=0;j<n;j++)
    a[j][i]=(f2[j]-f1[j])/(2*h);
}
//Cálculo de la columna de constantes de la matriz aumentada
f1=f(x);
for (i=0;i<n;i++)
  a[i][n]=-f1[i];
//Cálculo de los valores de delta x
dx=gauss(a);
//Fin del proceso si los valores de delta x son menores al error
for (i=0;i<n;i++)
  if (abs(dx[i][0])>err) break;
if (i==n) return x;
//Cálculo de los nuevos valores de x
for(i=0;i<n;i++)
  y[i]=x[i];
//Cálculo de los coeficientes de la matriz aumentada
for (i=0;i<n;i++){
  h= x[i]!=0 ? x[i]*1e-6: 1e-6;
  x[i]+=h; f2=f(x);
  x[i]-=2*h; f1=f(x);
  x[i]+=h;
  for(j=0;j<n;j++)
    a[j][i]=(f2[j]-f1[j])/(2*h);
}
//Cálculo de la columna de constantes de la matriz aumentada
f1=f(x);
for (i=0;i<n;i++)
  a[i][n]=-f1[i];
//Cálculo de los valores de delta x
dx=gauss(a);
//Fin del proceso si los valores de delta x son menores al error
for (i=0;i<n;i++)
  if (abs(dx[i][0])>err) break;
if (i==n) return x;
//Cálculo de los nuevos valores de x
for(i=0;i<n;i++)
  x[i]+=dx[i][0];
//Fin del proceso si los valores de x son cercanos a los de y
for (i=0;i<n;i++)
```

```
      if (abs(x[i]/y[i]-1)>err) break
    if (i==n) return x;
    //Fin del proceso si se ha llega al límite de iteraciones
    if (c++ == li)
      throw new Error("Convergencia no lograda (newtons).");
  }
}
```

Haciendo correr el programa con el sistema del ejemplo manual (con el error y límite de iteraciones por defecto) se obtiene:

```
> fe=function(x){var f1=x[0]*x[0]+2*x[1]*x[1]-22; var
  f2=-2*x[0]*x[0]+x[0]*x[1]-3*x[1]+11; return [f1,f2]}; newtonnl(fe,[1.5,2])
  [2.0000000000409597, 3.0000000002589995]
```

Que con los 9 dígitos (que es el error por defecto) es:

```
> precision(newtonnl(fe,[1.5,2]),9)
  [2, 3]
```

Que son las soluciones exactas del sistema.

7.4. EJEMPLOS

1, Encuentre las soluciones del siguiente sistema de ecuaciones lineales, empleando primero el método Simplex con 4 dígitos de precisión, luego con el método del descenso acelerado con 3 dígitos de precisión y finalmente con el método de Newton con 9 dígitos de precisión, empleando como valores iniciales los resultados obtenidos con los métodos Simplex y del descenso acelerado.

$$3x^{2.1}-5y=7.0$$
$$y^{1.2}+4z=14.3$$
$$x+y^2+z^2=14.0$$

La función de error para los métodos Simplex y del descenso acelerado es:

```
>>fe1=function(v){  var x=v[0],y=v[1],z=v[2];  return sqr(3*pow(x,2.1)-
5*y-7)+sqr(pow(y,1.2)+4*z-14.3)+sqr(x+y*y+z*z-14); }
function (v){
 var x=v[0],y=v[1],z=v[2];
 return sqr(3*pow(x,2.1)-5*y-7)+sqr(pow(y,1.2)+4*z-14.3)+sqr(x+y*y+z*z-
14);
}
```

Como se puede ver, en esta función, para facilitar la escritura de las ecuaciones y reducir la posibilidad de error, se asignan los elementos del vector (que es el parámetro de la función) a variables locales "x", "y" y "z". Por supuesto (al igual que se ha hecho con los ejemplos manuales) se puede trabajar directamente con los elementos vector "v".

Empleando el método Simplex, con valores iniciales iguales a 1.1, 1.2 y 1.3, se obtiene:

```
>>r1=precision(simplex(fe1,[1.1,1.2,1.3],1e-4),4)
[1.727, 0.4891, 3.469]
```

De la misma manera se procede con el descenso acelerado:

```
>>r2=precision(descensoa(fe1,[1.1,1.2,1.3],1e-3),3)
err: descensoa no converge, x=[2.008860078455649, 1.1971121589212677,
```

```
3.253451665429264]; li=500; fe=0.0027866546255015407
```

Como se puede ver, sin embargo, el método no logra convergencia en el límite de iteraciones por defecto (500), por lo que se incrementa dicho límite a 1000:

```
>>r2=precision(descensoa(fe1,[1.1,1.2,1.3],1e-3,1000),3)
err: descensoa no converge, x=[1.977673577392309, 1.112679881007041,
3.285768779920708]; li=1000; fe=0.0005565073714420744
```

Con lo que tampoco se logra convergencia, incrementando nuevamente el límite de iteraciones a 3000:

```
>>r2=precision(descensoa(fe1,[1.1,1.2,1.3],1e-3,3000),3)
err: descensoa no converge, x=[1.949090670706788, 1.0367235160627701,
3.31326448056072]; li=3000; fe=0.000099955514211432457
```

Tampoco se logra convergencia, por lo que se incrementa una vez más dicho límite:

```
>>r2=precision(descensoa(fe1,[1.1,1.2,1.3],1e-3,5000),3)
[1.95, 1.03, 3.32]
```

Con lo que se logra convergencia, por lo tanto, el método del descenso acelerado requiere más de 4000 iteraciones para llegar al resultado.

Para el método de Newton se debe crear otra función, pues en dicho método la función debe devolver un vector con los valores de cada ecuación en lugar del valor de la función de error:

```
>>fe2=function (v){  var x=v[0],y=v[1],z=v[2],f1,f2,f3;  f1=3*pow(x,2.1)-
5*y-7; f2=pow(y,1.2)+4*z-14.3; f3=x+y*y+z*z-14; return [f1,f2,f3]; }
function (v){
 var x=v[0],y=v[1],z=v[2],f1,f2,f3;
 f1=3*pow(x,2.1)-5*y-7; f2=pow(y,1.2)+4*z-14.3; f3=x+y*y+z*z-14;
return [f1,f2,f3];
}
```

Ahora se encuentran las soluciones empleando como valores iniciales los resultados obtenidos con Simplex y descenso acelerado:

```
>>precision(newtonnl(fe2,r1),9)
[1.72652615, 0.488925748, 3.46906694]
>>precision(newtonnl(fe2,r2),9)
[1.94379709, 1.02280352, 3.31814344]
```

Como se puede ver se trata de dos conjuntos de soluciones, por lo tanto, en este caso, Simplex converge a un conjunto de soluciones y el descenso acelerado a otro.

2. Encuentre las soluciones de los siguientes sistemas de ecuaciones no lineales empleando el método Simplex (con 5 dígitos de precisión), el método del descenso acelerado (con 5 dígitos de precisión) y el método de Newton (con 12 dígitos de precisión), empleando como valores iniciales los valores calculados con Simplex y el descenso acelerado.

$$3x_1-\cos(x_2x_3)-\frac{1}{2}=0$$

$$x_1^2-81(x_2+0.1)^2+\sin(x_3)+1.06=0$$

$$e^{-x_1x_2}+20x_3+\frac{1}{3}(10\pi-3)=0$$

Al igual que en el ejemplo anterior, primero se crea la función de error para los métodos Simplex y Descenso Acelerado:

```
>>fe1=function(v){  var x1=v[0],x2=v[1],x3=v[2];  return sqr(3*x1-
cos(x2*x3)-1/2)+sqr(x1*x1-81*sqr(x2+0.1)+sin(x3)+1.06)+    sqr(exp(-
x1*x2)+20*x3+(10*PI-3)/3); }
function (v){
 var x1=v[0],x2=v[1],x3=v[2];
 return sqr(3*x1-cos(x2*x3)-1/2)+sqr(x1*x1-81*sqr(x2+0.1)+sin(x3)+1.06)+
   sqr(exp(-x1*x2)+20*x3+(10*PI-3)/3);
}
```

Asumiendo valores iniciales iguales a 1, 2 y 3, con el método Simplex se obtiene:

```
>>r1=precision(simplex(fe1,[1,2,3],1e-5),5)
[0.49815, -0.19961, -0.52883]
```

Y con el método del descenso acelerado:

```
>>r2=precision(descensoa(fe1,[1,2,3],1e-5),5)
[0.49815, -0.19961, -0.52883]
```

En este caso con ambos métodos convergen a la misma solución y devuelven los mismos resultados, por lo que se puede emplear cualquiera de ellos como valores iniciales del método de Newton.

La función para el método de Newton, es:

```
>>fe2=function(v){  var x1=v[0],x2=v[1],x3=v[2],f1,f2,f3;  f1=3*x1-
cos(x2*x3)-1/2; f2=x1*x1-81*sqr(x2+0.1)+sin(x3)+1.06;  f3=exp(-
x1*x2)+20*x3+(10*PI-3)/3;  return [f1,f2,f3]; }
function (v){
 var x1=v[0],x2=v[1],x3=v[2],f1,f2,f3;
 f1=3*x1-cos(x2*x3)-1/2; f2=x1*x1-81*sqr(x2+0.1)+sin(x3)+1.06;
 f3=exp(-x1*x2)+20*x3+(10*PI-3)/3;
 return [f1,f2,f3];
}
```

Y las soluciones, con 12 dígitos de precisión, son:

```
>>precision(newtonnl(fe2,r2,1e-12),12)
[0.498144684589, -0.199605895544, -0.528825977573]
```

7.5. EJERCICIOS

Como en los anteriores temas para presentar los ejercicios que resuelva en la calculadora, copie las instrucciones en un editor de texto y luego, al momento de presentar el trabajo, vuelva a copiarlos y ejecutarlos en la calculadora. Alternativamente, puede guardar el ejercicio resuelto como una página HTML, haciendo clic derecho en página y eligiendo la opción "guardar como" (o su equivalente) y guardando la página como página web completa, o fichero único, asignándole un nombre adecuado como ejercicio1.html.

1. Calcule las soluciones de los siguientes sistemas de ecuaciones no lineales con el método Simplex, empleando: a) x=-7.5, y=1.5 y el error por defecto para el primera sistema; b) x=1.1, y=1.2, z=1.3 y un error igual a 1e-5 para el segundo sistema; c) x1=0.6, x2=0.4, x3=0.3, un error igual a 1e-7 y un límite de 600 iteraciones para el tercero.

$$\begin{aligned} 2x^2 + 5xy - 4x &= 115 \\ e^{\frac{x+y}{5}} + x^2y^2 - 70y &= 15 \end{aligned}$$

$$\begin{aligned} x^2+ y^2+ z^2 &= 17 \\ x\,y\,z &= 2 \\ x+ y- z^2 &= -4 \end{aligned}$$

$$\begin{aligned} 3\,x_1-\cos(x_2 x_3) &= 0.5 \\ x_1^2-81(x_2+ 0.1)^2+ \sin(x_3) &= -1.06 \\ \mathrm{e}^{-x_1 x_2}+ 20\,x_3 &= -9.472 \end{aligned}$$

2. Encuentre las soluciones de los siguiente sistema de ecuaciones no lineales con el método del descenso acelerado, empleando: a) x=1, y=2, z=3 y un error igual a 1e-3 para el primer sistema; b) x_1=1.1, x_2=1.2 y un error igual a 1e-8 para el segundo.

$$\begin{aligned} x\,y\,z-x^2+ y^2 &= 134 \\ x\,y- z^2 &= 0.09 \\ \mathrm{e}^x-\mathrm{e}^y+ z &= 0.41 \\ \sin(4\pi x_1 x_2)-2\,x_2-x_1 &= 0 \\ \left(\frac{4\pi-1}{4\pi}\right)(\mathrm{e}^{2x_1}-\mathrm{e})+ 4\,\mathrm{e}\,x_2^2-2\,\mathrm{e}\,x_1 &= 0 \end{aligned}$$

3. Encuentre las soluciones de los siguientes sistemas de ecuaciones no lineales con el método de Newton, empleando: a) x_1=1.5, x_2=2 y un error igual a 1e-15 para el primer sistema b) x_1=1.1, x_2=1.1, x_3=1.1, x_4=1.1, x_5=1.1, x_6=1.1, x_7=1.1, x_8=1.1, x_9=1.1, x_{10}=1.1, x_{11}=11.1 y un error igual a 1e-9 para el segundo.

$$\begin{aligned} \ln(x_1^2+ x_2^2)-\sin(x_1 x_2) &= \ln(2)+ \ln(\pi) \\ \mathrm{e}^{x_1-x_2}+ \cos(x_1 x_2) &= 0 \end{aligned}$$

$$\begin{aligned} x_1+ x_4 &= 3 \\ 2\,x_1+ x_2+ x_4+ x_7+ x_8+ x_9+ 2\,x_{10} &= 10+ r \\ x_2+ 2\,x_5+ x_6+ x_7 &= 8 \\ 2\,x_3+ x_5 &= 4r \\ x_1 x_5 &= a_1 x_2 x_4 \\ x_6\sqrt{x_2} &= a_2\sqrt{x_2 x_4 x_{11}} \\ x_7\sqrt{x_4} &= a_3\sqrt{x_1 x_4 x_{11}} \\ x_8 x_4 &= a_4 x_2 x_{11} \\ x_9 x_4 &= a_5 x_1\sqrt{x_3 x_{11}} \\ x_{10} x_4^2 &= a_6 x_4^2 x_{11} \\ x_{11} &= x_1+ x_2+ x_3+ x_4+ x_5+ x_6+ x_7+ x_8+ x_9+ x_{10} \end{aligned}$$

$$a_1=0.193;\ a_2=0.002597;\ a_3=0.003448;\ a_4=0.00001799;$$
$$a_5=0.0002155;\ a_6=0.00003846;\ r=4.056734$$

8. AJUSTE E INTERPOLACIÓN DE DATOS

Con frecuencia en la resolución de problemas en el campo de la ingeniería, los únicos datos disponibles se encuentran en forma tabular. Estos datos, que por lo general son el resultado de mediciones de campo o de laboratorio, deben ser empleados para realizar una serie de cálculos que van desde la estimación de valores intermedios (interpolación), pasando por el cálculo de raíces, hasta la integración y diferenciación numérica.

En estos casos existen dos alternativas: a) Ajustar los datos tabulados a una ecuación analítica y b) Calcular valores por interpolación o extrapolación de los datos tabulados. En este tema se estudia las dos alternativas.

Se prefiere el ajuste cuando los datos no son confiables (por ejemplo los resultantes de mediciones de campo) o cuando se tiene idea de la forma que tendrá la curva. Se prefiere la interpolación cuando los datos son confiables (por ejemplo mediciones precisas de laboratorio o datos tabulados resultantes de cálculos matemáticos).

8.1. MÉTODO DE LOS MÍNIMOS CUADRADOS

Este es el método más empleado para ajustar datos tabulados a funciones analíticas. En este método se busca minimizar el cuadrado de las diferencias que existen entre los valores calculados con la ecuación ajustada y los valores tabulados (valores conocidos).

Como en realidad se trata de un problema de optimización (se busca el mínimo) puede ser resuelto empleando diferentes métodos, en este capítulo se encontrará el mínimo recurriendo al cálculo diferencial, es decir igualando la derivada de la función a cero.

Aunque es posibledesarrollar un método general para ajustar datos a cualquier expresión, el resultado es un método iterativo. Por ello en este tema se presenta el método para ajustar datos específicamente a una ecuación de la forma:

$$y = c_1 f_1(x) + c_2 f_2(x) + \cdots + c_m f_m(x) \qquad (8.1)$$

Donde "y" es la variable dependiente, "x" la variable independiente, "f_i" son funciones que dependen de "x" y "c_1" a "c_m" son los coeficientes resultantes del ajuste, siendo "m" el número de coeficientes en la expresión analítica.

Aunque el desarrollo del método está limitado a la forma (8.1), una gran variedad de ecuaciones tienen esa forma o pueden ser colocadas en esa forma (incluyendo todas las expresiones polinomiales), por lo que el método es de gran utilidad práctica.

Para el desarrollo del método se asumirá que se cuenta con los siguientes datos experimentales:

x	4.0	3.2	6.8	9.3	1.2	0.7	5.5	7.4	6.5	2.4
y	6.2	5.3	9.1	12.0	3.3	1.5	7.9	10.6	8.7	6.3

Los que se quieren ajustar a la ecuación:

$$y = a + b\,x^2 + c\,x^4 + d\,\ln(x) \qquad (8.2)$$

Es decir se quiere encontrar los valores de los coeficientes "a", "b", "c" y "d", de manera que se pueda emplear esta ecuación en lugar de los datos tabulados. Esta ecuación se encuentra en forma (8.1), siendo las fun-

ciones en este caso: $f_1(x)=1$; $f_2(x)=x^2$; $f_3(x)=x^4$ y $f_4(x)=\ln(x)$, mientras que los coeficientes son: $c_1=a$; $c_2=b$; $c_3=c$ y $c_4=d$.

Para encontrar el valor de las constantes se reemplaza cada par de datos (x_i, y_i) en la ecuación a ajustar (ecuación 8.2):

$$\begin{aligned}
a+b\cdot 4.0^{0.2}+c\cdot 4.0^{0.2}+d\cdot\ln(4.0)-6.2&=r_1\\
a+b\cdot 3.2^{0.2}+c\cdot 3.2^{0.2}+d\cdot\ln(3.2)-5.3&=r_2\\
a+b\cdot 6.8^{0.2}+c\cdot 6.8^{0.2}+d\cdot\ln(6.8)-9.1&=r_3\\
&\cdots\\
a+b\cdot 6.5^{0.2}+c\cdot 6.5^{0.2}+d\cdot\ln(6.5)-8.7&=r_9\\
a+b\cdot 2.4^{0.2}+c\cdot 2.4^{0.2}+d\cdot\ln(1.4)-6.3&=r_{10}
\end{aligned}$$

Donde "r_1", "r_2" … "r_n" son los restos (las diferencias) entre los valores calculados con la ecuación ajustada ($f(x_i)$) y los valores conocidos (y_i). Si el ajuste es perfecto, los restos son cero, por lo tanto, para conseguir el mejor ajuste posible se deben minimizar estos restos.

Generalizando para una ecuación de la forma (8.1) los restos son:

$$\begin{aligned}
c_1 f_1(x_1)+c2\,f_2(x_1)+\cdots+c_m f_n(x_1)-y_1 &= r_1\\
c_1 f_1(x_2)+c2\,f_2(x_2)+\cdots+c_m f_n(x_2)-y_2 &= r_2\\
\cdots &= \cdots\\
c_1 f_1(x_{n-1})+c2\,f_2(x_{n-1})+\cdots+c_m f_n(x_{n-1})-y_{n-1} &= r_{n-1}\\
c_1 f_1(x_n)+c2\,f_2(x_n)+\cdots+c_m f_n(x_n)-y_n &= r_n
\end{aligned} \qquad (8.3)$$

El mejor ajuste se consigue minimizando estos restos, es decir:

$$\sum_{i=1}^{n} r_i = mínimo \qquad (8.4)$$

Pero, como ya se vio en la resolución de ecuaciones no lineales, al minimizar se debe evitar el signo (para evitar falsos mínimos) lo que se consigue simplemente elevando al cuadrado los restos:

$$\sum_{i=1}^{n} r_i^2 = mínimo \qquad (8.5)$$

Esta es la ecuación de los mínimos cuadrados y de la cual, como se puede ver, deriva su nombre.

Como ya se dijo, el mínimo puede ser encontrado empleando diferentes métodos y en este caso (para ecuaciones de la forma 8.1), se empleará el cálculo diferencial.

Para ello primero se escribe el sistema de ecuaciones (8.3) en forma matricial:

$$\begin{bmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_m(x_1)\\ f_1(x_2) & f_2(x_2) & \cdots & f_m(x_2)\\ \cdots & \cdots & \cdots & \cdots\\ f_1(x_n) & f_2(x_n) & \cdots & f_m(x_n) \end{bmatrix} * \begin{bmatrix} c_1\\ c_2\\ \cdots\\ c_m \end{bmatrix} - \begin{bmatrix} y_1\\ y_2\\ \cdots\\ y_n \end{bmatrix} = \begin{bmatrix} r_1\\ r_2\\ \cdots\\ r_n \end{bmatrix} \qquad (8.6)$$

Que en forma abreviada es:

$$F*C-Y=R \qquad (8.7)$$

Ahora se encuentra el mínimo, derivando la ecuación (8.1) con respecto a cada uno de los coeficientes e igualando el resultado a cero:

$$\frac{\partial}{\partial c_k}\sum_{i=1}^{n} r_i^2=\sum_{i=1}^{n}\frac{\partial r_i^2}{\partial c_k}=\sum_{i=1}^{n} 2\,r_i\frac{\partial r_i}{\partial c_k}=2\sum_{i=1}^{n} r_1\frac{\partial r_1}{\partial c_k}=0 \qquad (8.8)$$

De donde resulta:

$$\sum_{i=1}^{n} r_i\frac{\partial r_i}{\partial c_k}=0 \quad \{k=1\rightarrow n \qquad (8.9)$$

Que en forma desarrollada es:

$$\begin{aligned} r_1\frac{\partial r_1}{\partial c_1}+ r_2\frac{\partial r_2}{\partial c_1}+\cdots+ r_n\frac{\partial r_n}{\partial c_1}&=0\\ r_1\frac{\partial r_1}{\partial c_2}+ r_2\frac{\partial r_2}{\partial c_2}+\cdots+ r_n\frac{\partial r_n}{\partial c_2}&=0\\ &\cdots\\ r_1\frac{\partial r_1}{\partial c_m}+ r_2\frac{\partial r_2}{\partial c_m}+\cdots+ r_n\frac{\partial r_n}{\partial c_m}&=0 \end{aligned} \qquad (8.10)$$

Hasta esta parte, el desarrollo es general (no depende de la forma que tenga la ecuación a ajustar) y en consecuencia esta expresión puede ser empleada para ajustar datos a una expresión de cualquier forma, pero como se dijo, el proceso es iterativo.

Aplicándola específicamente a la forma (8.1), se puede comprobar fácilmente que las derivadas parciales con relación a los coeficientes son iguales a sus funciones respectivas. Así por ejemplo la derivada parcial de r_2 con respecto a c_1 ($\partial r_2/\partial c_1$) es igual a la función $f_1(x_2)$; la derivada parcial de r_1 con respecto a c_2, ($\partial r_1/\partial c_2$) es igual a la función $f_2(x_1)$ y en general para cualquier derivada se cumple que:

$$\frac{\partial r_i}{\partial c_k}=f_k(x_i) \quad \begin{cases} i=1\rightarrow n \\ k=1\rightarrow m \end{cases} \qquad (8.11)$$

Reemplazando los valores de esta expresión en la ecuación (8.10) se obtiene:

$$\begin{aligned} r_1 f_1(x_1)+ r_2 f_1(x_2)+\cdots+ r_n f_1(x_n)&=0\\ r_1 f_2(x_1)+ r_2 f_2(x_2)+\cdots+ r_n f_2(x_n)&=0\\ &\cdots\\ r_1 f_m(x_1)+ r_2 f_m(x_2)+\cdots+ f_m(x_n)&=0 \end{aligned} \qquad (8.12)$$

Que escrita en forma matricial es:

$$\begin{bmatrix} f_1(x_1) & f_1(x_2) & \cdots & f_1(x_n)\\ f_2(x_1) & f_2(x_2) & \cdots & f_2(x_2)\\ \cdots & \cdots & \cdots & \cdots\\ f_m(x_1) & f_m(x_2) & \cdots & f_m(x_n) \end{bmatrix} * \begin{bmatrix} r_1\\ r_2\\ \cdots\\ r_m \end{bmatrix} = \begin{bmatrix} 0\\ 0\\ \cdots\\ 0 \end{bmatrix} \qquad (8.13)$$

Como se puede ver, la matriz "F" de esta ecuación es la transpuesta de la matriz "F" de la ecuación (8.6). Entonces, se cumple que:

$$F^T * R=0 \qquad (8.14)$$

Reemplazando "R" por la expresión de la ecuación (8.7) y realizando algunas operaciones se obtiene:

$$F^T*(F*C-Y)=0$$

$$F^T*F*C=F^T*Y \qquad (8.15)$$

Esta expresión constituye un sistema de "m" ecuaciones lineales con "m" incógnitas (los valores de "C"), siendo la matriz de coeficientes F^T*F y la de constantes F^T*Y. En consecuencia, el cálculo de los coeficientes del ajuste se reduce a la resolución de un sistema de ecuaciones lineales.

Para verificar qué tan bueno (o malo) resulta el ajuste se pueden graficar los puntos conocidos y la ecuación ajustada y juzgar visualmente si la curva resultante representa bien (o no) los datos tabulados.

Una forma más objetiva de verificar el ajuste es mediante el cálculo del coeficiente de correlación, que para un ajuste en una variable, se define como:

$$cor=\sqrt{\frac{\sum_{i=1}^{n}(f(y_i)-\overline{y})^2}{\sum_{i=1}^{n}(y_i-\overline{y})^2}} \qquad (8.16)$$

Donde "f" es la ecuación ajustada, y_i son los datos conocidos de la variable dependiente y $\overline{y}$ es el promedio de dichos datos.

Cuanto más cercano es este valor a 1, mejor es el ajuste, siendo 1 el ajuste perfecto.

No obstante, se debe tener cuidado al juzgar los resultados en base al coeficiente de correlación, pues no siempre su valor es señal de un buen ajuste. Por ejemplo, cuando se ajustan "n" datos a una ecuación polinomial de grado "n", el coeficiente de correlación devuelve siempre 1 (ajuste perfecto), sin embargo, casi siempre ese es un mal ajuste (lo que puede verificarse graficando de la función ajustada).

8.1.1. Ejemplo manual

Para comprender mejor el proceso se ajustarán los siguientes datos:

x	1.0	2.5	3.5	4.0
Y	3.8	15.0	26.0	33.0

A la ecuación:

$$y=a+bx^2$$

Que tiene la forma de la ecuación (8.1), donde $f_1(x)=1$, $f_2(x)=x^2$, $c_1=a$ y $c_2=b$. En esta ecuación existen dos coeficientes (m=2) y se cuenta con cuatro puntos (n=4), en consecuencia los datos son:

```
>>x=[1,2.5,3.5,4]; y=[3.8,15,26,33]; n=x.length; m=2;
2
```

Las funciones son:

```
>>f=new Array(m); f[0]=function(x){return 1;}; f[1]=function(x){return
x*x;};
```

```
function (x){return x*x;}
```

Con estos datos se calcula la matriz "F":

```
>>F=new Array(n); for(i=0;i<n;i++){F[i]=new Array(m); for(j=0;j<m;j++)
F[i][j]=f[j](x[i]);} F
[[1, 1], [1, 6.25], [1, 12.25], [1, 16]]
```

Y la matriz traspuesta "FT":

```
>>FT=transpose(F)
[[1, 1, 1, 1], [1, 6.25, 12.25, 16]]
```

Entonces se calcula la matriz de coeficientes del sistema de ecuaciones lineales:

```
>>A=mul(FT,F)
[[4, 35.5], [35.5, 446.125]]
```

Y la columna de constantes:

```
>>B=transpose(mul(FT,y))
[[77.8], [944.05]]
```

Finalmente, se resuelve el sistema de ecuaciones, en este caso con Crout, obteniéndose así los coeficientes de la ecuación ajustada:

```
>>crout(A,B)
[[2.2789699570815394], [1.9347639484978545]]
```

Por lo tanto la ecuación analítica que representa los datos tabulados es:

$$y = 2.2789699570815394 + 1.9347639484978545\,x^2$$

Para determinar cuán bueno (o malo) es el ajuste primero se programa la función ajustada:

```
>>f=function(x){return 2.2789699570815394+1.9347639484978545*x*x;}
function (x){return 2.2789699570815394+1.9347639484978545*x*x;}
```

Entonces se pueden calcular los valores de "y" para los valores conocidos de "x" (empleando la función "map") mostrando los resultados redondeados al primer dígito después del punto (con "round"):

```
>>round(map(f,x),1)
[4.2, 14.4, 26, 33.2]
```

Comparando los valores calculados con los conocidos (los valores de "y"):

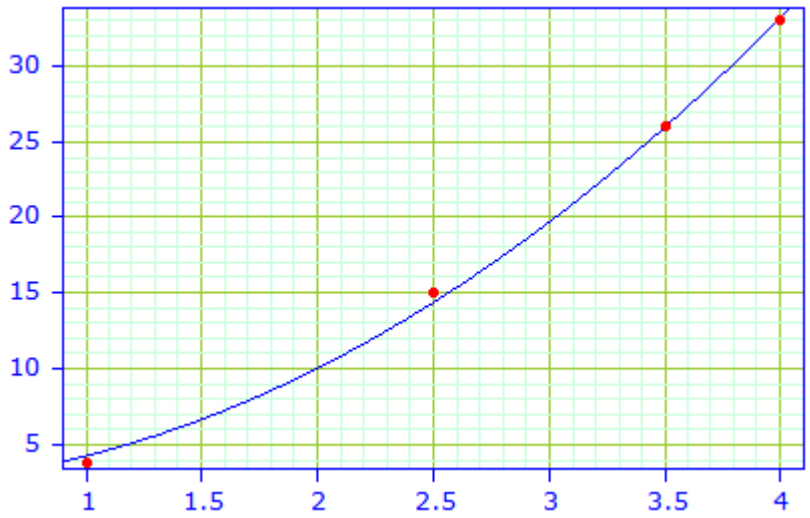
```
>>y
[3.8, 15, 26, 33]
```

Se puede ver que el ajuste no es perfecto, pero sí bueno, pues los valores calculados son próximos a los conocidos. Si se calcula el coeficiente de correlación (ecuación 8.16), se obtiene:

```
>>yp=0; for(i=0;i<n;i++) yp+=y[i]; yp/=n; s1=0; for(i=0;i<n;i++)
s1+=sqr(f(x[i])-yp); s2=0; for(i=0;i<n;i++) s2+=sqr(y[i]-yp); sqrt(s1/s2)
0.9993664587597932
```

Que nos indica que el ajuste ha sido muy bueno (muy cercano a uno). Finalmente, para juzgar visualmente el ajuste, se manda a "plot" la función ajustada y una matriz donde la primera fila son los valores de "x" y la segunda los de "y". Es conveniente fijar los límites "x" de la gráfica ligeramente por debajo y por encima de los límites tabulados:

```
> plot([f,[x,y]],0.9,4.1)
  undefined
```



Como se puede observar en este caso los datos (los puntos) se distribuyen bastante bien a ambos lados de la función ajustada, lo que constituye un buen ajuste y que es coherente con el coeficiente de correlación obtenido.

La ecuación ajustada puede ser empleada igual que cualquier función analítica. Así, aparte de calcular los valores de "y" se puede calcular por ejemplo la derivada, la integral, encontrar el mínimo, encontrar una o más soluciones, etc.

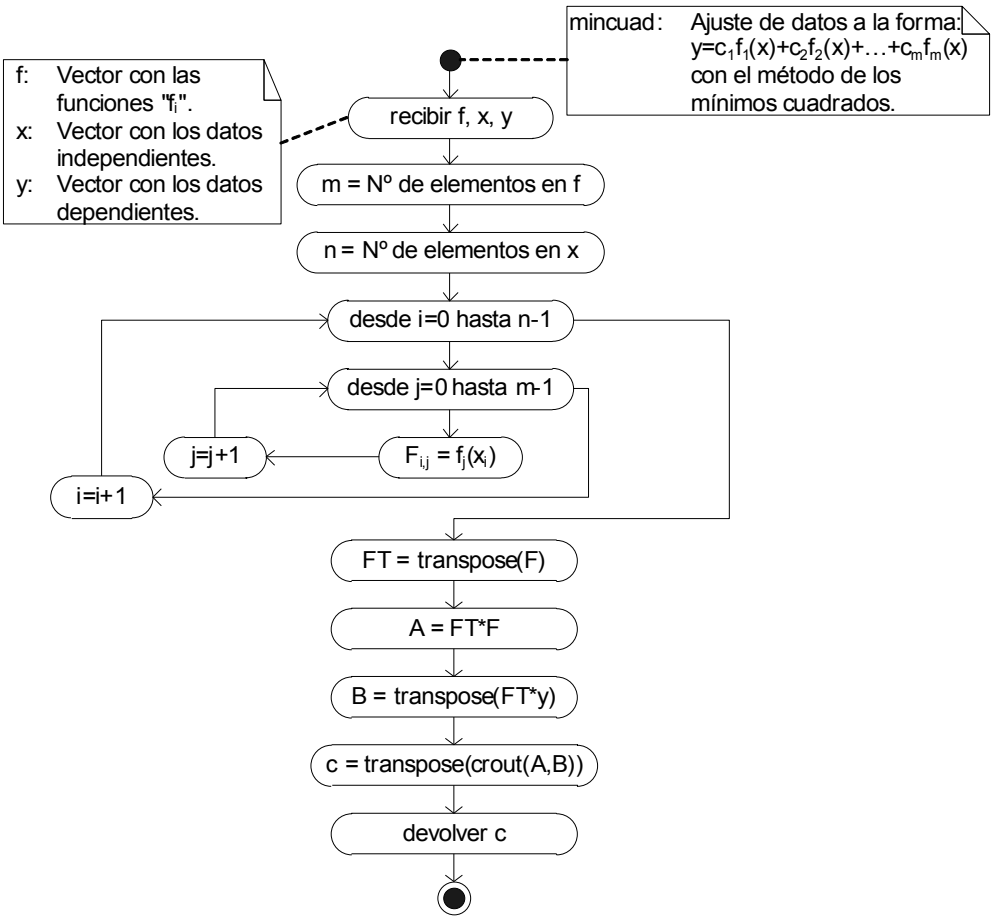
8.1.2. Algoritmo y código

El algoritmo que automatiza el proceso de cálculo se presenta en la siguiente página y el código respectivo, añadido a "mat205.js", es:

```
function mincuad(f,x,y) {
  var i,j,m,n,F,FT,A,B,c;
  m=f.length;
  n=x.length;
  F=new Array(n);
  for (i=0;i<n;i++) {
    F[i]=new Array(m);
    for (j=0;j<m;j++)
      F[i][j]=f[j](x[i]);
  }
  FT=transpose(F);
  A=mul(FT,F);
  B=transpose(mul(FT,y));
  c=transpose(crout(A,B));
  return c;
}
```

Haciendo correr el programa con los datos del ejemplo manual se obtienen los mismos resultados que en dicho ejemplo:

```
>>f=new Array(2); f[0]=function(x){return 1;}; f[1]=function(x){return
x*x;}; x=[1,2.5,3.5,4]; y=[3.8,15,26,33]
```




```
[3.8, 15, 26, 33]
>>c=mincuad(f,x,y)
[2.2789699570815394, 1.9347639484978545]
```

Para facilitar el uso de la ecuación ajustada, es conveniente contar con una generador que reciba los datos del ajuste y devuelva una función con la ecuación ajustada, de manera que la misma pueda ser empleada directamente para calcular valores de "y" a partir de valores conocidos de "x". Dicha función ha sido añadida a la librería "mat205.js" con el nombre "genmincuad". Empleando la misma, se puede generar, por ejemplo, la función ajustada para los datos del ejemplo manual:

```
>>f=new Array(2); f[0]=function(x){return 1;}; f[1]=function(x){return
x*x;}; x=[1,2.5,3.5,4]; y=[3.8,15,26,33];
>>fx=genmincuad(f,x,y)
function (x){
   y=0;
   for (i=0;i<m;i++) y+=c[i]*vf[i](x);
   return y;
 }
```

La función resultante puede ser empleada de la misma forma que una función creada para una ecuación analítica, por ejemplo, se pueden calcular valores de "y" para valores de "x" iguales a 2, 3 y 3.7:

```
>>map(fx,[2,3,3.7])
[10.018025751072958, 19.69184549356223, 28.76588841201717]
```

Con el fin de juzgar numéricamente el ajuste, es conveniente contar también con la función para el cálculo del coeficiente de correlación, dicha

función recibe la función ajustada y los vectores con los datos conocidos:

```
function corr(f,x,y) {
  var i,yp=0,s1=0,s2=0,n=x.length;
  for (i=0;i<n;i++)
    yp+=y[i];
  yp/=n;
  for (i=0;i<n;i++) {
    s1+=sqr(f(x[i])-yp);
    s2+=sqr(y[i]-yp);
  }
  return sqrt(s1/s2);
}
```

Para la función ajustada con los datos del ejemplo manual, se obtiene:

```
>>corr(fx,x,y)
0.9993664587597932
```

Que es el mismo resultado obtenido en el ejemplo manual.

8.2. AJUSTE DE DATOS CON LOS MÉTODOS SIMPLEX Y DESCENSO ACELERADO

Puesto que el ajuste de datos es un proceso de optimización, se pueden emplear los métodos estudiados en el anterior tema.

En este caso la función de error es simplemente la ecuación de definición de los mínimos cuadrados, es decir la ecuación (8.5). Por ejemplo, para ajustar los dato del ejemplo manual, la función de error es:

```
>>fe=function(c){var x=[1,2.5,3.5,4], y=[3.8,15,26,33], s=0, i; for
(i=0;i<x.length;i++) s+=sqr(c[0]+c[1]*x[i]*x[i]-y[i]); return s;};
```

Como se puede ver, la función de error recibe el vector de coeficientes "c" y con los datos tabulados, almacenados en las variables locales "x" y "y", calcula la sumatoria del cuadrado de los residuos de la ecuación a ajustar (ecuación 8.5) devolviendo esa sumatoria como resultado de la función. Los datos tabulados "x" y "y" pueden ser almacenados también en variables globales (teniendo cuidado de no interferir con otras variables).

Entonces, ajustando esta función (que constituye la función de error) con el método Simplex, se obtiene:

```
>>simplex(fe,[1.1,1.2],1e-6)
[2.2789701724530165, 1.9347639481018546]
```

Con el método del descenso acelerado se obtiene:

```
>>descensoa(fe,[1.1,1.2],1e-6)
[2.278969926059215, 1.9347639487803694]
```

Como estos métodos son iterativos y por lo tanto no siempre convergen, se recomienda emplear "mincuad", siempre que sea posible.

8.3. INTERPOLACIÓN LINEAL

A pesar de ser un método de poca precisión, es, por su simplicidad, uno de los métodos más empleados en la práctica, sobre todo en el área de la ingeniería donde en la mayoría de los casos sólo se requieren valores aproximados. Este método proporciona resultados satisfactorios cuando los datos

están distribuidos en segmentos cercanos, o, cuando a pesar de estar espaciados, tienen un comportamiento lineal.

En el método de interpolación lineal se busca el segmento donde se encuentra el valor de la variable independiente *"x"*, con los dos puntos de dicho segmento se calculan los coeficientes de la ecuación lineal y con la misma se calcula el valor de la variable dependiente "y". Si el valor de "x" es mayor al último valor tabulado, o menor al menor valor tabulado, se puede emplear el último par de puntos (o el primer par de puntos) para obtener el valor de "y" por extrapolación.

Si (x_i,y_i) y (x_j,y_j) son los puntos del segmento donde se encuentra el valor conocido: "x", entonces los coeficientes de la línea recta se calculan con:

$$b=\frac{y_j-y_i}{x_j-x_i} \tag{8.17}$$

$$a=y_i-b\,x_i \tag{8.18}$$

El valor interpolado: "y", se calcula con la ecuación de la línea recta:

$$y=a+b\,x \tag{8.19}$$

Se debe hacer notar que debido la necesidad de ubicar el segmento donde se encuentra el valor conocido, los datos deben estar ordenados ascendentemente en base a los valores de la variable independiente.

Para ilustrar el método se interpolará el valor de "y" para un valor de "x" igual a 0.7 aplicando el método de interpolación lineal a los siguientes datos:

x	0	0.2	0.4	0.6	0.8	1.0
y	0	0.199	0.389	0.565	0.717	0.841

Para ello, primero se guardan los datos en las variables "vx" y "vy", y el valor a interpolar en la variable "x":

```
>>vx=[0,0.2,0.4,0.6,0.8,1.0]; vy=[0,0.199,0.389,0.565,0.717,0.841]; x=0.7
0.7
```

Ahora se ubica el segmento en el que se encuentra el valor a interpolar (el valor de "x") y para ello simplemente se recorren los elementos del vector "vx" mientras "x" (el valor a interpolar) sea mayor a dichos elementos:

```
>>for (j=1;j<vx.length;j++) {if (x<vx[j]) break;}; j
4
```

Este número (4) nos informa que el valor de "x" se encuentra entre el cuarto (x_3=0.6) y quinto (x_4=0.8) elementos (recuerde que en Javascript el primer índice es siempre 0).

Observe que la búsqueda comienza a partir del segundo elemento (j=1), esto para asegurar que el menor segmento posible sea siempre el primero (entre 0 y 1). De lo contrario, si la búsqueda comenzara en 1 y el valor a interpolar sería menor al primer valor de la lista, el resultado sería 0 y el en consecuencia el segmento estaría entre -1 y 0, un resultado erróneo pues el menor índice posible en un vector es 0, no -1.

Una vez ubicado el segmento de interpolación (los puntos entre los cuales se encuentra el valor buscado) se aplican las ecuaciones (8.17) y (8.18) para calcular los coeficientes:

```
>>i=j-1; b=(vy[j]-vy[i])/(vx[j]-vx[i]); a=vy[i]-b*vx[i]; write(a,b)
```

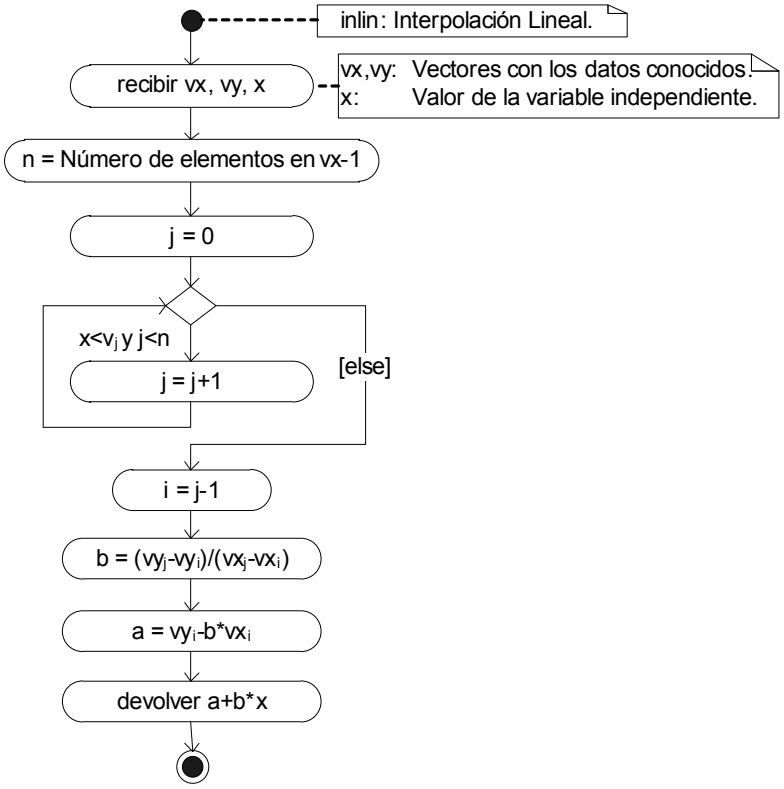
```
0.10900000000000004  0.7599999999999999
```

Finalmente el valor buscado (el valor interpolado) se calcula con la ecuación (8.19):

```
>>y=a+b*x
0.641
```

8.3.1. Algoritmo y código

El algoritmo para la interpolación lineal es:



El código respectivo, añadido a la librería "mat205.js" es:

```
function inlin(vx,vy, x){
  var a,b,n=vx.length-1,i,j=1;
  while (x>vx[j] && j<n) j++;
  i=j-1;
  b=(vy[j]-vy[i])/(vx[j]-vx[i]);
  a=vy[i]-b*vx[i];
  return a+b*x;
}
```

Empleando esta función con los datos del ejemplo manual se obtiene el mismo resultado que en dicho ejemplo:

```
>>vx=[0,0.2,0.4,0.6,0.8,1.0]; vy=[0,0.199,0.389,0.565,0.717,0.841]
[0, 0.199, 0.389, 0.565, 0.717, 0.841]
>>inlin(vx,vy,0.7)
0.641
```

Para interpolar varios valores para el mismo conjunto de datos, resulta más práctico crear una función de interpolación con dichos datos.

Se ha añadido un generador de dichas funciones a la librería "mat205.js" con el nombre "geninlin" y con la misma se puede crear por ejemplo la función de interpolación para los datos del ejemplo manual:

```
>>fx=geninlin(vx,vy)
function (x){
  j=1;
  while (x>vx[j] && j<n) j++;
  i=j-1;
  b=(vy[j]-vy[i])/(vx[j]-vx[i]);
  a=vy[i]-b*vx[i];
  return a+b*x;
}
```

Llamando a esta función con el valor del ejemplo manual (x=0.7), se obtiene el mismo resultado que en dicho ejemplo:

```
>>fx(0.7)
0.641
```

Este tipo de funciones se conoce con el nombre funciones arbitrarias y son de utilidad práctica en la solución de problemas en el campo de la ingeniería. Pueden ser empleadas en la misma forma que una función creada a partir de una ecuación. Así con dicha función, se pueden calcular valores de "y" para valores de "x" iguales a: 0.1, 0.3, 0.5, 0.6, 0.7, 0.9 y 1.1:

```
>>map(fx,[0.1,0.2,0.5,0.6,0.7,0.9,1.1])
[0.0995, 0.199, 0.477, 0.565, 0.641, 0.779, 0.903]
```

Y con la siguiente se crea la gráfica entre 0 y 3 (para lo cual la función arbitraria debe extrapolar para los puntos ubicados después de 1):

```
>>plot([fx],0,3)
```

8.4. INTERPOLACIÓN DE LAGRANGE

La interpolación de Lagrange es un método de interpolación polinomial. En el mismo se calculan los valores intermedios ajustando los datos a una ecuación polinomial de igual grado que el número de puntos y que pasa por todos ellos.

En este método la ecuación de interpolación es:

$$y=\sum_{k=1}^{n} y_k L_k(x) \qquad (8.20)$$

Los valores de L_k se calculan con:

$$L_k(x)=\frac{P_k(x)}{Q_k(x)} \qquad \{k=1 \rightarrow n \tag{8.21}$$

$$P_k(x)=\prod_{\substack{i=1 \\ i\neq k}}^{n}(x-x_i) \qquad \{k=1 \rightarrow n \tag{8.22}$$

$$Q_k(x_k)=\prod_{\substack{i=1 \\ i\neq k}}^{n}(x_k-x_i) \qquad \{k=1 \rightarrow n \tag{8.23}$$

Donde "*n*" es el número de datos conocidos, "*x*" es el valor a interpolar, "x_i" y "y_i" son los elementos con los datos de las variables independiente y dependiente respectivamente.

Como se puede observar, los valores de "Q" sólo dependen de los valores tabulados, por lo tanto pueden ser calculados independientemente del valor a interpolar "x", mientras que "P" debe ser calculado para cada valor de "x". Por otra parte, no pueden existir dos valores consecutivos de "x" iguales, porque daría lugar a una división entre cero.

Para ilustrar el método se interpolarán los datos de la siguiente tabla para valores de "x" iguales a 2 y 3.

x	0	1	4	6
y	1	-1	1	-1

Primero se guardan los datos de la tabla en dos vectores y el número de elementos en una variable:

```
>>vx=[0,1,4,6]; vy=[1,-1,1,-1]; n=vx.length
4
```

Entonces se calculan los valores de "Q" con la ecuación (8.23) (recordando que en Javascript los vectores siempre comienzan en cero):

```
>>q=[1,1,1,1]; for(k=0;k<n;k++) for(i=0;i<n;i++) q[k]*= i!=k ? vx[k]-
vx[i] : 1; q
[-24, 15, -24, 60]
```

Ahora se calculan los valores de "P", para x=2, con la ecuación (8.22):

```
>>x=2; p=[1,1,1,1]; for(k=0;k<n;k++) for(i=0;i<n;i++) p[k]*= i!=k ? x-vx[i]
: 1; p
[8, 16, -8, -4]
```

Se calculan los valores de "L" con la ecuación (8.21):

```
>>l=[0,0,0,0]; for (k=0;k<n;k++) l[k]=p[k]/q[k]; l
[-0.3333333333333333, 1.0666666666666667, 0.3333333333333333,
-0.06666666666666667]
```

Finalmente se calcula el valor de "y" (el valor interpolado) para "x=2" con la ecuación (8.20):

```
>>y=0; for(k=0;k<n;k++) y+=vy[k]*l[k]
-1
```

Ahora se repite el proceso para "x=3", excepto que no es necesario calcular de nuevo el valor de "q":

```
>>x=3; p=[1,1,1,1]; for(k=0;k<n;k++) for(i=0;i<n;i++) p[k]*= i!=k ? x-vx[i]
: 1; p
```

```
[6, 9, -18, -6]
>>l=[0,0,0,0]; for (k=0;k<n;k++) l[k]=p[k]/q[k]; l
[-0.25, 0.6, 0.75, -0.1]
>>y=0; for(k=0;k<n;k++) y+=vy[k]*l[k]
2.7755575615628914e-17
```

Que redondeado a 16 dígitos después del punto es 0:

```
>>round(ans,16)
0
```

Como se puede ver en este ejemplo, cada vez que cambia el valor de la variable independiente es necesario volver a calcular los valores de "P" y "L", mientras que el valor de "Q", si no cambian los datos, sólo debe ser calculado una vez.

8.4.1. Algoritmo y código

El algoritmo del método de Lagrange se presenta en la siguiente página y el código respectivo, añadido a la librería "mat205.js", es:

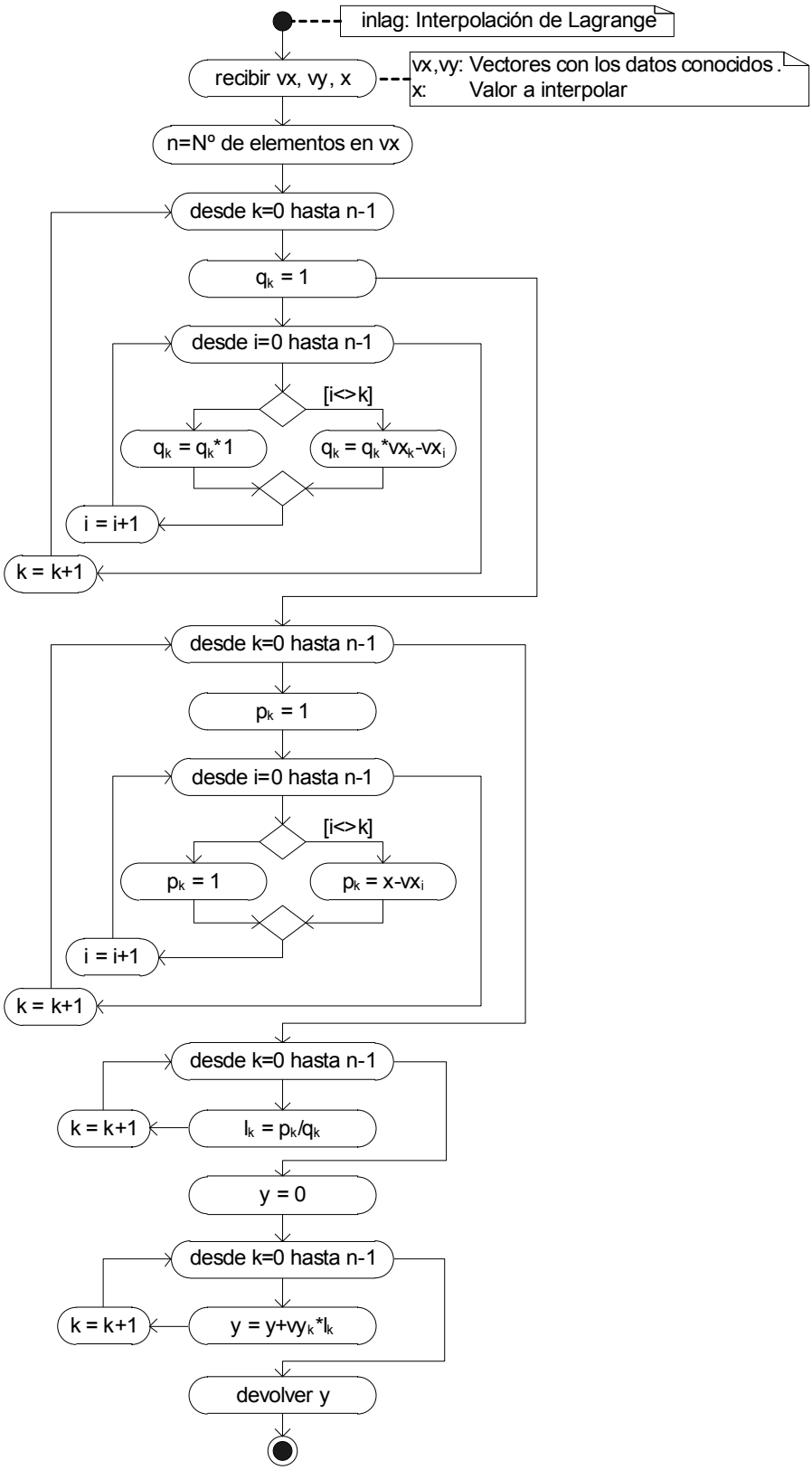
```
function inlag(vx,vy,x){
  var k,i,n=vx.length;
  var q=new Array(n),p=new Array(n),l=new Array(n),y=0;
  for (k=0;k<n;k++){
    q[k]=1;
    for (i=0;i<n;i++) q[k]*= i!=k ? vx[k]-vx[i] : 1;
  }
  for (k=0;k<n;k++){
    p[k]=1;
    for (i=0;i<n;i++) p[k]*= i!=k ? x-vx[i]: 1;
  }
  for (k=0;k<n;k++) l[k]=p[k]/q[k];
  for (k=0;k<n;k++) y+=vy[k]*l[k];
  return y;
}
```

Haciendo correr el programa con los datos del ejemplo manual se obtienen los mismos resultados que en dicho ejemplo:

```
>>vx=[0,1,4,6]; vy=[1,-1,1,-1]
[1, -1, 1, -1]
>>inlag(vx,vy,2)
-1
>>round(inlag(vx,vy,3),16)
0
```

Al igual que con el método de interpolación lineal, es conveniente contar con un generador de funciones arbitrarias para el método de Lagrange, de manera que, cuando se interpolan varios datos para una misma tabla, no sea necesario enviar dichos datos y recalcular el valor de Q para cada nuevo valor de "x". Dicho generador ha sido añadido a la librería "mat205.js" con el nombre "geninlag".

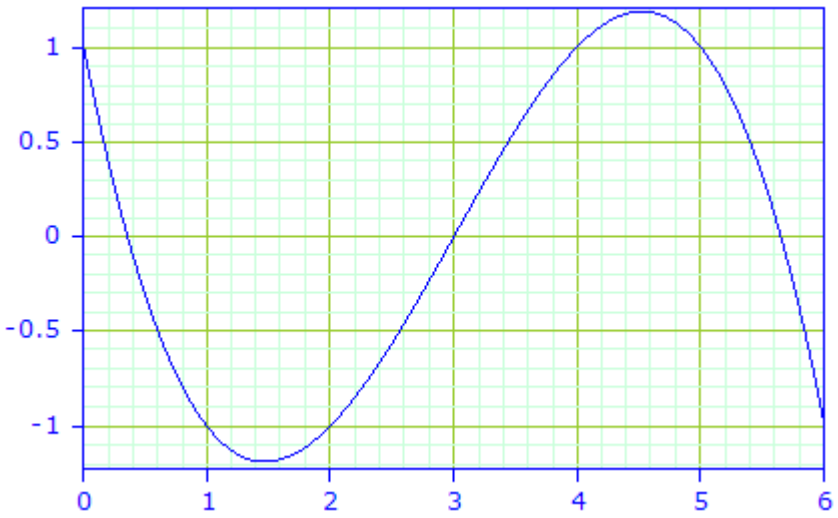
Al igual que con el método de interpolación lineal, la función devuelta por este generador puede ser empleada de la misma forma que una función matemática, sin embargo, con los métodos de interpolación polinomiales, los valores extrapolados no son confiables, por lo que deben ser empleados sólo para interpolar datos.



Por ejemplo, empleando este generador, se puede crear una función para los datos del ejemplo manual y construir la gráfica de la función entre 0 y 6:

```
>>fx2 = geninlag(vx,vy)
function (x){
```

```
    for (k=0;k<n;k++){
      p[k]=1;
      for (i=0;i<n;i++) p[k]*= i!=k ? x-vx[i]: 1;
    }
    for (k=0;k<n;k++) l[k]=p[k]/q[k];
    y=0;
    for (k=0;k<n;k++) y+=vy[k]*l[k];
    return y;
}
>>plot([fx2],0,6)
```



8.5. INTERPOLACIÓN DE NEWTON

Se trata de otro método de interpolación polinomial, en el cual el valor interpolado se calcula con la ecuación:

$$y=b_0+b_1(x-x_0)+b_2(x-x_0)(x-x_1)+\ldots+b_{n-1}(x-x_0)(x-x_1)(x-x_2)\ldots(x-x_{n-2})$$

$$y=\sum_{k=0}^{n-1}\left[b_k\prod_{i=1}^{k-1}(x-x_i)\right] \qquad (8.24)$$

Los valores de "b" se calculan con las siguientes relaciones:

$$f=y$$

$$\left.\begin{array}{l} b_k=f_0 \\ f_i=\dfrac{(f_{i+1}-f_i)}{x_{i+k+1}-x_i} \quad [i=0\rightarrow n-k-1] \end{array}\right\} \quad k=0\rightarrow n-2 \qquad (8.25)$$

$$b_n=f_1$$

En estas ecuaciones *"n"* es el número de datos, *"x_i"* y *"y_i"* son los datos para la variable independiente y dependiente respectivamente y *"x"* es el valor a interpolar.

Como los coeficientes "b" sólo dependen de los valores conocidos pueden ser calculados por separado (tal como ocurre con el parámetro "Q" del método de Lagrange). De esta manera es posible reducir el número de cálculos cuando se interpolan varios valores con los mismos datos.

Al igual que sucede con el método de Lagrange, el método de Newton no puede ser empleado cuando la variable independiente tiene 2 o más valores consecutivos repetidos iguales, pues como se puede observar en las ecuaciones se produciría una división entre cero, además, es necesario también que los valores de la variable independiente estén ordenados ascendentemente.

Para ilustrar el proceso de cálculo en el método de Newton se emplearán los datos de la siguiente tabla, para calcular valores de "y" para valores de "x" iguales a 2 y 5.5:

x	1	4	5	6
y	0	1.3862944	1.6094379	1.7917595

Para ello, primero se guardan los valores de la tabla en dos vectores "vx" y "vy" y el número de datos (puntos) en la variable "n":

```
>>vx=[1,4,5,6]; vy=[0,1.3862944,1.6094379,1.7917595]; n=vx.length;
```

Ahora se aplican las ecuaciones (8.25) para calculara los valores de "b":

```
>>f=copy(vy); b=new Array(n); for (k=0;k<n-1;k++){b[k]=f[0]; for (i=0;i<n-
1-k;i++) f[i]=(f[i+1]-f[i])/(vx[i+k+1]-vx[i]);}; b[n-1]=f[0]; b
[0, 0.4620981333333333, -0.05973865833333329, 0.007865541666666628]
```

Finalmente se calcula el valor interpolado con la ecuación (8.24), sin embargo, la aplicación directa de dicha ecuación implica la repetición innecesaria de operaciones, así para calcular el tercer término se realiza la operación: $(x-x_0)(x-x_1)$, para calcular el cuarto término se realiza la operación: $(x-x_0)(x-x_1)(x-x_2)$, donde como se puede observar se repite el cálculo de: $(x-x_0)(x-x_1)$, para calcular el cuarto término se realiza la operación: $(x-x_0)(x-x_1)(x-x_2)(x-x_3)$, volviéndose a repetir el cálculo de: $(x-x_0)(x-x_1)(x-x_2)$ y así sucesivamente.

Se puede evitar la inútil repetición de cálculos guardando el resultado de la operación anterior en una variable, en este caso "p" y empleando dicho resultado para calcular uno nuevo en la siguiente iteración. Procediendo de esa manera para "x=2", se obtiene:

```
>>x=2; p=1; s=b[0]; for(k=1;k<n;k++){p*=(x-vx[k-1]);s+=b[k]*p;}; s
0.6287686999999996
```

Y para "x=5.5":

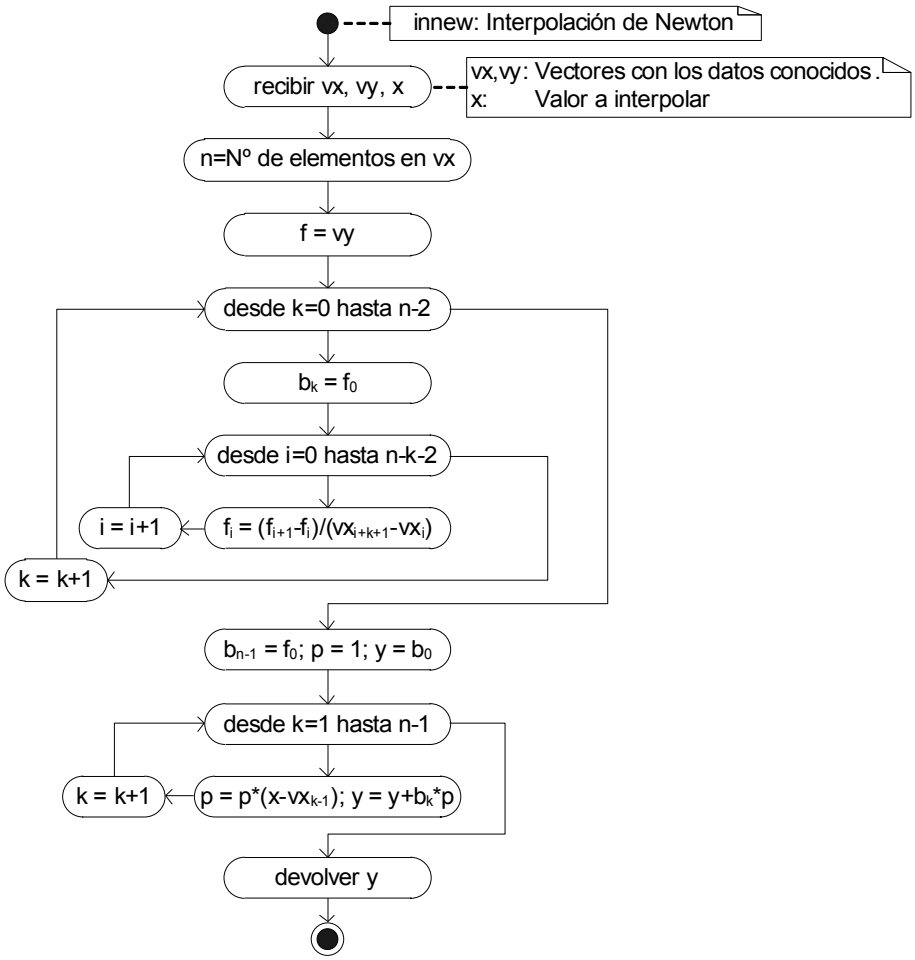
```
>>x=5.5; p=1; s=b[0]; for(k=1;k<n;k++){p*=(x-vx[k-1]);s+=b[k]*p;}; s
1.7027518593750002
```

8.5.1. Algoritmo y código

El algoritmo para la interpolación polinomial por el método de Newton se presenta en la siguiente página.

El código elaborado en base al mismo y añadido a la librería "mat205.js" es:

```
function innew(vx,vy,x){
  var k,i,n=vx.length,p,y,f,b;
  f=copy(vy);
  b=new Array(n);
  for (k=0;k<n-1;k++){
    b[k]=f[0];
    for (i=0;i<n-k-1;i++) f[i]=(f[i+1]-f[i])/(vx[i+k+1]-vx[i]);
  }
  b[n-1]=f[0];
```




```
    p=1; y=b[0];
    for (k=1;k<n;k++) {p*=(x-vx[k-1]); y+=b[k]*p;}
    return y
}
```

Haciendo correr este programa con los datos del ejemplo manual se obtienen los mismos resultados que en dicho ejemplo:

```
>>vx=[1,4,5,6]; vy=[0,1.3862944,1.6094379,1.7917595];
>>innew(vx,vy,2)
0.6287686999999996
>>innew(vx,vy,5.5)
1.7027518593750002
```

Al igual que con la interpolación lineal y Lagrange, es conveniente contar con un generador de funciones para este método, de manera que no sea necesario mandar los datos y calcular los parámetros cada vez que se interpola un valor. Dicho generador ha sido añadido a la librería "mat205.js" con el nombre "geninnew".

Empleando este generador se puede crear una función de interpolación para los datos del ejemplo manual:

```
>>fx3 = geninnew(vx,vy)
function (x){
   p=1; y=b[0];
   for (k=1;k<n;k++) {p*=(x-vx[k-1]); y+=b[k]*p;}
   return y
```

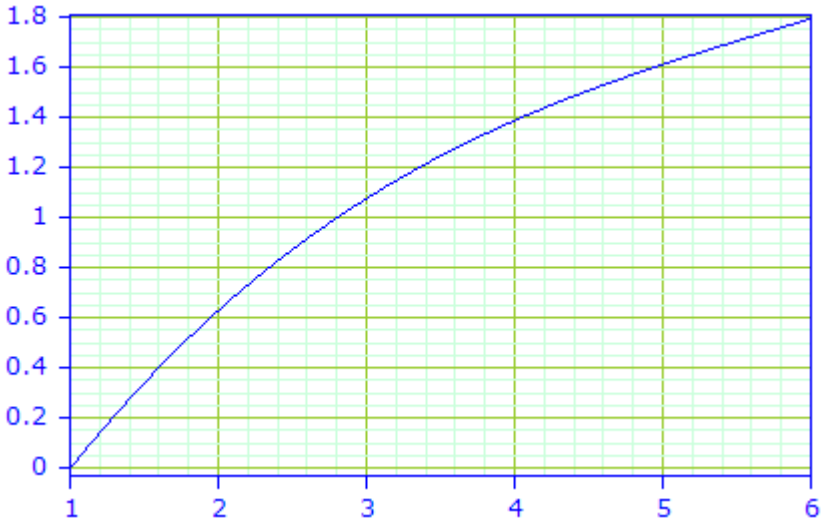
```
}
```

Con la cual se pueden interpolar los valores del ejemplo:

```
>>map(fx3,[2,5.5])
[0.6287686999999996, 1.7027518593750002]
```

Y se puede elaborar la gráfica de esta función entre los límites de los datos (es decir entre 1 y 6):

```
>>plot([fx3],1,6)
```



Se debe recordar no emplear estos métodos para extrapolar datos.

8.6. INTERPOLACIÓN SEGMENTARIA

Mientras que en la interpolación polinomial todos los datos se ajustan a una sola ecuación de enésimo grado, en la interpolación segmentaria cada par de datos se ajusta a una ecuación, generalmente cúbica. De esa manera los datos quedan representados no por una, sino por "n-1" ecuaciones de igual forma pero con coeficientes y constantes calculados para cada uno de los segmentos.

Al ajustar un conjunto reducido de datos a una ecuación, por lo general se consigue representar mejor el comportamiento intermedio de los datos que con una sola ecuación que pase por todos ellos.

La ecuación más empleada para este tipo de ajuste es la ecuación cúbica, por lo tanto las "n-1" ecuaciones que representan los "n" datos tienen la forma:

$$s_j(x)=a_j+b_j(x-x_j)+c_j(x-x_j)^2+d_j(x-x_j)^3 \quad \{j=0\rightarrow n-2 \tag{8.26}$$

Donde s_j es la ecuación ajustada para el segmento "j", "x" es el valor a interpolar, mismo que debe estar comprendido entre x_j y x_{j+1}; a_j, b_j, c_j y d_j son los coeficientes de la ecuación cúbica para el segmento "j".

Para calcular los coeficientes de la ecuación cúbica, se toma en cuenta que en el punto intermedio entre dos segmentos cualesquiera, las ecuaciones correspondientes a dichos segmentos deben devolver el mismo resultados para el mencionado punto, es decir se debe cumplir que:

$$\begin{gathered} s_j(x_{j+1})=s_{j+1}(x_{j+1})=y_{j+1} \\ a_j+b_j(x_{j+1}-x_j)+c_j(x_{j+1}-x_j)^2+d_j(x_{j+1}-x_j)^3= \\ a_{j+1}+b_{j+1}(x_{j+1}-x_j)+c_{j+1}(x_{j+1}-x_j)^2+d_{j+1}(x_{j+1}-x_j)^3=y_{j+1} \end{gathered} \tag{8.27}$$

Si se denomina "h_j" a la diferencia entre dos valores consecutivos de "x":

$$h_j = x_{j+1} - x_j \tag{8.28}$$

La anterior ecuación puede ser reescrita como:

$$a_j + b_j h_j + c_j h_j^2 + d_j h_j^3 = a_{j+1} = y_{j+1} \tag{8.29}$$

Por lo tanto los coeficientes "a", de la ecuación cúbica, son iguales a los valores conocidos de la variable dependiente, es decir:

$$a_j = y_j \tag{8.30}$$

Para calcular los coeficientes restantes ("b", "c" y "d") se hace que las derivadas de las ecuaciones cúbicas a ambos lados de un punto intermedio cualquiera sean iguales, es decir que tengan la misma pendiente, con ello no sólo se logra despejar una de las incógnitas, sino que además se segura que la curva resultante sea uniforme:

$$\begin{gathered} s'_j(x_{j+1}) = s'_{j+1}(x_{j+1}) \\ b_j + 2c_j h_j + 3d_j h_j^2 = b_{j+1} + 2c_{j+1}(x_{j+1} - x_{j+1}) + 3d_{j+1}(x_{j+1} - x_{j+1})^2 \\ b_j + 2c_j h_j + 3d_j h_j^2 = b_{j+1} \end{gathered} \tag{8.31}$$

Para garantizar aún más la uniformidad de la curva resultante y disminuir el número de incógnitas, se igualan también las derivadas segunda de las ecuaciones cúbicas a ambos lados de un punto cualquiera:

$$\begin{gathered} s''_j(x_{j+1}) = s''_{j+1}(x_{j+1}) \\ 2c_j + 6d_j h_j = 2c_{j+1} + 6d_{j+1}(x_{j+1} - x_{j+1}) \\ c_j + 3d_j h_j = c_{j+1} \end{gathered} \tag{8.32}$$

Despejando d_j de esta ecuación, se tiene:

$$d_j = \frac{c_{j+1} - c_j}{3h_j} \tag{8.33}$$

Reemplazando esta ecuación en la ecuación (8.31) se obtiene:

$$\begin{gathered} b_j + 2c_j h_j + \frac{3(c_{j+1} - c_j)}{3h_j} h_j^2 = b_{j+1} \\ b_j + 2c_j h_j + c_{j+1} h_j - c_j h_j = b_{j+1} \\ b_j + 2c_j h_j + c_{j+1} h_j - c_j h_j = b_{j+1} \\ b_j + (c_j + c_{j+1}) h_j = b_{j+1} \end{gathered} \tag{8.34}$$

Y reemplazando la ecuación (8.33) en la ecuación (8.29):

$$\begin{gathered} a_j + b_j h_j + c_j h_j^2 + \frac{c_{j+1} - c_j}{3h_j} h_j^3 = a_{j+1} \\ a_j + b_j h_j + \frac{3c_j h_j^2 + c_{j+1} h_j^2 - c_j h_j^2}{3} = a_{j+1} \\ b_j = \frac{a_{j+1} - a_j}{h_j} - \frac{2c_j h_j^2 + c_{j+1} h_j^2}{3h_j} \\ b_j = \frac{a_{j+1} - a_j}{h_j} - \frac{(2c_j + c_{j+1}) h_j}{3} \end{gathered} \tag{8.35}$$

Finalmente, reemplazando (8.35) en (8.34) y reajustando índices, se obtiene un sistema de ecuaciones lineales donde las únicas incógnitas son los los coeficientes "c":

$$\frac{a_{j+1}-a_j}{h_j}-\frac{(2c_j+c_{j+1})h_j}{3}+(c_j+c_{j+1})h_j=\frac{a_{j+2}-a_{j+1}}{h_{j+1}}-\frac{(2c_{j+1}+c_{j+2})h_{j+1}}{3}$$

$$\frac{-2c_jh_j-c_{j+1}h_j+3c_jh_j+3c_{j+1}h_j}{3}+\frac{(2c_{j+1}+c_{j+2})h_{j+1}}{3}=3\left(\frac{a_{j+2}-a_{j+1}}{h_{j+1}}-\frac{a_{j+1}-a_j}{h_j}\right) \quad (8.36)$$

$$h_jc_j+2(h_j+h_{j+1})c_{j+1}+h_{j+1}c_{j+2}=3\left(\frac{a_{j+2}-a_{j+1}}{h_{j+1}}-\frac{a_{j+1}-a_j}{h_j}\right)$$

No obstante, esta expresión sólo genera "n-2" ecuaciones lineales, porque el máximo valor posible para "j" es "n-3", esto porque sólo existen "n-1" segmentos y al estar presente en la expresión el término "c_{j+2}", se deduce fácilmente que el mayor valor posible de "j" es: j+2=n-1 => j=n-3.

Por ejemplo si se tienen 6 puntos (n=6), se forma un sistema con 4 ecuaciones lineales con 6 incógnitas (c_0, c_1, c_2, c_3, c_4 y c_5):

$$\begin{aligned}
h_0c_0+2(h_0+h_1)c_1+h_1c_2 &= 3\left(\frac{a_2-a_1}{h_1}-\frac{a_1-a_0}{h_0}\right) = p_0\\
h_1c_1+2(h_1+h_2)c_2+h_2c_3 &= 3\left(\frac{a_3-a_2}{h_2}-\frac{a_2-a_1}{h_1}\right) = p_1\\
h_2c_2+2(h_2+h_3)c_3+h_3c_4 &= 3\left(\frac{a_4-a_3}{h_3}-\frac{a_3-a_2}{h_2}\right) = p_2\\
h_3c_3+2(h_3+h_4)c_4+h_4c_5 &= 3\left(\frac{a_5-a_4}{h_4}-\frac{a_4-a_3}{h_3}\right) = p_3
\end{aligned}$$

Donde los elementos "p_i" simplemente representan los resultados obtenidos al reemplazar los valores en el lado derecho de cada una de las ecuaciones.

Por consiguiente, para resolver este sistema, se requieren 2 valores de "c". Como no se tiene ninguna información con relación al comportamiento de los datos antes del primer segmento y después del último, se puede asumir que en el primer segmento y después del último, las ecuaciones cúbicas no cuentan con el término de segundo grado, es decir que en dichos segmentos los valores de "c" son cero, por lo tanto se puede asumir que tanto c_0 como c_{n-1} son cero.

Con estos valores se puede resolver el sistema de ecuaciones y calcular los valores restantes de "c". Para visualizar mejor el sistema de ecuaciones que se forma, se escribe la ecuación (8.36) en forma matricial:

$$\begin{bmatrix}
h_0 & 2(h_0+h_1) & h_1 & 0 & 0 & \dots & 0 & 0 & 0\\
0 & h_1 & 2(h_1+h_2) & h_2 & 0 & \dots & 0 & 0 & 0\\
0 & 0 & h_2 & 2(h_2+h_3) & h_3 & \dots & 0 & 0 & 0\\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots\\
0 & 0 & 0 & 0 & 0 & \dots & h_{n-3} & 2(h_{n-3}+h_{n-2}) & h_{n-2}
\end{bmatrix} * \begin{bmatrix} c_1\\ c_2\\ c_3\\ \dots\\ c_{n-2}\end{bmatrix} = \begin{bmatrix} p_0\\ p_1\\ p_2\\ \dots\\ p_{n-3}\end{bmatrix} \quad (8.37)$$

Donde los elementos "p", como ya se vio, se calculan con el lado derecho de la ecuación (8.36), es decir con la expresión:

$$p_j = 3\left(\frac{a_{j+2}-a_{j+1}}{h_{j+1}} - \frac{a_{j+1}-a_j}{h_j}\right) \qquad (8.38)$$

Como se puede ver, el sistema de ecuaciones lineales que se forma es un sistema tridiagonal, por lo que puede ser resuelto con uno de los métodos estudiados para este fin (gausstd y gseideltd).

Una vez calculados los valores de "c", los valores de "b" se calculan con la ecuación (8.35), los valores de "d" con la ecuación (8.33), finalmente el valor de "y" (el valor interpolado) se calcula con la ecuación (8.26), ubicando previamente el segmento en el que se encuentra el valor de "x" (el valor a interpolar).

Para ilustrar el procedimiento, se empleará la interpolación segmentaria para calcular valores de "y" para valores de "x" iguales a 2 y 5.5, con los datos de la siguiente tabla:

x	1	4	5	6
y	0	1.3862944	1.6094379	1.7917595

Primero se guardan los datos de la tabla así como el número de datos:

```
>>vx=[1,4,5,6]; vy=[0,1.3862944,1.6094379,1.7917595];n=vx.length;
```

Se asigna a la variable "a" los valores de "y" (ecuación (8.30)) y se calculan los valores de "h" con la ecuación (8.28):

```
>>a=vy; h=new Array(n-1); for (i=0;i<n-1;i++) h[i]=vx[i+1]-vx[i]; h
[3, 1, 1]
```

Se calculan los elementos de la matriz tridiagonal con las expresiones de la ecuación (8.36):

```
>>mt=new Array(n-2); for(j=0;j<n-2;j++){mt[j]=new Array(4); mt[j]
[0]=h[j];mt[j][1]=2*(h[j]+h[j+1]);mt[j][2]=h[j+1];mt[j][3]=3*((a[j+2]-
a[j+1])/h[j+1]-(a[j+1]-a[j])/h[j]);}; mt
[[3, 8, 1, -0.7168638999999994], [1, 4, 1, -0.1224657000000009]]
```

Entonces se resuelve este sistema de ecuaciones con "gausstd":

```
>>c=gausstd(mt)
[-0.08854806129032249, -0.008479409677419605]
```

Al que se le añaden los valores asumidos (los dos ceros):

```
>>c.unshift(0);c.push(0);c
[0, -0.08854806129032249, -0.008479409677419605, 0]
```

Con los valores de "c" se calculan los de "b" y "d" (ecuaciones (8.35) y (8.33)):

```
>>b=new Array(n-1); for(j=0;j<n-1;j++) b[j]=(a[j+1]-a[j])/h[j]-(2*c[j]
+c[j+1])*h[j]/3; b
[0.5506461946236558, 0.2850020107526884, 0.18797453978494627]
>>d=new Array(n-1); for(j=0;j<n-1;j++) d[j]=(c[j+1]-c[j])/(3*h[j]); d
[-0.009838673476702498, 0.026689550537634294, 0.0028264698924732015]
```

Con todos los coeficientes calculados, se puede crear la función de interpolación:

```
>>fx=function(x){var h,j=0; while(x>vx[j+1] && j<n-2) j++; h=x-vx[j]; re-
turn a[j]+b[j]*h+c[j]*h*h+d[j]*h*h*h;};
```

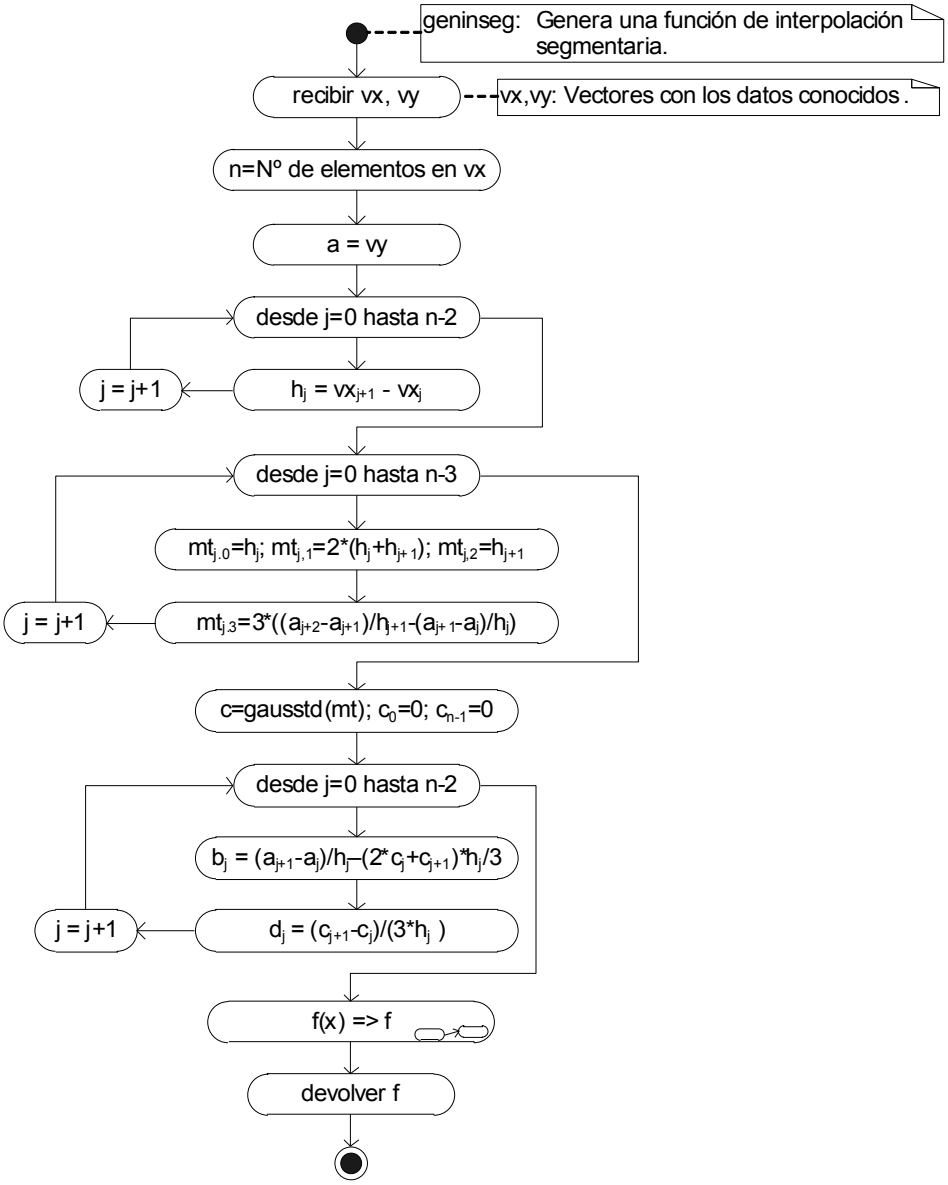
Y con la misma calcular los valores pedidos:

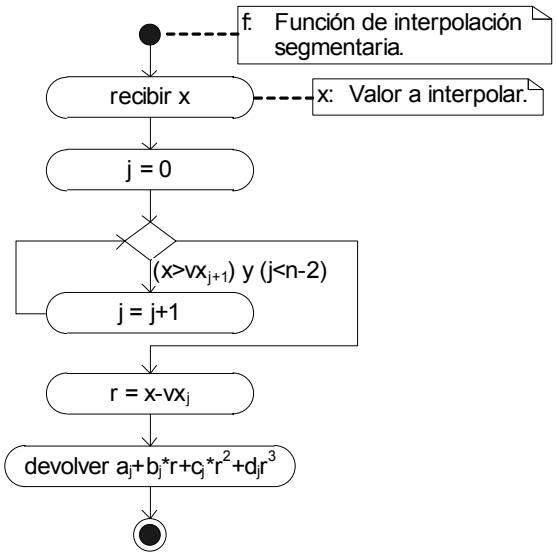
```
>>map(fx,[2,5.5])
[0.5408075211469533, 1.7016586262096776]
```

8.6.1. Algoritmo y código

Debido a la gran cantidad de cálculos que implica la determinación de los coeficientes, el método de interpolación segmentaria se emplea casi exclusivamente para interpolar varios valores a partir de un mismo conjunto de datos. Por lo tanto es de mayor utilidad contar con un generador de funciones arbitrarias, en lugar de una función de interpolación. El algoritmo para dicho generador se presenta en los siguientes diagramas, siendo el código respectivo, añadido a la librería "mat205.js", el siguiente:

```
function geninseg(vx,vy){
  var n=vx.length,a=vy,j,c,h=new Array(n-1),r;
  var b=new Array(n-1),d=new Array(n-1),mt=new Array(n-2);
  for (j=0;j<n-1;j++) h[j]=vx[j+1]-vx[j];
  for (j=0;j<n-2;j++){
    mt[j]=new Array(4);
    mt[j][0]=h[j]; mt[j][1]=2*(h[j]+h[j+1]); mt[j][2]=h[j+1];
    mt[j][3]=3*((a[j+2]-a[j+1])/h[j+1]-(a[j+1]-a[j])/h[j]);
  }
```






```
    c=gausstd(mt); c.unshift(0); c.push(0);
    for (j=0;j<n-1;j++) {
      b[j]=(a[j+1]-a[j])/h[j]-(2*c[j]+c[j+1])*h[j]/3;
      d[j]=(c[j+1]-c[j])/(3*h[j]);
    }
    return function(x){
      j=0;
      while (x>vx[j+1] && j<n-2) j++;
      r=x-vx[j];
      return a[j]+b[j]*r+c[j]*r*r+d[j]*r*r*r;
    }
  }
```

Por ejemplo,se puede emplear este programa para crear una función de interpolación para los datos del ejemplo manual (se puede evitar visualizar la función resultante escribiendo un punto y coma ";" al final de la instrucción):

```
>>fx=geninseg([1,4,5,6],[0,1.3862944,1.6094379,1.7917595])
function (x){
   j=0;
   while (x>vx[j+1] && j<n-2) j++;
   r=x-vx[j];
   return a[j]+b[j]*r+c[j]*r*r+d[j]*r*r*r;
 }
```

Con esta función, como era de esperar, se obtienen los mismos resultados que en el ejemplo manual:

```
>>map(fx,[2,5.5])
[0.5408075211469533, 1.7016586262096776]
```

8.7. EJEMPLOS

1. Ajuste los siguientes datos a la ecuación: $y=a+bx^2+cx^4+d\ln(x)$. Determine gráfica y numéricamente cuan apropiado resulta el ajuste y calcule los valores de "y" para "x" = 1, 2, 3, 4, 5, 6, 7, 8, 9 y 10.

x	4.0	3.2	6.8	9.3	1.2	0.7	5.5	7.4	6.5	2.4
Y	6.2	5.3	9.1	12.0	3.3	1.5	7.9	10.6	8.7	6.3

Puesto que la ecuación tiene la forma de la ecuación (Error: No se encuentra la fuente de referencia1), la mejor alternativa consiste en ajustarla empleando el método de los mínimos cuadrados, pero como además la ecuación resultante es empleada para obtener varios resultados, lo más práctico es generar la función ajustada con "genmincuad".

Los vectores con las funciones y los datos son:

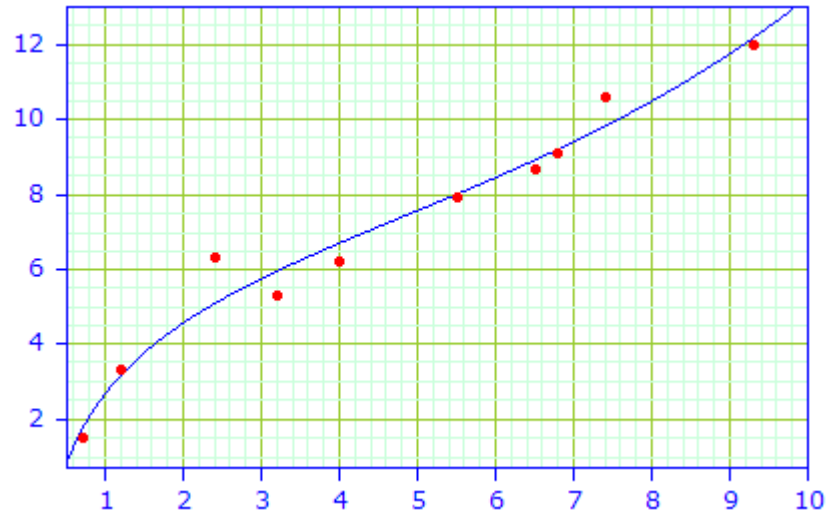
```
>>vx=[4,3.2,6.8,9.3,1.2,0.7,5.5,7.4,6.5,2.4];
>>vy=[6.2,5.3,9.1,12,3.3,1.5,7.9,10.6,8.7,6.3];
>>vf=new Array(4); vf[0]=function(x){return 1;}; vf[1]=function(x){return
x*x;}; vf[2]=function(x){return x*x*x*x;}; vf[3]=function(x){return
log(x);};
```

Con los que se obtiene la función ajustada:

```
>>fx=genmincuad(vf,vx,vy);
```

Para juzgar visualmente cuan bueno es el ajuste se elabora la gráfica de la función de ajustada y los puntos conocidos:

```
>>plot([fx,[vx,vy]],0.5,10)
```



Como se puede ver, los datos se distribuyen uniformemente a ambos lados de la curva, por lo que visualmente se puede concluir que el ajuste es bueno.

Para juzgar numéricamente cuan bueno es el ajuste se calcula el coeficiente de correlación:

```
>>corr(fx,vx,vy)
0.9842448608441499
```

Como este valor es cercano a uno se concluye que el ajuste es bueno, corroborando así la apreciación visual.

Ahora, con la ecuación ajustada se calculan los valores de "y" para los valores dados de "x", los cuales se redondean al primer dígito para que tengan la misma precisión que los datos:

```
>>round(map(fx,[1,2,3,4,5,6,7,8,9,10]),1)
[2.7, 4.6, 5.8, 6.7, 7.6, 8.5, 9.4, 10.5, 11.8, 13.3]
```

Si la ecuación a ajustar no se encontrara en forma de la ecuación (8.1), se tendría que recurrir al método Simplex o del descenso acelerado, en cuyo caso la función ajustada tendría que ser creada manualmente empleando los coeficientes calculados.

2. Cree funciones de interpolación para los datos de la siguiente tabla con los métodos de interpolación lineal, interpolación de Lagrange, interpolación de Newton e interpolación segmentaria. Grafique las funciones resultantes (conjuntamente los puntos) y calcule valores de "y" para valores de "x" iguales a 1, 4, 6, 13, 17, 23, 25.5, 26.5, 28, 30, 42 y 45.

x	3	5	7	11	14	16	19	21	24	25	26	27	29	33	36	40	44
y	0	0	0	0	1	3	13	22	29	26	18	8	3	0	0	0	0

Como el mismo conjunto de datos se emplea para calcular varios resultados, se deben generar las funciones de interpolación con "geninlin", "geninlag", "geninnew" y "geninseg".

Primero se guardan los datos en dos vectores, recordando que deben estar ordenados con respecto a la variable independiente:

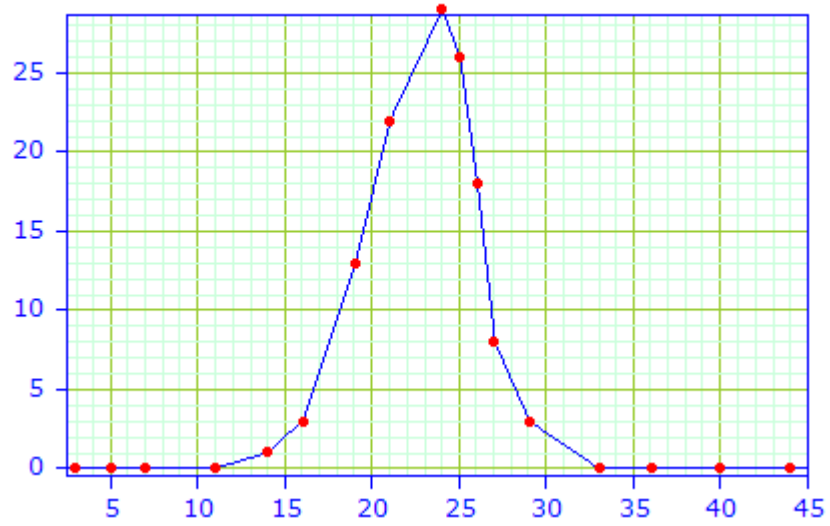
```
>>vx=[3,5,7,11,14,16,19,21,24,25,26,27,29,33,36,40,44];
>>vy=[0,0,0,0,1,3,13,22,29,26,18,8,3,0,0,0,0];
```

Ahora se genera la función de interpolación para el método de interpolación lineal:

```
>>fx=geninlin(vx,vy);
```

Y con la misma se elabora la gráfica de la función de interpolación y de los datos conocidos:

```
>>plot([fx,[vx,vy]],2.5,45)
```



Como se puede ver las líneas rectas entre los datos predicen de manera aceptable los valores intermedios (los valores a interpolar).

Los valores interpolados, para los datos proporcionados (redondeados al primer dígito para concordar con la precisión de los datos proporcionados) son:

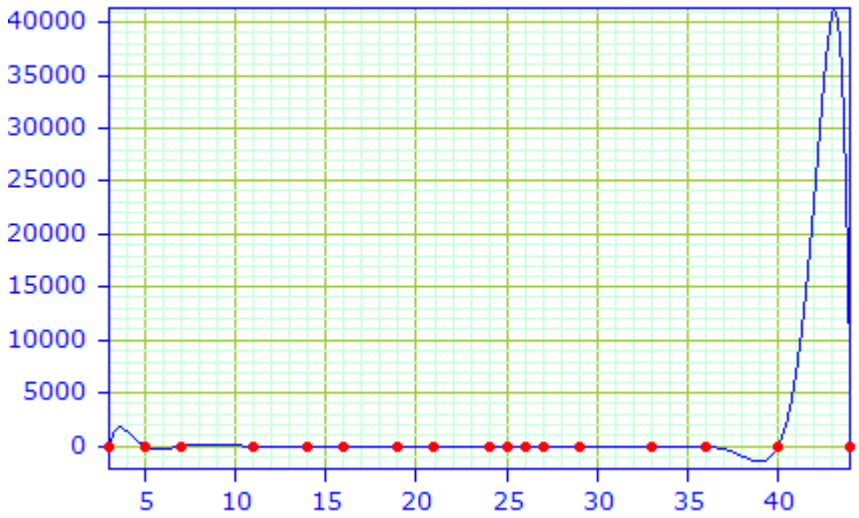
```
>>dx=[1,4,6,13,17,23,25.5,26.5,28,30,42,45]; round(map(fx,vx),1)
[0, 0, 0, 0.7, 6.3, 26.7, 22, 13, 5.5, 2.3, 0, 0]
```

Se procede de manera similar con el método de Lagrange. Primero se genera la función de interpolación:

```
>>fx1=geninlag(vx,vy);
```

Se elabora la gráfica para determinar cuan confiable es el método:

```
>>plot([fx1,[vx,vy]],3,44)
```



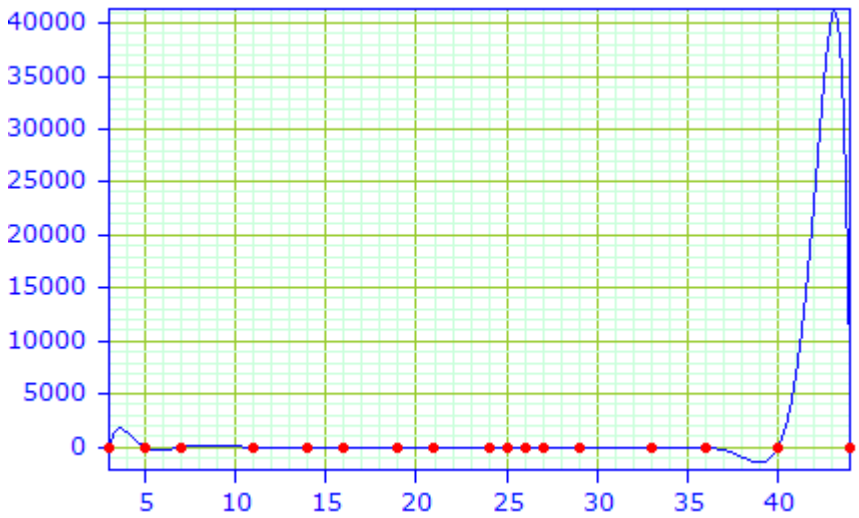
Como se puede ver, en este caso el método de Lagrange predice mal los valores intermedios, por lo que no es el método adecuado para llevar a cabo la interpolación, algo que se corrobora al calcular los valore pedidos:

```
>>round(map(fx1,dx),1)
[-167193.5, 1387.2, -274.2, -10.7, 3, 27.8, 22.5, 12.9, 1.7, 9.3, 23896,
-298043.6]
```

Que evidentemente no corresponde a los resultados correctos.

Se procede de forma similar con el método de Newton, que al igual que Lagrange, predice mal los datos intermedios:

```
>>fx2=geninnew(vx,vy);
>>plot([fx2,[vx,vy]],3,44)
```



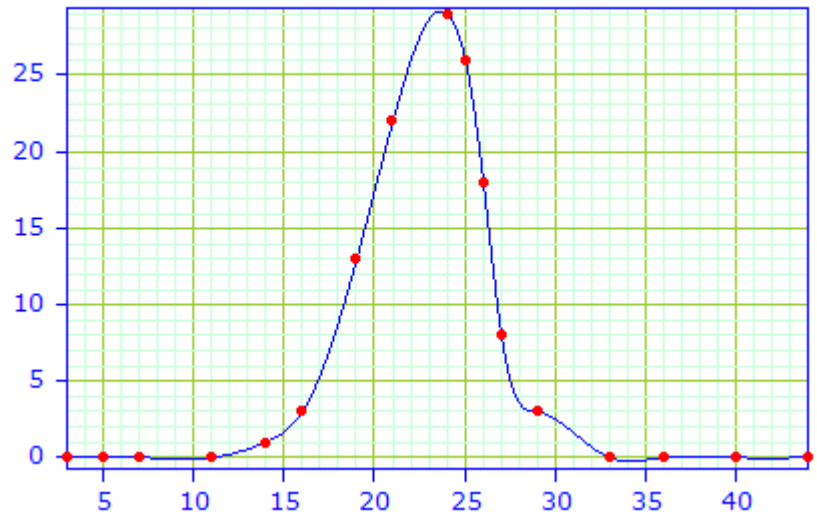

```
>>round(map(fx2,dx),1)
[-167193.5, 1387.2, -274.2, -10.7, 3, 27.8, 22.5, 12.9, 1.7, 9.3, 23896,
-298043.6]
```

Se procede de igual forma con el método de interpolación segmentaria. La función de interpolación se obtiene con:

```
>>fx3=geninseg(vx,vy);
```

Siendo la gráfica de la misma:

```
>>plot([fx3,[vx,vy]],3,44)
```



Como se puede ver el método de interpolación segmentaria predice bastante bien los valores intermedios (con excepción de unos 4 segmentos ubicados en la parte inferior de la curva).

Los valores interpolados con este método son:

```
>>round(map(fx3,dx),1)
[0, 0, 0, 0.5, 5.5, 28.6, 22.6, 12.7, 3.6, 2.5, 0, 0]
```

8.8. EJERCICIOS

Como en los anteriores temas para presentar los ejercicios que resuelva en la calculadora, copie las instrucciones en un editor de texto y luego, al momento de presentar el trabajo, vuelva a copiarlos y ejecutarlos en la calculadora. Alternativamente, puede guardar el ejercicio resuelto como una página HTML, haciendo clic derecho en página y eligiendo la opción "guardar como" (o su equivalente) y guardando la página como página web completa (o fichero único) asignándole un nombre adecuado como ejercicio1.html.

1. Ajuste los datos de la siguiente tabla a la ecuación: $y=a\,x^2+\frac{b}{x}+c\,e^{\frac{x}{8}}$.

   Luego grafique la ecuación ajustada y los puntos conocidos, calcule su coeficiente de correlación y calcule los valores de "y" para los siguientes valores de "x": 2, 4, 7, 9, 11, 12, 13, 14, 16, 17, 18.

x	2.3	4.5	6.7	8.4	9.2	10.1	11.2	12.6	14.5	16.3	17.2	18.1
y	11	41	90	141	169	205	251	318	421	545	592	655

2. Ajuste los datos de la siguiente tabla a la ecuación: $y=a\,x^3+b\sqrt{x}+\frac{c}{\sqrt[3]{x}}$.

   Luego grafique la ecuación ajustada y los puntos conocidos, calcule el coeficiente de correlación y resuelva la ecuación ajustada empleando el método "Incremental" (valor inicial 2, incremento 1).

x	3.8	1.3	9.2	3.4	1.0	5.1	5.7	9.0	8.0	6.4	6.6	8.1
y	15	8	124	13	8	27	35	116	84	46	50	87

3. Ajuste los datos de la siguiente tabla a la ecuación: $y=a\ln(x)+b\sin(x)-c\cos(x)$. Luego grafique la ecuación ajustada y los puntos conocidos, calcule el coeficiente de correlación y resuelva la ecuación ajustada con el método de Regula Falis encontrando el segmento de solución con el método incremental (valor inicial 5, incremento -1).

x	1	3	4	5	8	11	14	15	20
y	1	3	2	2	5	4	6	6	7

4. Empleando los datos de la siguiente tabla calcule valores de "y" para valores de "x" iguales a 3.9, 4.8, 6.1, 6.57, 6.7 y 7.1, con los métodos de interpolación lineal, Lagrange, Newton y segmentaria. En todos los casos, para juzgar visualmente el método, elabore la gráfica de la función de interpolación conjuntamente los puntos conocidos.

x	3.5	4	4.5	5	5.5	6	6.2	6.5	6.8
y	0.002	0.030	0.155	0.396	0.658	0.903	0.933	0.975	0.993

5. Empleando los datos de la siguiente tabla calcule valores de "y" para valores de "x" iguales a 0.5, 2.5, 3.2, 4.3, 5.5, 6.5, 8.5, 10.5, 13.5, 15.5, 18.5, 20.5, 22.5 y 24, con los métodos de interpolación lineal, Lagrange, Newton y segmentaria. En todos los casos, para juzgar visualmente el método, elabore la gráfica de la función de interpolación conjuntamente los puntos conocidos.

x	1	2	3	3.5	4	4.5	5	6	7	8	9	10	12	13	14	15	16	17	18	19	20	21	22	23
y	0	0	0.5	1.5	3	5	9	11.8	12	12.5	12.5	12.5	12.5	12.4	12.0	11.5	10.5	9.5	8.3	7.0	5.5	4.0	3.0	2.5

6. Elabore un módulo que resuelva la siguiente función (encuentre el valor de x_i) empleando el método de Newton Raphson (valor inicial 0.5). Los valores de "y_i", necesarios en el cálculo, deben ser obtenidos por interpolación de los datos de la tabla con el método de Newton:

x_i	0	0.05	0.10	0.15	0.20	0.25	0.30	0.35
y_i	0	0.022	0.052	0.087	0.131	0.187	0.265	0.385

$$f(x_i)=\frac{k_x}{k_y}+\frac{y-y_i}{x-x_i}=0$$

$$k_y=\frac{k'_y}{(1-y)_m};\quad (1-y)_m=\frac{(1-y_i)-(1-y)}{\ln\left(\frac{1-y_i}{1-y}\right)}$$

$$k_x=\frac{k'_x}{(1-x)_m};\quad (1-x)_m=\frac{(1-x)-(1-x_i)}{\ln\left(\frac{1-x}{1-x_i}\right)}$$

$$k'_x=1.967\text{x}10^{-3};\quad k'_y=1.465\text{x}10^{-3};\quad y=0.39;\quad x=0.1$$

9. INTEGRACIÓN Y DIFERENCIACIÓN NUMÉRICA

La integración y diferenciación numérica se emplean cuando las funciones a integrar o diferenciar se encuentran en forma tabular, gráfica o son muy complejas como para integrarlas o derivarlas analíticamente.

Sin embargo, aún en situaciones donde las funciones son sencillas, se suele recurrir a la integración y diferenciación numérica, simplemente porque es más rápido y sencillo resolver una integral o calcular el valor de una derivada con la ayuda de un dispositivo programable.

En la diferenciación numérica se aplica fórmulas, deducidas en base al concepto de la derivada, que buscan aproximar la diferencia infinitesimal mediante diferencias finitas.

En la integración numérica se cuenta con varios métodos que emplean algún tipo de simplificación para calcular el área bajo la curva.

9.1. DIFERENCIACIÓN NUMÉRICA

Como se dijo, el cálculo numérico de las derivadas se basa en el concepto de la derivada, es decir:

$$\frac{dx}{dy} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \qquad (9.1)$$

Numéricamente se consigue una aproximación al límite empleando valores suficientemente pequeños de "$\grave{a}x$", de manera que "$\grave{a}x$" tienda a cero. En su aproximación más simple, se toman dos valores alrededor de "x" de manera que su diferencia se aproxime a cero (tienda a cero), con estos valores se calculan los correspondientes valores de "y" (reemplazando los valores de "x" en la función a diferenciar) y con los mismos se calcula el valor de la ecuación (9.1).

Por ejemplo para calcular la derivada primera de una función "f" en un punto cualquiera "x", se toman dos valores cercanos a "x": uno ubicado al lado izquierdo (x-h) y otro al derecho (x+h) (donde "h" es una fracción pequeña de "x") y reemplazando estas expresiones en la ecuación (9.1), se obtiene:

$$\frac{dy}{dx} \approx \frac{f(x+h) - f(x-h)}{(x+h) - (x-h)} = \frac{f(x+h) - f(x-h)}{2 \cdot h} \qquad (9.2)$$

Que, es la fórmula de diferencia central empleada en el método de Newton-Raphson. Se conoce como fórmula de diferencia central porque los puntos se calculan a ambos lados de "x", que es el valor para el cual se quiere calcular la derivada.

Si en lugar de dos valores a ambos lados de "x" se toma uno solo a la izquierda: "x-h", se obtiene:

$$\frac{dy}{dx} \approx \frac{f(x) - f(x-h)}{x - (x-h)} = \frac{f(x) - f(x-h)}{h} \qquad (9.3)$$

Que se conoce como fórmula de diferencia hacia atrás, porque el valor con el que se calcula la diferencia ($\grave{a}x$) se calcula con uno ubicado antes del valor "x" (x-h).

Si, por el contrario, se toma sólo un valor a la derecha de "x": "x+h", se obtiene:

$$\frac{dy}{dx}\approx\frac{f(x+h)-f(x)}{(x+h)-x}=\frac{f(x+h)-f(x)}{h} \qquad (9.4)$$

Que se conoce como fórmula de diferencia hacia adelante, porque el valor con el que se calcula la diferencia (**à***x*) se encuentra delante del valor "x"(x+h).

Estas ecuaciones pueden ser deducidas tomando no sólo uno o dos puntos, sino cuatro o más, antes y/o después del valor de "x". Mientras más puntos se tomen para el cálculo de Δx más exacto es el resultado obtenido, así, de las tres fórmulas deducidas, la más exacta es la de diferencia central (ecuación 9.2), porque en su deducción se emplean dos puntos.

La deducción de las fórmulas para las derivadas de segundo, tercer y cuarto orden siguen el mismo principio, pero en lugar de la función original se emplean las fórmulas deducidas para las derivadas de orden inferior. Así para obtener la derivada segunda se emplean las fórmulas de la derivada primera (pues como se sabe la derivada segunda se obtiene derivando a su vez la derivada primera), para la tercer las fórmulas de la derivada segunda y así sucesivamente.

En función al número de puntos que se toman para el cálculo de "Δx" se tienen diferentes tipos de errores: de primer orden si sólo se toma un valor; de segundo orden si se toman dos valores y de cuarto orden si se toman 4 valores.

A continuación se presentan las fórmulas de diferenciación (central, hacia adelante y hacia atrás) hasta de cuarto orden, con errores hasta de cuarto orden:

9.1.1. Fórmulas de diferencia central. Error de orden h^2

$$\begin{aligned}
f'(x)&=\frac{f(x+h)-f(x-h)}{2h}\\
f''(x)&=\frac{f(x+h)-2f(x)+f(x-h)}{h^2}\\
f'''(x)&=\frac{f(x+2h)-2f(x+h)+2f(x-h)-f(x-2h)}{2h^3}\\
f''''(x)&=\frac{f(x+2h)-4f(x+h)+6f(x)-4f(x-h)+f(x-2h)}{h^4}
\end{aligned} \qquad (9.5)$$

9.1.2. Fórmulas de diferencia central. Error de orden h^4

$$\begin{aligned}
f'(x)&=\frac{-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h)}{12h}\\
f''(x)&=\frac{-f(x+2h)+16f(x+h)-30f(x)+16f(x-h)-f(x-2h)}{12h^2}\\
f'''(x)&=\frac{f(x+3h)+8f(x+2h)-13f(x+h)+13f(x-h)-8f(x-2h)+f(x-3h)}{8h^3}\\
f''''(x)&=\frac{-f(x+3)+12f(x+2h)-39f(x+h)+56f(x)-39f(x-h)+12f(x-2h)-f(x-3h)}{6h^4}
\end{aligned} \qquad (9.6)$$

9.1.3. Fórmulas de diferencia hacia adelante. Error de orden h

$$
\begin{aligned}
f'(x) &= \frac{f(x+h)-f(x)}{h} \\
f''(x) &= \frac{f(x+2h)-2f(x+h)+f(x)}{h^2} \\
f'''(x) &= \frac{f(x+3h)-3f(x+2h)+3f(x+h)-f(x)}{h^3} \\
f''''(x) &= \frac{f(x+4h)-4f(x+3h)+6f(x+2h)-4f(x+h)+f(x)}{h^4}
\end{aligned}
\tag{9.7}
$$

9.1.4. Fórmulas de diferencia hacia adelante. Error de orden h^2

$$
\begin{aligned}
f'(x) &= \frac{-f(x+2h)+4f(x+h)-3f(x)}{2h} \\
f''(x) &= \frac{-f(x+3h)+4f(x+2h)-5f(x+h)+2f(x)}{h^2} \\
f'''(x) &= \frac{-3f(x+4h)+14f(x+3h)-24f(x+2h)+18f(x+h)-5f(x)}{2h^3} \\
f''''(x) &= \frac{-2f(x+5h)+11f(x+4h)-24f(x+3h)+26f(x+2h)-14f(x+h)-3f(x)}{h^4}
\end{aligned}
\tag{9.8}
$$

9.1.5. Fórmulas de diferencia hacia atrás. Error de orden h

$$
\begin{aligned}
f'(x) &= \frac{f(x)-f(x-h)}{h} \\
f''(x) &= \frac{f(x)-2f(x-h)+f(x-2h)}{h^2} \\
f'''(x) &= \frac{f(x)-3f(x-h)+3f(x-2h)-f(x-3h)}{h^3} \\
f''''(x) &= \frac{f(x)-4f(x-h)+6f(x-2h)-4f(x-3h)+f(x-4h)}{h^4}
\end{aligned}
\tag{9.9}
$$

9.1.6. Fórmulas de diferencia hacia atrás. Error de orden h^2

$$
\begin{aligned}
f'(x) &= \frac{3f(x)-4f(x-h)+f(x-2h)}{2h} \\
f''(x) &= \frac{2f(x)-5f(x-h)+4f(x-2h)-f(x-3h)}{h^2} \\
f'''(x) &= \frac{5f(x)-18f(x-h)+24f(x-2h)-14f(x-3h)+3f(x-4\mathrm{h})}{2h^3} \\
f''''(x) &= \frac{3f(x)-14f(x-h)+26f(x-2h)-24f(x-3h)+11f(x-4h)-2f(x-5h)}{h^4}
\end{aligned}
\tag{9.10}
$$

Como se dijo, cuanto mayor es el orden de la fórmula más exacto es el resultado obtenido, no obstante, las fórmulas de mayor orden implican también un mayor número de operaciones, lo que a su vez repercute en un mayor error en los resultados (debido a los errores de redondeo). Por esta razón (a menos que se trabaje con números de alta precisión) es recomendable trabajar con fórmulas de primer o segundo orden.

Igualmente, en teoría, cuanto más pequeño sea el valor de "h" más exacto será el resultado (porque más se acerca Δx al límite 0), una vez más esto no es cierto en la práctica debido a los errores de redondeo (para valores muy pequeños de "h" el error de redondeo puede ser muy grande). Por ello el valor de "h" suele estar comprendido entre 10^{-2} y 10^{-6}, siendo más grande cuanto mayor es el orden de la derivada.

Una vez que se tienen las fórmulas de diferenciación numérica, simplemente se las programa para calcular la derivada requerida.

Por ejemplo, a continuación se presentan las funciones para calculara las las derivadas primera a cuarta, empleando las fórmulas de diferencia central (funciones que han sido añadidas a la librería "hpvlib.js"):

```
function df1(f,x){
  var h=x*1e-6;
  return (f(x+h)-f(x-h))/(2*h);
}
function df2(f,x){
  var h=x*1e-5;
  return (f(x+h)-2*f(x)+f(x-h))/(h*h);
}
function df3(f,x){
  var h=x*1e-4;
  return (f(x+2*h)-2*f(x+h)+2*f(x-h)-f(x-2*h))/(2*h*h*h);
}
function df4(f,x){
  var h=x*1e-3;
  return (f(x+2*h)-4*f(x+h)+6*f(x)-4*f(x-h)+f(x-2*h))/(h*h*h*h);
}
```

Donde los valores de "h" se han elegido por prueba y error, pero no constituyen una regla y en consecuencia pueden ser modificados en caso de comprobar que otros valores resultan más adecuados.

Por ejemplo para calcular las derivadas primera a tercera de la siguiente función, para "x=7.2":

$$f(x)=x^3+2x^2+3x+4=0 \tag{9.11}$$

Se programa la función:

```
>>f=function(x){return x*x*x+2*x*x+3*x+4;}
function (x){return x*x*x+2*x*x+3*x+4;}
```

Y luego se emplean "df1", "df2" y "df3" para calcular las derivadas en "x=7.2":

```
>>df1(f,7.2)
187.31999999330483
```

```
>>df2(f,7.2)
47.19997232566836

>>df3(f,7.2)
5.999620858039511
```

Que en todos los casos son valores próximos a los correctos (187.32, 47.2 y 6). Si se redondean los resultados al tercer dígito, se obtienen los resultados exactos:

```
>>round(df1(f,7.2),3)
187.32

>>round(df2(f,7.2),3)
47.2

>>round(df3(f,7.2),3)
6
```

Y se procede de la misma forma para cualquier función, por ejemplo, para calcular las derivadas primera a cuarta de la siguiente función en "x=3.5" (redondeando los resultados a 2 dígitos después del punto):

$$f(x)=x^6+2x^5+3x^4+4x^3+5x^2+6x+7 \qquad (9.12)$$

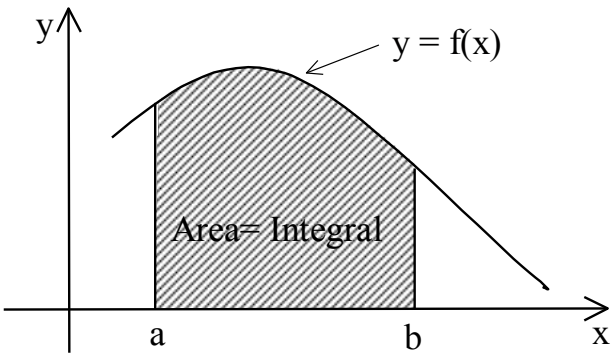
Se escribe:

```
>>f2=function(x){return
x*x*x*x*x*x+2*x*x*x*x*x+3*x*x*x*x+4*x*x*x+5*x*x+6*x+7;}
function (x){return
x*x*x*x*x*x+2*x*x*x*x*x+3*x*x*x*x+4*x*x*x+5*x*x+6*x+7;}
>>round(df1(f2,3.5),1)
5354.4
>>round(df2(f2,3.5),1)
6751.9
>>round(df3(f2,3.5),1)
6891
>>round(df4(f2,3.5),1)
5322
```

Que así redondeados corresponden a los resultados exactos.

9.2. INTEGRACIÓN POR EL MÉTODO DEL TRAPECIO

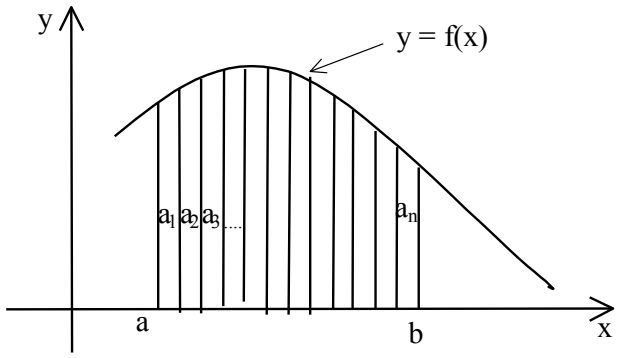
Como se sabe, la integral de una función es el área bajo la curva, es decir:



El valor numérico se obtiene justamente calculando esta área:

$$\int_a^b f(x)\cdot dx = Área\ bajo\ la\ curva \qquad (9.13)$$

Con ese fin, la mayoría de los métodos dividen el área bajo la curva en una serie de segmentos:

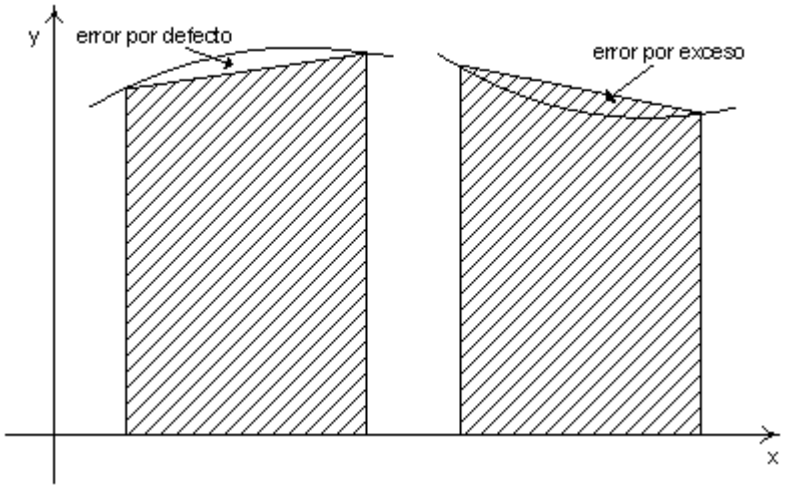


Entonces la integral (el área bajo la curva) se calcula sumando las áreas de cada uno de estos segmentos, es decir:

$$\int_a^b f(x)\cdot dx = \sum_{i=1}^{n} a_i \qquad (9.14)$$

Los métodos difieren unos de otros en la forma en que calculan estos segmentos, así en el caso del método del trapecio se asume que los segmentos tienen la forma de un trapecio, o lo que es lo mismo, se asume que los segmentos están unidos entre si por líneas rectas.

Como se puede observar en la siguiente figura, el asumir que los segmentos están unidos por una línea recta da lugar a un error en el cálculo, error que es más grande mientras más grande es el segmento y menos se ajusta la curva a una línea recta:



Por eso, a menos que los puntos estén muy juntos, este método es uno de los menos exactos.

El área para cualquier segmento "i" se calcula con la ecuación del trapecio:

$$a_i=\frac{(x_{i+1}-x_i)\cdot(y_{i+1}+y_i)}{2} \qquad ((9.15)$$

Si se denomina h_i al ancho del segmento:

$$h_i=x_{i+1}-x_i \qquad (9.16)$$

La ecuación (9.15) queda en la forma:

$$a_i=\frac{h_i}{2}\left(y_i+y_{i+1}\right) \qquad (9.17)$$

Si la integral se calcula a partir de datos tabulados, el cálculo debe ser llevado a cabo con las ecuaciones (9.17), (9.16) y (9.14). Sin embargo, si la integral se calcula a partir de una expresión analítica, los cálculos pueden ser simplificados si todos los segmentos se toman del mismo ancho, es decir $h_1 = h_2 = \ldots = h_n$, entonces la integral, la sumatoria de los segmentos, toma la forma:

$$\int_a^b f(x)\cdot dx=\sum_{i=1}^{n}\left(\frac{h}{2}(y_1+y_2)+\frac{h}{2}(y_2+y_3)+\cdots+\frac{h}{2}(y_n+y_{n+1})\right)$$

$$\int_a^b f(x)\cdot dx=\frac{h}{2}\cdot\sum_{i=1}^{n}\left(y_1+2\,y_2+2\,y_3+\cdots+2\,y_n+y_{n+1}\right)$$

$$\int_a^b f(x)\cdot dx=\frac{h}{2}\cdot\left(y_1+2\cdot\sum_{i=2}^{n}y_i+y_{n+1}\right) \qquad (9.18)$$

Que es la ecuación conocida como la fórmula del Trapecio.

Para comprender mejor como se aplica esta ecuación se calculará manualmente la siguiente integral, dividiendo el área bajo la curva en 20 segmentos (n=20):

$$\int_{1.2}^{4.5}\left(x^3+2x^2+3x+4\right)dx \qquad (9.19)$$

Primero se programa la función:

```
>>f=function(x){return x*x*x+2*x*x+3*x+4;}
function (x){return x*x*x+2*x*x+3*x+4;}
```

Se inicializan variables y se calcula el valor del incremento:

```
>>a=1.2;b=4.5;n=20;h=(b-a)/n
0.16499999999999998
```

Con el incremento se calculan los valores de "x_i" desde "x_2" hasta "x_n" empleando la función "range", que recibe como datos el límite inferior, el incremento y el límite superior (si no se especifica el incremento su valor por defecto es 1):

```
>>x=range(a+h,h,b-h)
[1.365, 1.53, 1.695, 1.86, 2.025, 2.19, 2.355, 2.52, 2.685, 2.85, 3.015,
3.18, 3.345, 3.5100000000000002, 3.6750000000000003, 3.8400000000000003,
4.005, 4.17, 4.335]
```

Para estos valores se calculan los correspondientes de "y_i":

```
>>y=map(f,x)
[14.364752124999999, 16.853377000000002, 19.700827375000003, 22.934056,
26.580015624999998, 30.665658999999998, 35.217938875, 40.263808000000004,
```

```
45.830219125, 51.944125, 58.632478375000005, 65.92223200000001,
73.840338625, 82.41375100000002, 91.66942187500001, 101.63430400000003,
112.335350125, 123.799513, 136.05374537499998]
```

Finalmente, con la ecuación (9.18), se calcula el valor de la integral, llevando a cabo la sumatoria con la función "sum", que calcula la sumatoria de los elementos del vector o matriz que se le manda:

```
>>r=h/2*(f(a)+2*sum(y)+f(b))
203.1681980625
```

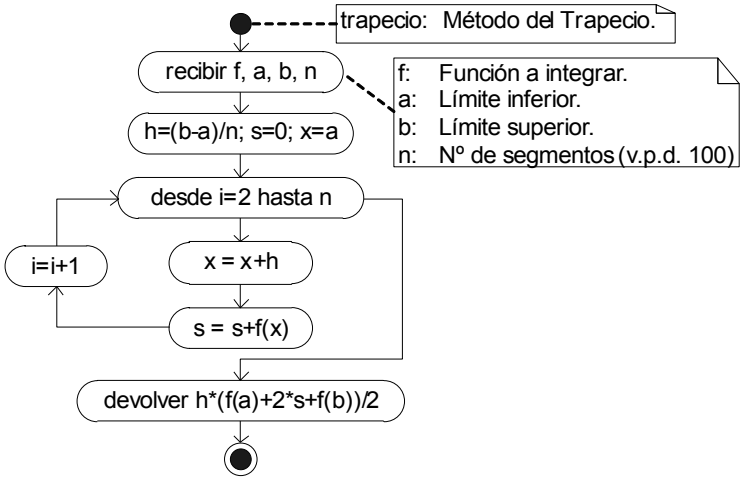
El resultado más exacto, calculado programando la integral analítica es:

```
>>fi=function(x){return 0.25*x*x*x*x+2*x*x*x/3+1.5*x*x+4*x;};
>>fi(4.5)-fi(1.2)
203.010225
```

Como se puede ver, el resultado obtenido con el método del Trapecio es solo aproximado. Si se quiere mejorar la exactitud del resultado se debe incrementar el número de divisiones "n" (o lo que es lo mismo, disminuir el valor del incremento "h").

9.2.1. Algoritmo y código

Como los puntos intermedios x_i, y_i sólo se utilizan una vez en los cálculos, no es necesario almacenarlos, ese hecho se toma en cuenta en el siguiente algoritmo para lograr un ahorro tanto en memoria como tiempo:



El código respectivo, añadido a la librería "hpvlib.js", es:

```
function trapecio(f,a,b,n) {
  if (n==null) n=100;
  var h=(b-a)/n, s=0, x=a;
  for (i=2;i<=n;i++)
    {x+=h; s+=f(x);}
  return h*(f(a)+2*s+f(b))/2;
}
```

Haciendo correr el programa con los datos del ejemplo manual, con 20 divisiones, se obtiene el mismo resultado que dicho ejemplo (si la función "f" aún está en memoria, no es necesario volver a programarla):

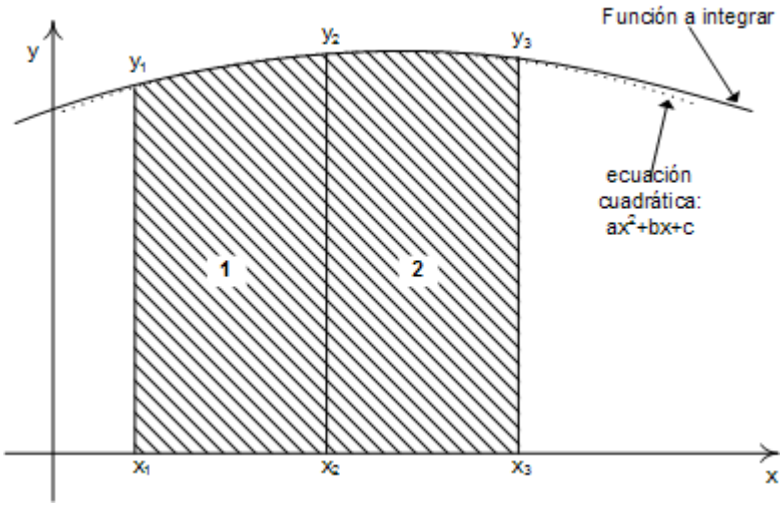
```
>>trapecio(f,1.2,4.5,20)
203.1681980625
```

Con 200 segmentos, en lugar de 20, se obtiene un resultado más exacto:

```
>>trapecio(f,1.2,4.5,200)
203.01180473062652
```

9.3. INTEGRACIÓN POR EL MÉTODO DE SIMPSON

En el método de Simpson, a diferencia del método del Trapecio, se asume que los segmentos están unidos por una ecuación cuadrática:



Como se trata de una ecuación cuadrática se requieren 3 puntos (2 segmentos) para resolver la ecuación (hallar los valores de los coeficientes). Por lo tanto, en el método de Simpson se requiere siempre un número par de segmentos.

$$Area_{dos\ segmentos} = \int_{x_1}^{x_3} \left(a\,x^2 + b x + c\right)\cdot dx = \frac{a}{3}\left(x_3^3 - x_1^3\right) + \frac{b}{2}\left(x_3^2 - x_1^2\right) + c\cdot\left(x_3 - x_1\right) \qquad (9.20)$$

Los coeficientes pueden ser calculados tomando dos segmentos cualesquiera. Si los puntos del segmento son "i", "j" y "k", el sistema de ecuaciones que se forma es:

$$\begin{aligned} y_i &= a + b\,x_i + c\,x_i^2 \\ y_j &= a + b\,x_j + c\,x_j^2 \\ y_k &= a + b\,x_k + c\,x_k^2 \end{aligned} \qquad (9.21)$$

Si el ancho de los segmentos es constante e igual a "h", la solución del sistema de ecuaciones puede ser expresado en la siguiente forma:

$$\begin{aligned} a &= \frac{y_i - y_j + y_k}{2h^2} \\ b &= \frac{y_k - y_i}{2h} \\ c &= y_j \end{aligned} \qquad (9.22)$$

Reemplazando estas expresiones en la ecuación (9.20), se obtiene:

$$Area_{dos\ segmentos} = \frac{h}{3}\left(y_i + 4\,y_j + y_k\right) \qquad (9.23)$$

Entonces la sumatoria de "n" segmentos (siendo "n" un número par) es:

$$\int_a^b f(x)\cdot dx=\frac{h}{3}\left(y_1+4y_2+y_3\right)+\frac{h}{3}\left(y_3+4y_4+y_5\right)+\frac{h}{3}\left(y_5+4y_6+y_7\right)+\cdots+\frac{h}{3}\left(y_{n-1}+4y_n+y_{n+1}\right)$$

$$\int_a^b f(x)\cdot dx=\frac{h}{3}\left(y_1+4\left(y_2+y_4+y_6+\cdots+y_n\right)+2\left(y_3+y_5+y_7+\cdots+y_{n-1}\right)+y_{n+1}\right)$$

$$\int_a^b f(x)\cdot dx=\frac{h}{3}\left(y_1+4\sum_{i=2,4,6}^{n} y_i+2\sum_{i=3,5}^{n-1} y_i+y_{n+1}\right) \qquad (9.24)$$

Esta ecuación es conocida como la fórmula de Simpson.

Para comprender mejor como se aplica esta ecuación se calculará la integral (9.19), siendo el número de segmentos 20 (n=20).

Primero (como de costumbre) se programa la ecuación:

```
>>f=function(x){return x*x*x+2*x*x+3*x+4;};
```

Se inician las variables y se calcula el incremento "h":

```
>>a=1.2; b=4.5; n=20; h=(b-a)/n;
```

Luego se calculan los valores de "x", para los incrementos impares, y los correspondientes valores de "y":

```
>>x=range(a+2*h,2*h,b-2*h)
[1.5299999999999998, 1.8599999999999999, 2.19, 2.52, 2.85, 3.18,
3.5100000000000002, 3.8400000000000003, 4.17]
>>yi=map(f,x)
[16.853377, 22.934055999999998, 30.665658999999998, 40.263808000000004,
51.944125, 65.92223200000001, 82.41375100000002, 101.63430400000003,
123.799513]
```

Se hace lo mismo para los incrementos pares:

```
>>x=range(a+h,2*h,b-h)
[1.365, 1.6949999999999998, 2.025, 2.355, 2.685, 3.015, 3.345,
3.6750000000000003, 4.005, 4.335]
>>yp=map(f,x)
[14.364752124999999, 19.700827374999996, 26.580015624999998, 35.217938875,
45.830219125, 58.632478375000005, 73.840338625, 91.66942187500001,
112.335350125, 136.05374537499998]
```

Finalmente, con los valores de "y" para los puntos pares e impares, se calcula la integral con la ecuación (9.23):

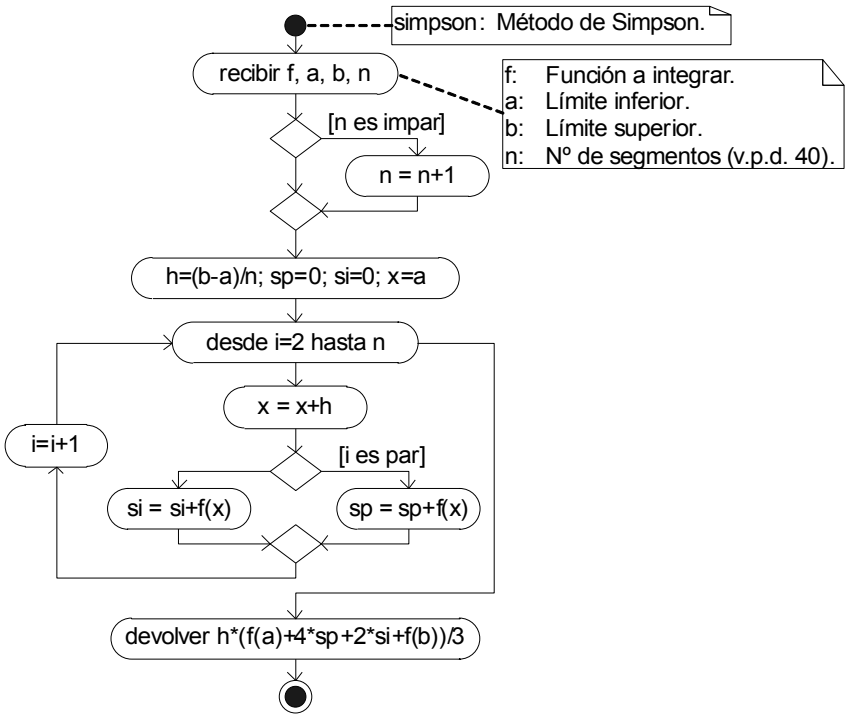
```
>>r=h/3*(f(a)+4*sum(yp)+2*sum(yi)+f(b))
203.010225
```

Que es el resultado exacto.

El método de Simpson devuelve resultados más exactos que el método del trapecio porque, casi siempre, una ecuación cuadrática se ajusta mejor a los puntos de la función que una ecuación lineal.

9.3.1.1. Algoritmo y código

El algoritmo que automatiza el proceso de cálculo se presenta en la siguiente página. En el mismo, al igual que en el método del trapecio, no se guardan los puntos intermedios x_i, y_i, pues se utilizan en los cálculos una sola vez (cuando se calculan).



El código respectivo, añadido a la librería "hpvlib.js", es:

```
function simpson(f,a,b,n) {
  if (n==null) n=40;
  if (n%2) n++;
  var h=(b-a)/n, sp=0, si=0, x=a, d=true;
  for (i=2;i<=n;i++) {
    x+=h;
    if (d)
      sp+=f(x);
    else
      si+=f(x);
    d=!d;
  }
  return h*(f(a)+4*sp+2*si+f(b))/3;
}
```

Haciendo correr el programa con los datos del ejemplo manual se obtiene, como era de esperar, el mismo resultado que en dicho ejemplo:

```
>>f=function(x){return x*x*x+2*x*x+3*x+4;};
>>simpson(f,1.2,4.5,20)
203.010225
```

Con este método, inclusive con 10 divisiones se obtiene el resultado exacto:

```
>>simpson(f,1.2,4.5,10)
203.010225
```

9.3.2. Método de Romberg

El método de Romberg es básicamente la regla del trapecio a la que se aplica el método de extrapolación de Richardson.

En este método se aplica la regla del trapecio para calcular inicialmente dos áreas: una tomando toda el área como un solo segmento $(I_{0,1})$ y la segunda

dividiendo el área en dos segmentos $(I_{0,2})$. Entonces con la fórmula de Richardson se obtiene un resultado extrapolado a partir de estos dos valores.

La fórmula de extrapolación de Richardson es:

$$I_{n,k}=\frac{4^n \cdot I_{n-1,k+1}-I_{n-1,k}}{4^n-1} \qquad (9.25)$$

Donde "n" es el nivel de extrapolación, "k" es el k-ésimo valor dentro del nivel al que pertenece ("n" o "n-1"), así $I_{n-1,k+1}$ es el valor de la integral en el nivel de extrapolación "n-1" y en la posición "k+1", $I_{n-1,k}$, es el valor de la integral en el nivel de extrapolación "n-1" y en la posición "k".

Aplicando esta ecuación a los valores calculados con el método del trapecio (que como no son extrapolados pertenecen al nivel 0), se tiene:

$$I_{1,1}=\frac{4^1 \cdot I_{0,2}-I_{0,1}}{4^1-1}$$

Si el valor extrapolado $(I_{1,1})$ es aproximadamente igual al último valor del nivel anterior $(I_{0,2})$, el proceso concluye, siendo el valor de la integral el valor extrapolado $(I_{1,1})$. Caso contrario se vuelve a aplicar la regla del trapecio para calcular un nuevo valor de la integral $(I_{0,3})$, pero empleando esta vez un número de segmentos igual al doble del cálculo anterior $(I_{0,2})$ y como en el último cálculo se emplearon 2 segmentos, en este nuevo cálculo se emplean 4.

Con el nuevo valor de la integral y el penúltimo valor de este nivel $(I_{0,2})$ se realiza otra extrapolación:

$$I_{1,2}=\frac{4^1 \cdot I_{0,3}-I_{0,2}}{4^1-1}$$

Ahora se cuenta con dos valores en el nivel de extrapolación 1 $(I_{1,1}$ e $I_{1,2})$, por lo que es posible realizar una extrapolación de segundo nivel:

$$I_{2,1}=\frac{4^2 \cdot I_{1,2}-I_{1,1}}{4^2-1}$$

Se comprueba entonces si este valor es aproximadamente igual al último valor del nivel anterior $(I_{1,2})$ y de ser así el proceso concluye, siendo el valor de la integral el último valor extrapolado $(I_{2,1})$. Caso contrario se vuelve a calcular un nuevo valor de la integral con la regla del Trapecio $(I_{0,4})$, pero empleando ahora un número de segmentos igual al doble del cálculo anterior y como en el cálculo anterior se emplearon 4 segmentos, ahora se emplean 8.

Con el nuevo valor de la integral y el penúltimo valor de este nivel $(I_{0,3})$ se realiza una nueva extrapolación:

$$I_{1,3}=\frac{4^1 \cdot I_{0,4}-I_{0,3}}{4^1-1}$$

Ahora en el nivel 1 se tienen tres valores. Con los dos últimos de este nivel $(I_{1,2}$ e $I_{1,3})$ se realiza una segunda extrapolación:

$$I_{2,2}=\frac{4^2 \cdot I_{1,3}-I_{1,2}}{4^2-1}$$

Con lo que se tienen dos valores en el nivel 2 ($I_{2,1}$ e $I_{2,2}$). Entonces se puede realizar una tercera extrapolación al tercer nivel:

$$I_{3,1}=\frac{4^3\cdot I_{2,2}-I_{2,1}}{4^3-1}$$

Ahora se compara este valor con el último del nivel anterior ($I_{2,2}$) y si son aproximadamente iguales el proceso concluye, siendo el resultado el último valor extrapolado, caso contrario se vuelve a repetir el proceso antes descrito, calculando un nuevo valor de la integral con un número de segmentos igual al doble del cálculo anterior y realizando luego las extrapolaciones respectivas.

Como se deduce del proceso descrito, en el método sólo se requieren los dos últimos valores de cualquiera de los niveles de extrapolación, porque los valores anteriores, una vez empleados, ya no vuelven a ser empleados. En consecuencia sólo es necesario guardar los dos últimos valores de todos los niveles de extrapolación, además, dado que en cada iteración siempre se emplea un número de segmentos igual al doble del anterior, es posible aprovechar los valores calculados en iteraciones previas para calcular el nuevo valor de la integral.

Si los valores de cada nivel de extrapolación se guardan en los vectores "r" y "s", la fórmula de Richardson puede ser reescrita como:

$$s_{i+1}=\frac{4^i\cdot s_i-r_i}{4^i-1}\quad \{i=1\rightarrow n \tag{9.26}$$

Y si en la regla del trapecio se aprovechan los valores calculados en la iteración previa, la fórmula del trapecio puede ser reescrita como:

$$s_1=\frac{r_1+h\cdot\sum_{i=1}^{m} f\left(x+(i-1)\cdot h\right)}{2} \tag{9.27}$$

Donde "r_1" es la integral calculada en la iteración anterior, "m" es el número de segmentos y "h" es el ancho del segmento (el cual se reduce a la mitad en cada iteración). Para aplicar esta ecuación, la variable "x" debe ser inicializada en "a+h/2".

En la primera iteración existe un sólo segmento, por lo tanto "h" es igual a "b-a" (siendo "a" y "b" los límites inferior y superior de la integral). En esta iteración, el valor de la integral es:

$$r_1=\frac{h}{2}\cdot\left(f(a)+f(b)\right) \tag{9.28}$$

Para comprender mejor como se aplica el método, se calculará manualmente la integral (9.19) aplicando el mismo.

Primero se programa la función y se inician variables:

```
>>f=function(x){return x*x*x+2*x*x+3*x+4;}; a=1.2, b=4.5; n=1; m=1; h=(b-
a); r=new Array(); s=new Array();
```

Se calcula el primer valor de la integral con la ecuación (9.28):

```
>>r[0]=h/2*(f(a)+f(b))
266.19944999999996
```

Y se calcula el segundo valor de la integral con la ecuación (9.27):

```
>>x=a+h/2; s0=0; for(i=1;i<=m;i++) s0+=f(x+(i-1)*h); s[0]=(r[0]+h*s0)/2
218.80753124999995
```

Con estos valores se extrapola aplicando la ecuación (9.26):

```
>>for(i=0;i<n;i++) s[i+1]=(pow(4,i+1)*s[i]-r[i])/(pow(4,i+1)-1); s
[218.80753124999995, 203.01022499999996]
```

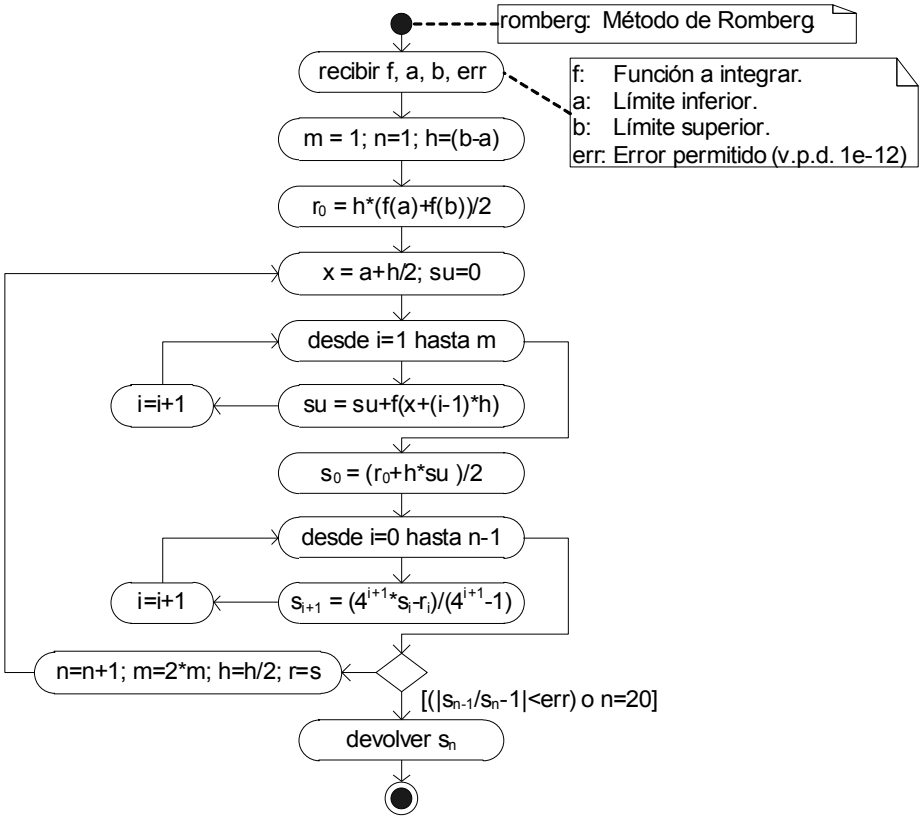
Como estos valores no son iguales se repite el proceso, actualizando los valores de "n", "m" y "h":

```
>>n++; m*=2; h/=2; r=s.slice(); x=a+h/2; s0=0; for(i=1;i<=m;i++) s0+=f(x+
(i-1)*h); s[0]=(r[0]+h*s0)/2
206.95955156249994
>>for(i=0;i<n;i++) s[i+1]=(pow(4,i+1)*s[i]-r[i])/(pow(4,i+1)-1); s
[206.95955156249994, 203.0102249999999, 203.0102249999999]
```

Y como lo dos últimos valores son iguales, el proceso concluye siendo el último valor extrapolado la solución, es decir 203.0102249999999 (que es la solución exacta, excepto por un pequeño error de redondeo).

9.3.2.1. Algoritmo y código

El algoritmo que automatiza el proceso de cálculo es el siguiente:



Como se puede ver, en este algoritmo se emplea un límite de iteraciones fijo igual a 20, esto porque casi nunca es necesario realizar más de 10 extrapolaciones, por lo que 20 es un límite seguro.

El código respectivo, añadido a la librería "hpvlib.js", es:

```
function romberg(f,a,b,err) {
  if (err==null) err=1e-12;
  var i,j,x,m=1,n=1,h=(b-a)/n,r=new Array(20),s=new Array(20),vt,su;
  r[0]=h*(f(a)+f(b))/2;
```

```
    while (true) {
      x=a+h/2;
      su=0;
      for (i=1;i<=m;i++)
        su+=f(x+(i-1)*h);
      s[0]=(r[0]+h*su)/2;
      vt=4;
      for (i=0;i<n;i++){
        s[i+1]=(vt*s[i]-r[i])/(vt-1);
        vt*=4;
      }
      if (abs(s[n-1]/s[n]-1)<err || n==20) return s[n];
      n++; m*=2; h/=2; for (i=0;i<n;i++) r[i]=s[i];
    }
  }
```

Haciendo correr el programa con los datos del ejemplo manual y con el error por defecto (1e-12), se obtienen los mismos resultados que en dicho ejemplo:

```
>>f=function(x){return x*x*x+2*x*x+3*x+4;};
>>romberg(f,1.2,4.5)
203.0102249999999
```

9.4. EJERCICIOS

Como en los anteriores temas para presentar los ejercicios que resuelva en la calculadora, copie las instrucciones en un editor de texto y luego, al momento de presentar el trabajo, vuelva a copiarlos y ejecutarlos en la calculadora. Alternativamente, puede guardar el ejercicio resuelto como una página HTML, haciendo clic derecho en página y eligiendo la opción "guardar como" (o su equivalente) y guardando la página como página web completa (o fichero único) asignándole un nombre adecuado como ejercicio1.html.

1. Calcule las derivadas primera, segunda, tercera y cuarta de las siguientes ecuaciones, en x=0.75.

$$f(x)=\frac{\sin(x)+2\cos(x)}{2x+3}$$

$$f(x)=\sqrt{\frac{3x^{2.1}+5x-3}{x^{1.7}+3x+1}}$$

$$f(x)=\ln\left(\frac{5x+8x^2-9x^{1.2}}{4x+7}\right)$$

2. Calcule el valor de las siguientes integrales empleando: a) El método del Trapecio con 300 segmentos; b) El método de Simpson con 50 segmentos y c) El método de Romberg con un error igual a 1e-14:

$$\int_{1.1}^{3.2}\left(x^6+2x^5+3x^4+4x^3+5x^2+6x+7\right)\cdot dx$$

$$\int_{0.1}^{1.7} \left(\frac{\sin(2*x) + 2\cos(x)}{3x + 5} \right) \cdot dx$$

$$\int_{2.3}^{7.2} \left(\sqrt{\frac{x^{2.3} + 4x^{1.3} - 3}{x^{1.6} + 3x^{1.2} + 1}} \right)$$

$$\int_{0.6}^{1.4} \left(\frac{x + 4x^{2.2} - 9x^{1.4}}{4x^{1.4} + 7x} \right) \cdot dx$$

10. ECUACIONES DIFERENCIALES ORDINARIAS

Con frecuencia en la solución de problemas en el campo de la ingeniería es necesario resolver una o más ecuaciones diferenciales, tanto ordinarias como entre derivadas parciales.

En este tema se estudian algunos de los métodos que nos permiten resolver ecuaciones y sistemas de ecuaciones diferenciales ordinarias.

Una ecuación diferencial ordinaria de enésimo grado puede ser representada en la siguiente forma:

$$y^n = f\left(x, y, y', y'', y''', \ldots, y^{n-1}\right) \tag{10.1}$$

Donde *"x"* es la variable independiente, *"y"* la variable dependiente, "y'" "y''", *etc.*, son las derivadas primera: $y'=dy/dx$; segunda: $y''=d^2y/dx^2$, *etc.*

Las ecuaciones diferenciales surgen al momento de modelar matemáticamente la solución de problemas en el campo de la ingeniería y deben ser resueltas porque es necesario calcular valores de la variable dependiente "y" para valores conocidos de la variable independiente "x".

Este cálculo no es directo porque se desconoce la forma que tiene la función $y = f(x)$. Esa es justamente la función que se busca cuando se resuelve la ecuación diferencial mediante los métodos analíticos (los mismos que han sido estudiados en cálculo III).

No obstante, la mayoría de las ecuaciones diferenciales que surgen en la solución de problemas prácticos son demasiado complejas como para ser resueltas analíticamente. Esto sumado a lo moroso del proceso analítico hace que dichas ecuaciones se resuelvan casi exclusivamente mediante métodos numéricos implementados en dispositivos programables.

El encontrar la forma que tiene la función en una ecuación diferencial es complejo, porque una misma función puede dar lugar a un infinito número de ecuaciones diferenciales. Para ilustrar este hecho, se puede tomar por ejemplo la siguiente función:

$$y = f(x) = 3x^3 + 2x^2 + 5x \tag{10.2}$$

Las derivadas primera, segunda y tercera de esta función son:

$$y' = 9x^2 + 4x + 5 \tag{10.3}$$

$$y'' = 18x + 4 \tag{10.4}$$

$$y''' = 18 \tag{10.5}$$

Con estas derivadas y la función original se pueden generar, por ejemplo, las siguientes ecuaciones diferenciales:

$$\begin{gathered} 2xy' + 3y'' = 2x(9x^2 + 4x) + 3(18x + 4) = 18x^3 + 10x + 54x + 12 \\ 2xy' + 3y'' - 18x^3 - 64x - 12 = 0 \end{gathered} \tag{10.6}$$

$$\begin{gathered} 3yy''' - 2y'' = 3y(18) - 2(18x + 4) = 54y - 36x - 8 \\ 3yy''' - 2y'' - 54y + 36x + 8 = 0 \end{gathered} \tag{10.7}$$

La solución de cualquiera de estas dos ecuaciones diferenciales es la ecuación (10.2). De manera similar podemos generar un número infinito de ecuaciones diferenciales y la solución de todas ellas seguiría siendo la ecuación (10.2).

En un caso real no se cuenta con la función que da lugar a las ecuaciones diferenciales, sino solamente las ecuaciones diferenciales. Como ya se dijo es esa función la que se busca al resolver el sistema con los métodos analíticos. Con los métodos numéricos, por el contrario, no se busca la función, sino la manera de predecir con el menor error posible valores de "y" para valores conocidos de "x".

10.1. PROBLEMAS DEL VALOR INICIAL Y PROBLEMAS DEL VALOR LÍMITE

En general al resolver ecuaciones diferenciales ordinarias se presentan dos tipos de problemas:

Problemas del valor inicial: cuando se conoce el valor de la variable dependiente y de todas sus derivadas, para un valor conocido de la variable independiente.

Problemas del valor límite: Cuando algunos valores de la variable dependiente y de sus derivadas se conocen para un valor de la variable independiente y otros para otro.

Los métodos que se estudian en este tema permiten resolver problemas del valor inicial. Estos métodos combinados con otros métodos iterativos (como el de Newton – Raphson) permiten resolver también los problemas del valor límite.

De los muchos métodos disponibles para resolver los problemas del valor inicial se estudiarán los métodos de *Euler* y *Runge-Kutta*. El primero porque, al ser sencillo, permite comprender el procedimiento que se sigue al resolver numéricamente ecuaciones diferenciales y el segundo porque al proporcionara resultados precisos, es uno de los métodos más utilizados en la práctica por devolver resultados confiables.

10.2. MÉTODO DE EULER

El método de Euler, como todos los métodos numéricos sólo permite resolver ecuaciones diferenciales de primer orden pues, como se demostrará más adelante, una ecuación diferencial de enésimo orden puede ser transformada en un sistema de *"n"* ecuaciones diferenciales de primer orden.

Por esta razón se estudia en primer lugar la solución de ecuaciones diferenciales de primer orden.

10.2.1. Ecuaciones diferenciales de primer orden

Consideremos la siguiente ecuación diferencial de primer orden:

$$y' = f(x, y) \tag{10.8}$$

Para los problemas del valor inicial, se conoce el valor de la variable dependiente (y_0) para un valor inicial de la variable independiente (x_0).

El problema radica en calcular el valor de la variable dependiente (y_n) para un valor conocido de la variable independiente (x_n).

Si se conociera la forma que tiene la curva *y'* versus *f(x,y)* el valor de la variable dependiente podría ser calculado integrando la función entre x_0 y x_n, es decir calculando el área bajo la curva entre esos límites, pues como se sabe, la integral de "y'" es "y", es decir:

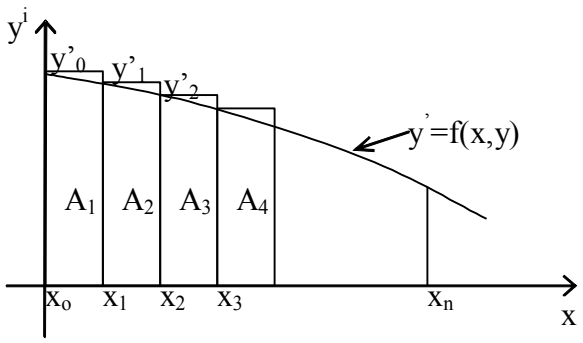
$$\int_{y_0}^{y_n} y' = y_n - y_0 = \int_{x_0}^{x_n} f(x, y)\, dx$$

$$y_n = y_0 + \int_{x_0}^{x_n} f(x, y)\,dx \qquad (10.9)$$

Como en la práctica no se conoce la forma de la curva, lo que hacen los diferentes métodos numéricos es predecir la forma de la curva y emplear la curva resultante para calcular el área bajo la curva (la integral).

La manera en que se predice la curva es lo que diferencia a un método de otro.

En el método de *Euler* se divide el área comprendida entre x_0 y x_n en un número de segmentos igualmente espaciados y se asume que el área de cada uno de esos segmentos es un rectángulo:



Como el ancho de los segmentos es conocido:

$$h = \frac{x_n - x_0}{n} \qquad (10.10)$$

El área del primer segmento (A_1) se calcula multiplicando la altura del rectángulo (y'_0) por su ancho (h). La altura (y'_0) se calcula sustituyendo los valores iniciales (x_0, y_0) en la ecuación (10.8):

$$y'_0 = f(x_{0,}, y_0) \qquad (10.11)$$

Entonces el valor de la variable dependiente en el segundo punto (y_1), se calcula con:

$$\int_{y_0}^{y_1} y' = y_1 - y_0 \approx A_1 = y'_0 h$$

$$y_1 = y_0 + y'_0 h \qquad (10.12)$$

El valor de la variable independiente en este punto (x_1) es $x_1=x_0+h$. Ahora con este nuevo punto y la ecuación ((10.8), se calcula el valor de y'_1:

$$y'_1 = f(x_1, y_1) \qquad (10.13)$$

Entonces se puede calcular el área del segundo segmento (A_2) y con este el valor de y_2:

$$\int_{y_1}^{y_2} y' = y_2 - y_1 \; aprox \; A_2 = y'_1 h$$

$$y_2 = y_1 + y'_1 h \qquad (10.14)$$

Donde el valor de x_2 se calcula con:

$$x_2 = x_1 + h$$

Con este nuevo punto se puede calcular A_3 y con A_3 el valor y_3, prosiguiendo de esta manera hasta llegar a A_n, segmento en el cual se obtiene el valor de y_n.

Por lo tanto, las ecuaciones para calcular los n puntos, hasta llegar al segmento A_n y en consecuencia al valor de y_n, son:

$$\left.\begin{array}{l} y_{j+1} = y_j + y'_j h = y_j + f(x_j, y_j) h \\ x_{j+1} = x_j + h \end{array}\right\} \quad j = 0 \rightarrow n-1 \qquad (10.15)$$

Que son las ecuaciones del método de Euler.

Sin embargo, en lugar de aplicar repetidamente las ecuaciones (10.15) para calcular valores de "y" para cada nuevo valor de "x", es más eficiente dar un límite adecuado a "x" (el valor de x_n) de manera que el mismo no sea superado en la práctica. Entonces, empleando alguno de los métodos estudiados en el tema de interpolación, generar, con los puntos intermedios (x_i,y_i) calculados en la aplicación del método de Euler, una función de interpolación. Dicha función constituye la solución numérica de la ecuación diferencial: el equivalente a la función analítica ("f(x)") que se busca cuando se resuelve la ecuación diferencial de forma analítica.

El método Euler no devuelve resultados confiables, porque se asumen áreas rectangulares para los segmentos en los que se divide el área bajo la curva y con ello se introduce un error apreciable en los cálculos. Por esta razón, el método de Euler no se emplea la solución de problemas reales, sin embargo, la lógica que se sigue en este método es esencialmente la misma que en otros métodos más elaborados.

Si en lugar de asumir áreas rectangulares para los segmentos se asumen áreas trapezoidales, la exactitud del método de Euler mejora considerablemente (con esta modificación, las ecuaciones resultantes se conocen con el nombre del *método de Euler modificado*).

10.2.1.1. Ejemplo manual

Para comprender mejor el método de Euler, se resolverá la siguiente ecuación diferencial, encontrando el valor de "y" para "x=6":

$$3y' + 4xy - 8x^3 + 4x = 0 \quad \{ y = 4 \;\; para \;\; x = 2$$

Primero se despeja la derivada primera para que se encuentre en forma de la ecuación (10.8):

$$y' = f(x, y) = \frac{8x^3 - 4xy - 4x}{3}$$

Entonces se programa esta función:

```
>>fxy=function(x,y){return (8*x*x*x-4*x*y-4*x)/3;};
```

Para este ejemplo se tomará un número de segmentos igual a 10 (n=10), el valor inicial de "x" (x_0) es 2, siendo su valor final (x_n) 6. Se sabe que para el valor inicial de "x" (x_0=2), el valor de y (y_0) es 4. Se guardan estos datos en variables y se calcula el ancho ("h") de cada segmento:

```
>>n=10; x0=2; xn=6; y0=4; h=(xn-x0)/n;
```

Si el único valor de interés es el pedido (el valor de "y" para "x=2"), no

es necesario guardar ningún punto intermedio, pero deben ser guardados si se quiere calcular uno más valores intermedios. En este ejemplo (aunque no es realmente necesario) se guardarán dichos valores en los vectores "vx" y "vy", siendo sus primeros elementos los valores iniciales conocidos:

```
>>vx=[x0]; vy=[y0];
```

Los "n" puntos, de los "n" segmentos, se calculan con las ecuaciones (10.15), empleando un ciclo "for" que vaya desde 0 hasta n-1:

```
>>for (j=0;j<n;j++){vy[j+1]=vy[j]+fxy(vx[j],vy[j])*h; vx[j+1]=vx[j]+h;};
```

El valor buscado es el correspondiente al último valor calculado, es decir el último elemento guardado en el vector "vy":

```
>>vy[n]
67.91863273250716
```

Si se quieren calcular otros valores de "y", para valores de "x" comprendidos entre 2 y 6 (los límites empleados en el anterior cálculo) es más eficiente emplear un método de interpolación con los valores guardados en los vectores "vx" y "vy":

```
>>vx
[2, 2.4, 2.8, 3.1999999999999997, 3.5999999999999996, 3.9999999999999996,
4.3999999999999995, 4.8, 5.2, 5.6000000000000005, 6.000000000000001]
>>vy
[4, 7.2, 11.4496, 16.273663999999993, 21.745810773333332,
27.840254088533325, 34.58104536632888, 41.94712557334376,
49.96728410558373, 58.59988285276487, 67.91863273250716]
```

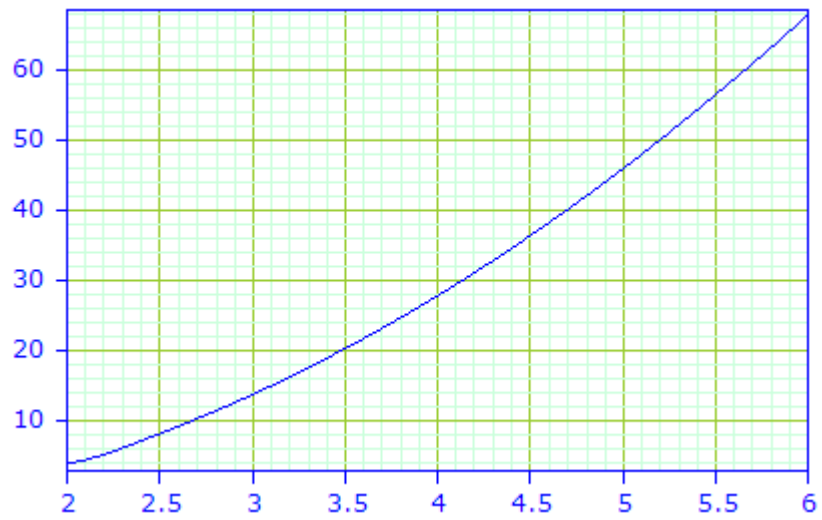
Por ejemplo, empleando el método de Newton, con estos datos se puede generar la función de interpolación respectiva:

```
>>fi=geninnew(vx,vy);
```

Esta función constituye la solución numérica de la ecuación diferencial (el equivalente a la solución analítica) y con la misma se pueden calcular valores de la variable dependiente "y" siempre y cuando los valores de la variable dependiente "x" estén comprendidos entre los límites empleados en el método de Euler (en este caso entre 2 y 6).

Así por ejemplo, se puede emplear esta función, para dibujar la curva de solución de la ecuación diferencial:

```
>>plot([fi],2,6)
```

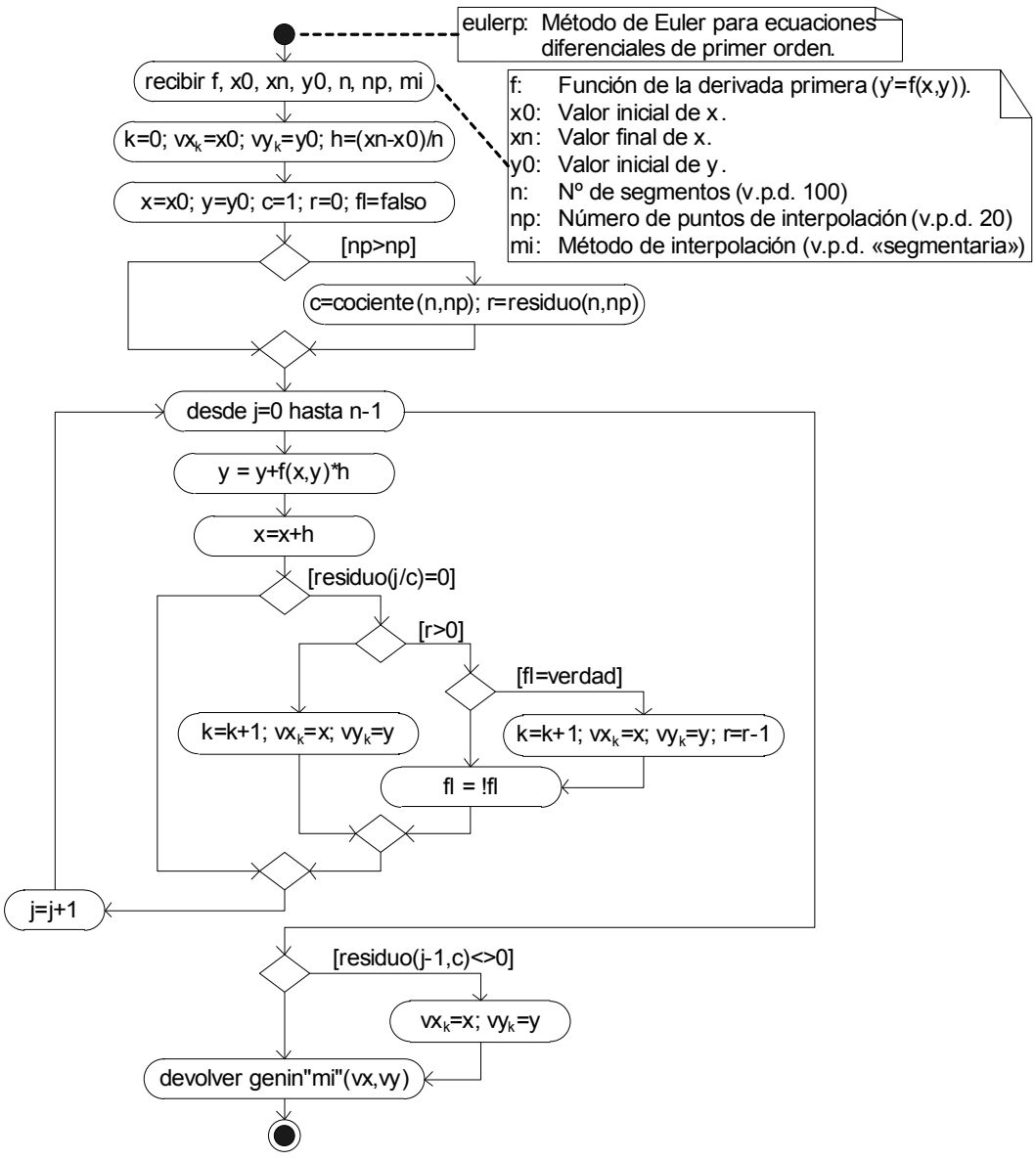


Así como calcular valores puntuales, por ejemplo calcular valores de "y" para valores de "x" iguales a 2.3, 3.7, 4.6 y 5.7:

```
>>map(fi,[2.3,3.7,4.6,5.7])
[6.209878565272479, 23.210004938771107, 38.18679323571563,
60.82012700753234]
```

10.2.1.2. Algoritmo y código

El algoritmo del método de Euler, que devuelve la función de interpolación correspondiente a la solución, es:



Como se puede ver, por defecto, la función de interpolación se genera empleando sólo 20 de los "n" puntos calculados en el proceso (21 en total pues siempre se incluyen los valores iniciales conocidos). Esto porque un número excesivo de puntos provoca errores de redondeo en las funciones de interpolación (sobre todo en los métodos polinomiales).

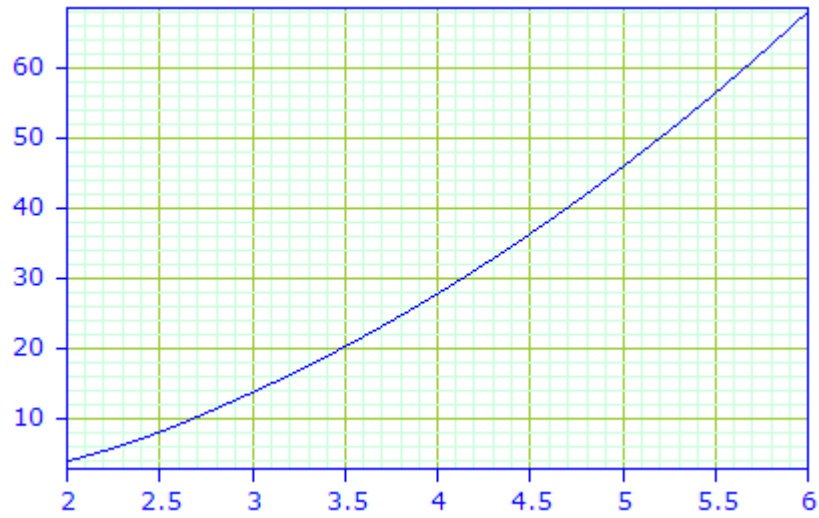
El código respectivo, añadido a la librería "hpvlib.js" es:

```
function eulerp(f,x0,xn,y0,n,np,mi){
```

```
    if (n==null) n=100;
    if (np==null) np=20;
    if (mi==null) mi="segmentaria";
    mi=mi.toLowerCase();
    var vx=[x0],vy=[y0],h=(xn-x0)/n,x=x0,y=y0,j,k=0,c=1,r=0,fl=false;
    if (n>np) {c=quot(n,np); r=n%np;};
    for (j=0;j<n;j++){
      y+=f(x,y)*h;
      x=round(x+h,15);
      if (j%c == 0)
        if (r>0) {
          if (fl) {vx[++k]=x; vy[k]=y; r--};
          fl=!fl;}
        else
          {vx[++k]=x; vy[k]=y;};
    }
    if (--j%c != 0) {vx[k]=x; vy[k]=y;}
    switch(mi) {
      case "lagrange": return geninlag(vx,vy);
      case "newton": return geninnew(vx,vy);
      case "lineal": return geninlin(vx,vy);
      default: return geninseg(vx,vy);
    }
  }
```

Haciendo correr el programa con los datos del ejemplo manual se obtiene, prácticamente los mismos resultados que en dicho ejemplo:

```
>>fxy=function(x,y){return (8*x*x*x-4*x*y-4*x)/3;};
>>fx=eulerp(fxy,2,6,4,10,null,"newton");
>>plot([fx],2,6)
```




```
>>map(fx,[2.3,3.7,4.6,5.7])
[6.20987856527241, 23.21000493877111, 38.18679323571563, 60.82012700753221]
```

Estos resultados no concuerdan exactamente con los del ejemplo manual, porque en el programa, para reducir errores, se redondean los valores de la variable independiente.

Sin embargo, 10 segmentos es un número muy pequeño para el método de Euler. Para obtener resultados más precisos es necesario incrementar el número de segmentos, así con 200 se obtiene:

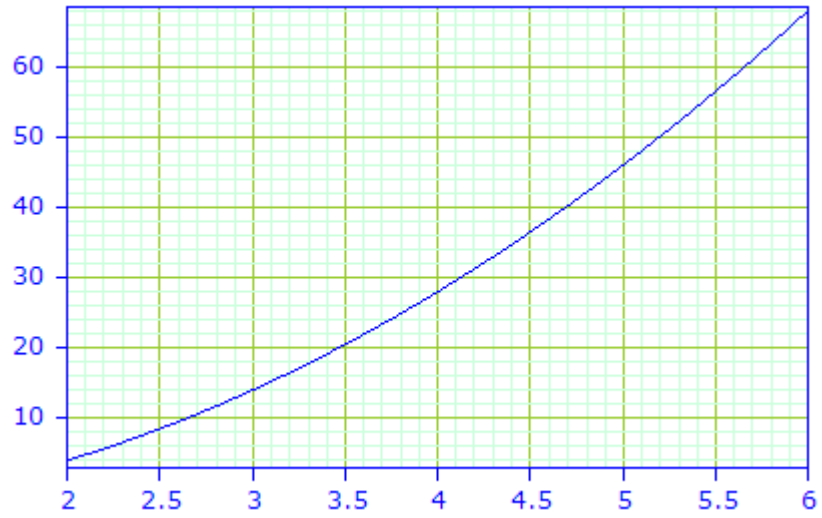
```
>>fx=eulerp(fxy,2,6,4,200,null,"newton");
```

```
>>fx(6)
67.994888960236
```

Resultado que es más cercano a la solución correcta (68) que el obtenido con 10 segmentos (67.91863273250716).

Para comprender el por qué la función de interpolación no se genera con todos los puntos calculados en el proceso (sino por defecto sólo con 21), se dibuja primero la gráfica de la función generada con los 21 puntos:

```
>>plot([fx],2,6)
```

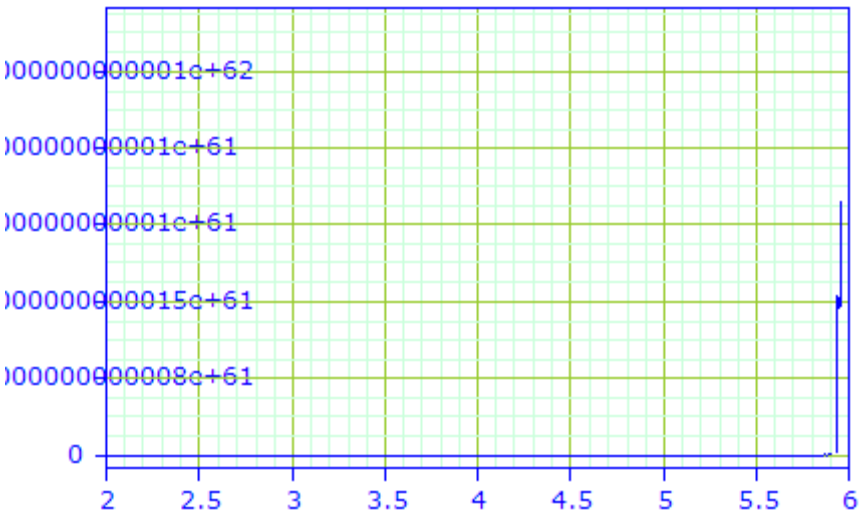


Ahora se crea otra función empleando el mismo número de segmentos y el mismo método de interpolación (de Newton), pero generando la función de interpolación con todos los puntos:

```
>>fx2=eulerp(fxy,2,6,4,200,200,"newton");
```

Y se dibuja la gráfica de la función resultante:

```
>>plot([fx2],2,6)
```



Como se puede apreciar, en este caso, la función de interpolación predice resultados erróneos. Ello se debe a errores de redondeo que se generan cuando el método de interpolación trabaja con valores muy cercanos entre sí.

Como otro ejemplo, se resolverá la siguiente ecuación diferencial y se calcularán los valores de "y" para valores de "x" iguales a 0.3, 0.7, 0.9, 1.2, 1.5, 1.7 y 2.0:

$$y'=4e^{0.8x}-0.5y \quad \{y=2 \;\; para \;\; x=0$$

Primero se programa la función correspondiente a la derivada primera:

```
>>fxy2=function(x,y){return 4*exp(0.8*x)-0.5*y;};
```

Luego se crea la función de interpolación con valores de "x" que van desde 0 (el valor inicial) hasta 2 (el máximo valor a calcular), empleando en este caso el número de divisiones por defecto (n=100), el número de puntos por defecto (20) y el método de interpolación por defecto ("segmentaria"):

```
>>fx2=eulerp(fxy2,0,2,2);
```

Finalmente se calculan los valores de "y" (redondeados al cuarto dígito después del punto) empleando la función generada:

```
>>round(map(fx2,[0.3,0.7,0.9,1.2,1.5,1.7,2]),4
[2.9791, 4.6133, 5.6147, 7.4155, 9.6658, 11.4777, 14.7764]
```

10.2.2. Ecuaciones diferenciales de enésimo orden

Como se mencionó previamente, una ecuación de enésimo orden puede ser transformada en un sistema de ecuaciones diferenciales de primer orden. Para ilustrar el procedimiento se trabajará con la siguiente ecuación diferencial:

$$\begin{aligned} &3x\frac{d^4y}{dx^4}+(x-y)\frac{d^3y}{dx^3}+5\frac{d^2y}{dx^2}-\frac{dy}{dx}-\frac{x^2-y^2}{xy} \\ &=3xy''''+(x-y)y'''+5y''-y'-\frac{x^2-y^2}{xy}=0 \end{aligned} \qquad (10.16)$$

Esta ecuación de cuarto orden puede ser transformada en un sistema de 4 ecuaciones diferenciales de primer orden realizando los siguientes cambios de variable:

$$\begin{aligned} &\frac{dy}{dx}=y'=y_1 \\ &\frac{d^2y}{dx^2}=\frac{d}{dx}\left(\frac{dy}{dx}\right)=\frac{d}{dx}(y_1)=y'_1=y_2 \\ &\frac{d^3y}{dx^3}=\frac{d}{dx}\left(\frac{d^2y}{dx^2}\right)=\frac{d}{dx}(y_2)=y'_2=y_3 \\ &\frac{d^4y}{dx^4}=\frac{d}{dx}\left(\frac{d^3y}{dx^3}\right)=\frac{d}{dx}(y_3)=y'_3=y_4 \end{aligned} \qquad (10.17)$$

Donde y_1, y_2, y_3, etc., son nombres de variables (no valores). Con estas variables, la ecuación (10.16) puede ser reescrita de la siguiente forma:

$$\begin{aligned} &y'_3=f(x,y_0,y_1,y_2,y_3)=\frac{(y_0-x)y_3-5y_2+y_1+\dfrac{(x^2-y_0^2)}{xy_0}}{3x}=y_4 \\ &y'_0=y_1 \end{aligned} \qquad (10.18)$$

$$y'_1 = y_2$$
$$y'_2 = y_3$$

Donde se ha denominado y_0 a la variable dependiente "y". Como se están resolviendo los problemas del valor inicial, los valores de y_0, y_1, y_2 y y_3 deben ser conocidos para un determinado valor de "x".

Si se aplican las ecuaciones de Euler (10.15) a cada una de las ecuaciones de este sistema (en un segmento "j" cualquiera), empleando para todas ellas el mismo ancho de segmento "h", se obtiene:

$$y_{4,j} = f(x_j, y_j, y_{1,j}, y_{2,j}, y_{3,j}) = \frac{(y_j - x_j)\, y_{3,j} - 5\, y_{2,j} + y_{1,j} + \dfrac{x_j^2 - y_j^2}{x_j\, y_j}}{3\, x_j} \qquad (10.19)$$

$$y_{0,j+1} = y_{0,j} + y'_{0,j}\, h = y_{0,j} + y_{1,j}\, h$$
$$y_{1,j+1} = y_{1,j} + y'_{1,j}\, h = y_{1,j} + y_{2,j}\, h$$
$$y_{2,j+1} = y_{2,j} + y'_{2,j}\, h = y_{2,j} + y_{3,j}\, h$$
$$y_{3,j+1} = y_{3,j} + y'_{3,j}\, h = y_{3,j} + y_{4,j}\, h$$
$$x_{j+1} = x_j + h$$

Como se puede observar, la única función que realmente necesita ser evaluada es la correspondiente a la derivada de enésimo orden (y_4 en el ejemplo), pues las otras funciones son simplemente los valores de variable dependiente y de las derivadas del punto anterior.

Generalizando las ecuaciones de Euler para una ecuación de orden "m", se tiene:

$$\left.\begin{aligned} y_{m,j} &= f(x_j, y_{0,j}, y_{1,j}, y_{2,j}, \ldots, y_{m-1,j}) \\ y_{i,j+1} &= y_{i,j} + y'_{i,j}\, h = y_{i,j} + y_{i+1,j}\, h \quad \{i = 0 \rightarrow m-1 \\ x_{j+1} &= x_j + h \end{aligned}\right\} \; j = 0 \rightarrow n-1 \qquad (10.20)$$

Donde, como ya se explicó y_0, y_1, y_2, y_3, etc., son los nombres que se dan a la variable dependiente (y_0) y a sus derivadas:

$$\frac{d^i y}{dx^i} = y'_{i-1} = y_i \quad \left\{ i = 1 \rightarrow n \right. \qquad (10.21)$$

10.2.2.1. Ejemplo Manual

Para comprender mejor el procedimiento que se sigue en el método de Euler, se resolverá manualmente la siguiente ecuación diferencial, encontrando el valor de la variable dependiente "y" para "x" igual a 4.

$$3\, y'''' - 3\, x\, y'' + y' + 1.8711\, x^{-1.9} + 14.49\, x^{1.1} - 5 = 0$$
$$y = 4,\; y' = 11.3,\; y'' = 6.93,\; y''' = 0.693 \;\; para \;\; x = 1$$

Se sabe que la solución analítica de esta ecuación diferencial es:

$$y = 3x^{2.1} + 5x - 4$$

Para efectos de comparación se programa esta función:

```
>>f4c=function(x){return 3*pow(x,2.1)+5*x-4;};
```

Ahora se despeja el término de mayor orden:

$$y'''' = \frac{3\,x\,y'' - y' - 1.8711\,x^{-1.9} - 14.49\,x^{1.1} + 5}{3}$$

$$y'_3 = y_4 = \frac{3\,x\,y_2 - y_1 - 1.8711\,x^{-1.9} - 14.49\,x^{1.1} + 5}{3}$$

Y se programa esta función resultante:

```
>>y4=function(x,y){return (3*x*y[2]-y[1]-1.8711*pow(x,-1.9)-
14.49*pow(x,1.1)+5)/3;};
```

Observe que en este caso sólo entran en los cálculos los valores de y_1 y y_2, pero aún así, la función recibe un vector (y) con el valor de la variable dependiente (y_0) y de sus derivadas (y_1, y_2, y_3).

Para comenzar a aplicar el método se fija el número de segmentos (para este ejemplo 10), se guardan los valores iniciales y se calcula el ancho de los segmentos ("h"):

```
>>n=10; x0=1; xn=4; y0=[4,11.3,6.93,0.693]; m=y0.length; h=(xn-x0)/n;
```

Ahora se pueden aplicar la ecuaciones de Euler (ecuaciones 10.20), para calcular los 10 puntos de los 10 segmentos (n=10) considerados en este ejemplo. Para ello, se inicializan las variables que forman parte del proceso:

```
>>vx=[x0]; vy=[y0[0]]; k=0; x=x0; y=copy(y0);
```

Donde "vx" y "vy" son los vectores en los que se guardan los puntos calculados (para generar luego la función de interpolación). El cálculo propiamente se lleva a cabo dentro de un ciclo "for" que va desde "j=0" hasta "n-1" (para los 10 puntos) y otro ciclo anidado "for" que va desde "i=0" hasta "m-1" (para calcular las "m" derivadas).

```
>>for(j=0;j<n;j++){y[4]=y4(x,y); for(i=0;i<m;i++) y[i]=y[i]+y[i+1]*h; x+=h;
vx[++k]=x; vy[k]=y[0];};
```

El valor buscado (el valor de "y" para "x=4") corresponde al último elemento almacenado en el vector "vy" (es decir el último valor calculado para y[0]):

```
>>vy[n]
67.54874149514693
```

El valor exacto, obtenido con la solución conocida y programada en la función "f4c", es:

```
>>f4c(4)
71.13752103985769
```

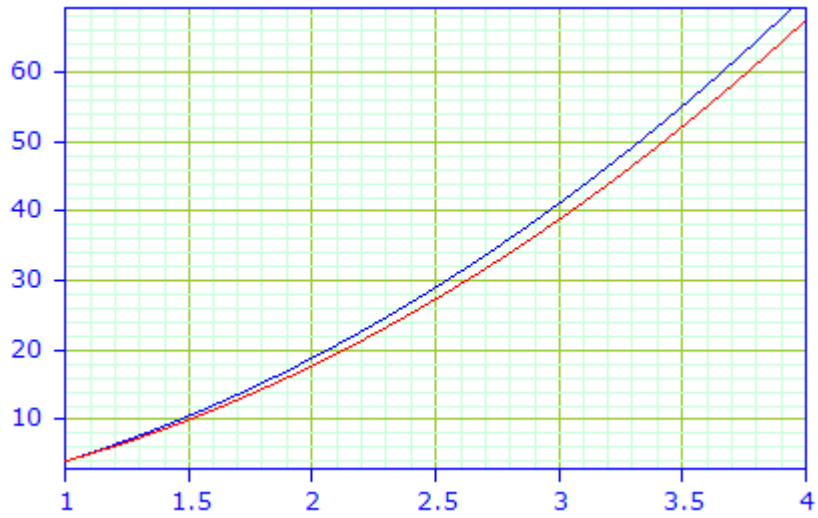
En consecuencia, en este caso, el resultado obtenido con el método de Euler tiene un error del 5%.

Para apreciar mejor el error que presenta el método de Euler, es conveniente crear la función de interpolación, correspondiente a la solución numérica, empleando los valores guardados en los vectores "vx" y "vy":

```
>>f4i=geninnew(vx,vy);
```

Y graficar luego tanto la solución conocida ("f4c") como la función de interpolación ("f4i"):

```
>>plot([f4c,f4i],1,4)
```



Como se puede ver la solución numérica se va separando (hacia abajo) de la solución analítica, a medida que incrementa el valor de "x". Este resultado no es de extrañar si se toma en cuenta que el método de Euler es un método poco preciso. Para mejorar el resultado se puede incrementar el número de divisiones, por ejemplo, si en lugar de 10, se toman 100 divisiones, se obtiene:

```
>>n=100; h=(xn-x0)/n; vx=[x0]; vy=[y0[0]]; k=0; x=x0; y=copy(y0);
for(j=0;j<n;j++){y[4]=y4(x,y); for(i=0;i<m;i++) y[i]=y[i]+y[i+1]*h; x+=h;
vx[++k]=x; vy[k]=y[0];};
>>vy[n]
70.77819975337361
```

Que ahora tiene un error de sólo el 0.5%.

Creando una función de interpolación (no se emplea un método polinomial por las razones ya estudiadas):

```
>>f4i=geninlin(vx,vy);
```

Se puede apreciar que ahora, la solución numérica es muy cercana a la analítica:

```
>>plot([f4c,f4i],1,4)
```

Con la función de interpolación, se pueden calcular otros valores de "y" para valores de "x" comprendidos entre 1 y 4, por ejemplo para valores de "x" igual a 1.3, 1.5, 1.8, 2, 2.4, 2.9, 3.3, 3.7 y 3.8, se obtiene los siguientes resultados (redondeados al segundo dígito después del puento):

```
>>round(map(f4i,[1.3,1.5,1.8,2,2.4,2.9,3.3,3.7,3.8]),2)
[7.39, 10, 14.44, 17.76, 25.28, 36.38, 46.62, 58.11, 61.18]
```

Que son cercanos (pero no iguales) a los resultados correctos:

```
>>round(map(f4c,[1.3,1.5,1.8,2,2.4,2.9,3.3,3.7,3.8]),2)
[7.7, 10.53, 15.31, 18.86, 26.86, 38.56, 49.31, 61.31, 64.51]
```

Como un segundo ejemplo se resolverá la siguiente ecuación diferencial, encontrando los valores de "y" para valores de "t" iguales a 0.2, 0.3, 0.4, 0.8, 1.1, 1.4 y 1.5:

$$y'''+t\,y''-t\,y'-2\,y=t \quad \{y=0;\ y'=0;\ y''=0 \ \ para\ \ t=0$$

Al igual que en el ejemplo anterior primero se despeja la derivada de mayor orden:

$$y'''=t+2\,y+t\,y'-t\,y''$$

$$y'_2=y_3=t+2\,y_0+t\,y_1-t\,y_2$$

Y se programa la función resultante:

```
>>y3=function(t,y){return t+2*y[0]+t*y[1]-t*y[2];};
```

Puesto que los valores a calcular son menores a 1.5, ese será el valor final de "t" (tn). Los valores iniciales vienen dados en la definición de la ecuación. Asumiendo 25 segmentos (n=25), los valores iniciales y el incremento "h" para este ejemplo son:

```
>>t0=0; tn=1.5; y0=[0,0,0]; m=y0.length; n=25; h=(tn-t0)/n;
```

Ahora, al igual que en el anterior ejemplo, se aplican las ecuaciones de Euler (10.20) y se guardan los puntos calculados en los vectores *"vt"* y *"vy"*:

```
>>vt=[t0]; vy=[y0[0]]; k=0; t=t0; y=copy(y0); for(j=0;j<n;j++)
{y[m]=y3(t,y); for(i=0;i<m;i++) y[i]+=y[i+1]*h; t+=h; vt[++k]=t;
vy[k]=y[0];};
```

Con los vectores "vt" y "vy" se crea la función de interpolación (en este caso con el método de interpolación segmentaria):

```
>>f3i=geninseg(vt,vy);
```

Y con esta función se calculan los valores de "y" requeridos (redondeados al quinto dígito después del punto):

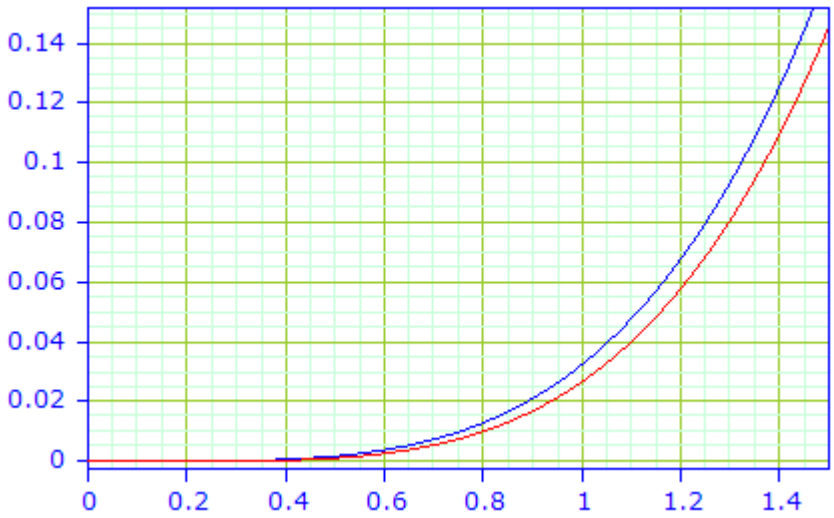
```
>>round(map(f3i,[0.2,0.3,0.4,0.8,1.1,1.4,1.5]),5)
[0, 0.00006, 0.00035, 0.01003, 0.04012, 0.10998, 0.14606]
```

Como en este caso no se cuenta con la solución analítica, para determinar si los resultados obtenidos son o no confiables, se repite el cálculo pero empleando un número de segmentos igual al doble del anterior:

```
>>n=50; h=(tn-t0)/n; vt=[t0]; vy=[y0[0]]; k=0; t=t0; y=copy(y0);
for(j=0;j<n;j++){y[m]=y3(t,y); for(i=0;i<m;i++) y[i]+=y[i+1]*h; t+=h; vt[+
+k]=t; vy[k]=y[0];};
>>f3ib=geninseg(vt,vy);
```

Y se grafican las dos funciones de interpolación:

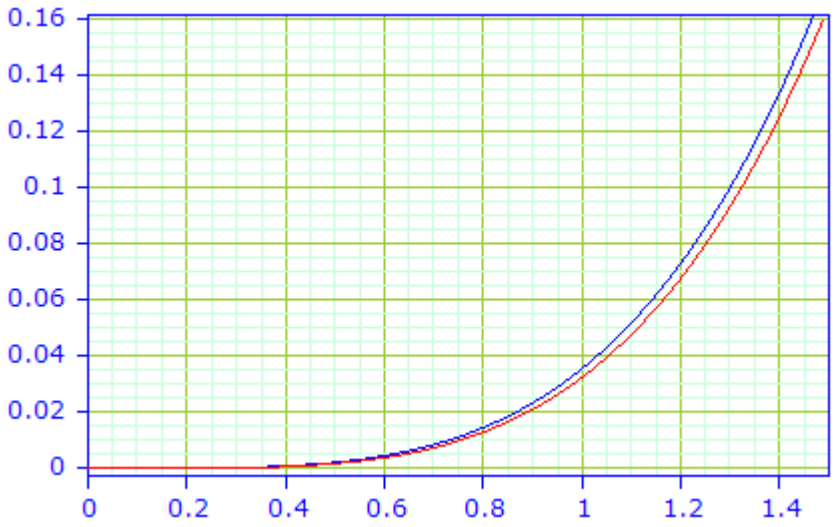
```
>>plot([f3ib,f3i],t0,tn)
```



Como se puede ver existe una diferencia apreciable entre las dos curvas, por lo que los valores calculados no son confiables.

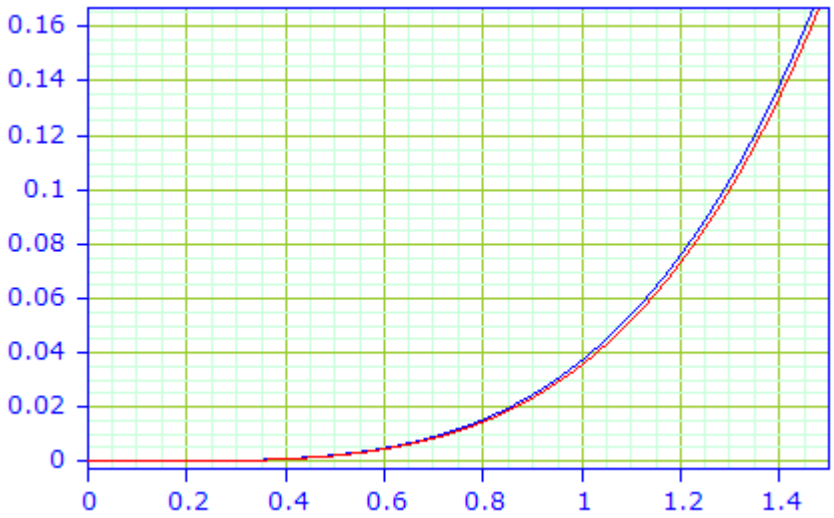
Para determinar si la nueva función de interpolación es o no confiable, se repite el proceso con el doble de segmentos:

```
>>n=100; h=(tn-t0)/n; vt=[t0]; vy=[y0[0]]; k=0; t=t0; y=copy(y0);
for(j=0;j<n;j++){y[m]=y3(t,y); for(i=0;i<m;i++) y[i]+=y[i+1]*h; t+=h; vt[+
+k]=t; vy[k]=y[0];};
>>f3ic=geninseg(vt,vy);
>>plot([f3ic,f3ib],t0,tn)
```



Una vez más se concluye que la función de interpolación no es confiable: las dos curvas presentan aún una diferencia apreciable entre sí. Por lo que se vuelve a repetir el proceso, empleando un número de segmentos igual al doble del anterior:

```
>>n=200; h=(tn-t0)/n; vt=[t0]; vy=[y0[0]]; k=0; t=t0; y=copy(y0);
for(j=0;j<n;j++){y[m]=y3(t,y); for(i=0;i<m;i++) y[i]+=y[i+1]*h; t+=h; vt[+
+k]=t; vy[k]=y[0];};
>>f3id=geninseg(vt,vy);
>>plot([f3id,f3ic],t0,tn)
```



Siendo aún la diferencia es apreciable, por lo que se repite el proceso empleando el doble de segmentos del anterior (n=400):

```
>>n=400; h=(tn-t0)/n; vt=[t0]; vy=[y0[0]]; k=0; t=t0; y=copy(y0);
for(j=0;j<n;j++){y[m]=y3(t,y); for(i=0;i<m;i++) y[i]+=y[i+1]*h; t+=h; vt[+
+k]=t; vy[k]=y[0];};
>>f3ie=geninseg(vt,vy);
>>plot([f3ie,f3id],t0,tn)
```

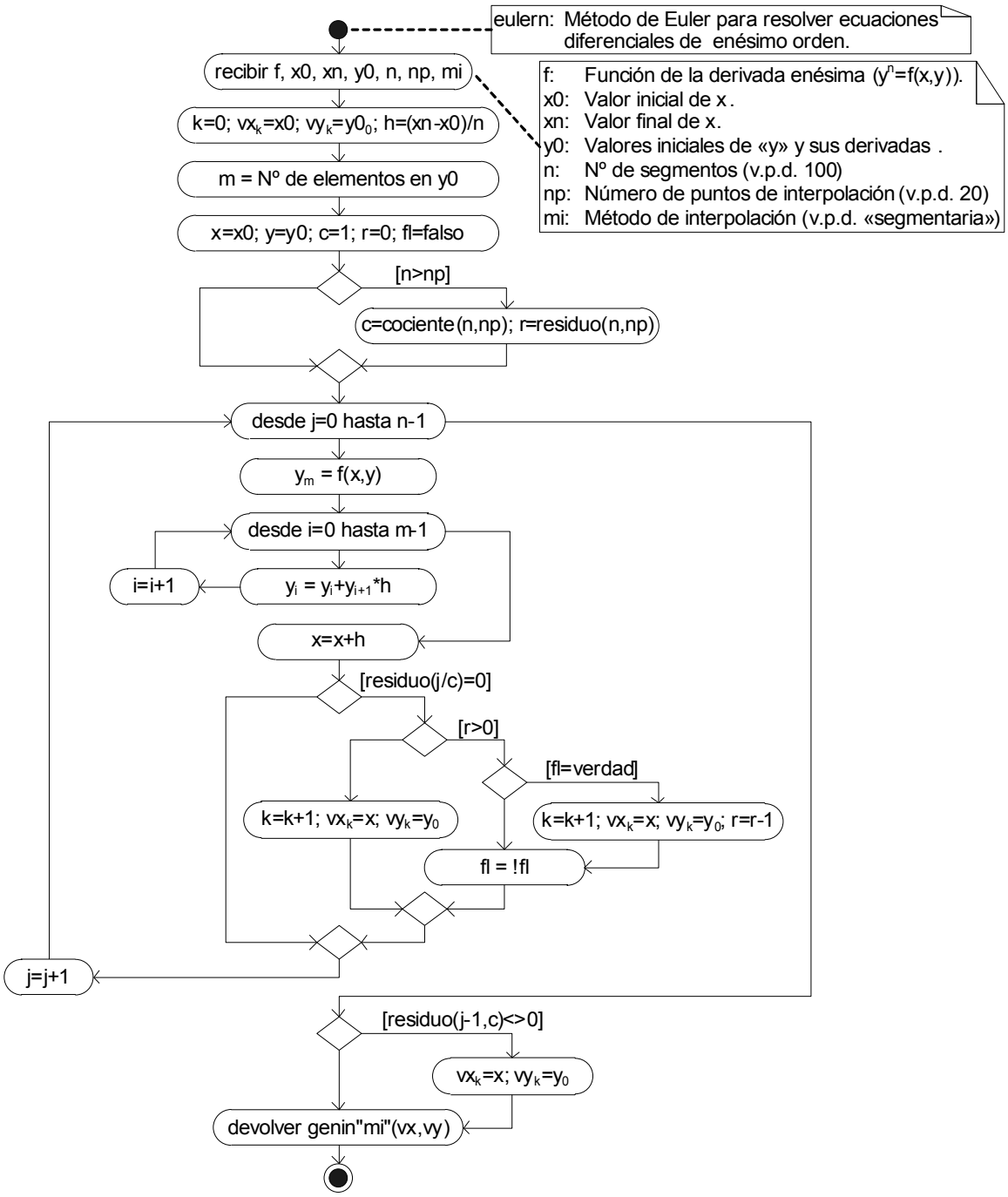
Ahora la diferencia es despreciable, por lo que se puede asumir que la última función de interpolación predice los resultados de forma confiable, por lo tanto los valores pedidos, calculados con esta función y redondeados al quinto dígito, son:

```
>>round(map(f3ie,[0.2,0.3,0.4,0.8,1.1,1.4,1.5]),5)
[0.00006, 0.00031, 0.00099, 0.01582, 0.05524, 0.14118, 0.18453]
```

Resultados, que como se puede ver, difieren considerablemente de los calculados con la primera función de interpolación.

10.2.2.2. Algoritmo y código

El algoritmo para resolver una ecuación diferencial ordinaria de enésimo orden, que básicamente sigue el proceso descrito en los ejemplos manuales, se presenta en la siguiente página.



Al igual que en las ecuaciones de primer orden, en dicho algoritmo no se emplean todos los puntos calculados, sino por defecto sólo 21. El código respectivo, añadido a la librería "hpvlib.js" es:

```
function eulern(f,x0,xn,y0,n,np,mi){
  if (n==null) n=100;
  if (np==null) np=20;
  if (mi==null) mi="segmentaria";
  mi.toLowerCase();
  var k=0,vx=[x0],vy=[y0[0]],h=(xn-x0)/n,m=y0.length;
  var x=x0,y=copy(y0),j,c=1,r=0,fl=false;
  if (n>np) {c=quot(n,np); r=np;};
  for (j=0;j<n;j++){
```

```
      y[m]=f(x,y);
      for (i=0;i<m;i++) y[i]+=y[i+1]*h;
      x=round(x+h,15);
      if (j%c == 0)
        if (r>0) {
          if (fl) {vx[++k]=x; vy[k]=y[0]; r--};
          fl=!fl;}
        else
          {vx[++k]=x; vy[k]=y[0];};
    }
    if (--j%c != 0) {vx[k]=x; vy[k]=y[0];}
    switch(mi) {
      case "lagrange": return geninlag(vx,vy);
      case "newton": return geninnew(vx,vy);
      case "lineal": return geninlin(vx,vy);
      default: return geninseg(vx,vy);
    }
  }
```

Haciendo correr el programa con los datos del último ejemplo manual, se obtienen prácticamente los mismos resultados que en dicho ejemplo.

```
>>y3=function(t,y){return t+2*y[0]+t*y[1]-t*y[2];};
>>f3r=eulern(y3,0,1.5,[0,0,0],400);
>>round(map(f3r,[0.2,0.3,0.4,0.8,1.1,1.4,1.5]),5)
[0.00006, 0.00031, 0.00099, 0.01583, 0.0553, 0.14318, 0.18453]
```

No son exactamente los mismos resultados que en el ejemplo, porque por una parte el método de interpolación es diferente y por otra sólo se emplean 20 puntos para generar la función de interpolación.

10.2.3. Sistemas de ecuaciones diferenciales

Cuando en lugar de una ecuación diferencial se debe resolver un sistema de ecuaciones diferenciales y en dicho sistema existen ecuaciones de segundo o mayor orden, las mismas deben ser transformadas previamente en ecuaciones diferenciales de primer orden (siguiendo el procedimiento descrito en el anterior acápite). De esa manera es posible transformar cualquier sistema de ecuaciones diferenciales, en un sistema de "m" ecuaciones diferenciales de primer orden:

$$\begin{aligned} y'_0 &= f_0(t, y_0, y_1, y_2, \dots, y_{m-1}) \\ y'_1 &= f_1(t, y_0, y_1, y_2, \dots, y_{m-1}) \\ &\dots \\ y'_{m-1} &= f_{m-1}(t, y_0, y_1, y_2, \dots, y_{m-1}) \end{aligned} \qquad (10.22)$$

En este sistema "*t*" es la variable independiente y y_0, y_1, y_2,..., y_{m-1} son las variables dependientes. Aplicando las ecuaciones de Euler (10.15) a cada una de las ecuaciones de este sistema, en un punto "*j*" cualquiera, se tiene:

$$\begin{aligned} y_{0,j+1} &= y_{0,j} + y'_{0,j} h = y_{0,j} + f_0(t, y_{0,} y_{1,\dots}, y_m - 1)h \\ y_{1,j+1} &= y_{1,j} + y'_{1,j} h = y_{1,j} + f_1(t, y_{0,} y_{1,\dots}, y_m - 1)h \\ &\dots \\ y_{m-1,j+1} &= y_{m-1,j} + y'_{m-1,j} h = y_{m-1,j} + f_1(t, y_{0,} y_{1,\dots}, y_m - 1)h \\ t_{j+1} &= t_j + h \end{aligned} \qquad (10.23)$$

Como se supondrá, estas ecuaciones pueden ser empleadas también para resolver una ecuación diferencial de enésimo orden, transformándola en un sistema de ecuaciones diferenciales de primer orden, sin embargo, si lo que se quiere es resolver una sola ecuación de enésimo orden es más sencillo emplear la función desarrollada en el tema anterior.

Cuando se resuelven sistemas de ecuaciones diferenciales se generan "m" vectores (uno por cada variable) con los cuales se pueden crear "m" funciones de interpolación, cada una de las cuales corresponde a las "m" soluciones del sistema.

10.2.3.1. Ejemplo manual

Para comprender mejor el proceso que se sigue al resolver sistemas de ecuaciones diferenciales, se resolverá el siguiente sistema de ecuaciones diferenciales, calculando los valores de las variables dependientes ("u", "v" y "w") para valores de "t" iguales a: 0.05, 0.1, 0.14, 0.20, 0.23, 0.26, 0.30, 0.33 y 0.35.

$$\left.\begin{aligned} u'-v+2w+t^2-2t^{1.5}-1.2t^{0.2}+10=0 \\ v'+5w-5t^{1.5}-2t+35=0 \\ w'-2u+v-t^2+2t^{1.2}-1.5t^{0.5}+10=0 \end{aligned}\right\} \; u=3,\; v=-4,\; w=-7 \;\; para \;\; t=0$$

En este caso se sabe que las soluciones analíticas de este sistema son:

$$\begin{aligned} u&=t^{1.2}+3 \\ v&=t^2-4 \\ w&=t^{1.5}-7 \end{aligned}$$

Primero, para estar de acuerdo con la simbología empleada previamente, se escriben las ecuaciones en función de las variables y_0, y_1 y y_2 (en lugar de "u", "v" y "w"):

$$\left.\begin{aligned} y_0'-y_1+2y_2+t^2-2t^{1.5}-1.2t^{0.2}+10=0 \\ y_1'+5y_2-5t^{1.5}-2t+35=0 \\ y_2'-2y_0+y_1-t^2+2t^{1.2}-1.5t^{0.5}+10=0 \end{aligned}\right\} \; y_0=3,\; y_1=-4,\; y_2=-7 \;\; para \;\; t=0$$

Que colocadas en la forma $y_i=f(t,y)$ son:

$$\begin{aligned} y_0'&=f_0(t,y)=y_1-2y_2-t^2+2t^{1.5}+1.2t^{0.2}-10 \\ y_1'&=f_1(t,y)=-5y_2+5t^{1.5}+2t-35 \\ y_2'&=f_2(t,y)=2y_0-y_1+t^2-2t^{1.2}+1.5t^{0.5}-10=0 \end{aligned}$$

Siendo, con esta nomenclatura, las soluciones conocidas:

$$\begin{aligned} y_0&=t^{1.2}+3 \\ y_1&=t^2-4 \\ y_2&=t^{1.5}-7 \end{aligned}$$

Para efectos de comparación, se programa primero estas soluciones:

```
>>fc=[];
>>fc[0]=function(t){return pow(t,1.2)+3;};
```

```
>>fc[1]=function(t){return sqr(t)-4;};
>>fc[2]=function(t){return pow(t,1.5)-7;};
```

Se programan las funciones correspondientes a las derivadas primeras del sistema de ecuaciones diferenciales:

```
>>f=[];
>>f[0]=function(t,y){return y[1]-2*y[2]-sqr(t)+2*pow(t,1.5)+1.2*pow(t,0.2)-
10;};
>>f[1]=function(t,y){return -5*y[2]+5*pow(t,1.5)+2*t-35;};
>>f[2]=function(t,y){return 2*y[0]-y[1]+sqr(t)-2*pow(t,1.2)+1.5*sqrt(t)-
10;};
```

Y para iniciar el proceso, se guardan los valores iniciales y se calcula el valor de "h", fijando en este caso el número de segmentos en 40:

```
>>t0=0; tn=0.35; y0=[3,-4,-7]; m=y0.length; n=40; h=(tn-t0)/n;
```

Ahora se deben generar los puntos para cada una de las ecuaciones del sistema aplicando las ecuaciones de Euler (ecuaciones 10.23) a cada uno de los segmentos. Estos valores se guardarán en el vector "vt" (para los valores de "t") y la matriz "vy" (para los valores del vector "y").

Como de costumbre, el primer punto lo constituyen los valores iniciales conocidos, es decir:

```
>>k=0; vt=[t0]; vy=[y0];
```

Donde "k" es el contador. Los valores iniciales de las variables "t", "y" y la variable auxiliar "y_" son:

```
>>t=t0; y=copy(y0); y_=copy(y0);
```

Recuerde que para crear una copia de un vector o matriz se debe emplear la función "copy". Ahora se generan los 40 puntos (para los 40 segmentos) empleando un ciclo "for" que va desde j=0 hasta "n-1" y, dentro del mismo, se calculan los valores de "y" (las variables dependientes) empleando otro ciclo "for" que va desde i=0 hasta "m-1":

```
>>for(j=0;j<n;j++){for(i=0;i<m;i++) y[i]+=f[i](t,y_)*h; t=round(t+h,15);
vt[++k]=t; vy[k]=y_=copy(y);};
```

Donde los valores de "t" se redondean al décimo quinto dígito para eliminar los errores de redondeo.

Los puntos calculados con la anterior instrucción son:

```
>>vt
[0, 0.00875, 0.0175, 0.02625, 0.035, 0.04375, 0.0525, 0.06125, 0.07,
0.07875, 0.0875, 0.09625, 0.105, 0.11375, 0.1225, 0.13125, 0.14, 0.14875,
0.1575, 0.16625, 0.175, 0.18375, 0.1925, 0.20125, 0.21, 0.21875, 0.2275,
0.23625, 0.245, 0.25375, 0.2625, 0.27125, 0.28, 0.28875, 0.2975, 0.30625,
0.315, 0.32375, 0.3325, 0.34125, 0.35]
>>round(vy,4)
[[3, -4, -7], [3, -4, -7], [3.0041, -3.9998, -6.9988], [3.0088, -3.9995,
-6.9972], [3.0139, -3.9989, -6.9951], [3.0193, -3.9982, -6.9927], [3.0249,
-3.9974, -6.99], [3.0308, -3.9964, -6.9871], [3.0368, -3.9952, -6.9839],
[3.043, -3.9939, -6.9805], [3.0494, -3.9924, -6.9769], [3.0559, -3.9907,
-6.9731], [3.0625, -3.9889, -6.9691], [3.0693, -3.987, -6.9649], [3.0761,
-3.9848, -6.9606], [3.0831, -3.9825, -6.9561], [3.0901, -3.9801, -6.9514],
[3.0973, -3.9775, -6.9466], [3.1045, -3.9747, -6.9416], [3.1119, -3.9717,
-6.9365], [3.1193, -3.9686, -6.9312], [3.1268, -3.9654, -6.9258], [3.1343,
-3.962, -6.9202], [3.142, -3.9584, -6.9146], [3.1497, -3.9547, -6.9087],
[3.1575, -3.9508, -6.9028], [3.1653, -3.9467, -6.8968], [3.1733, -3.9425,
-6.8906], [3.1812, -3.9381, -6.8843], [3.1893, -3.9336, -6.8779], [3.1974,
```

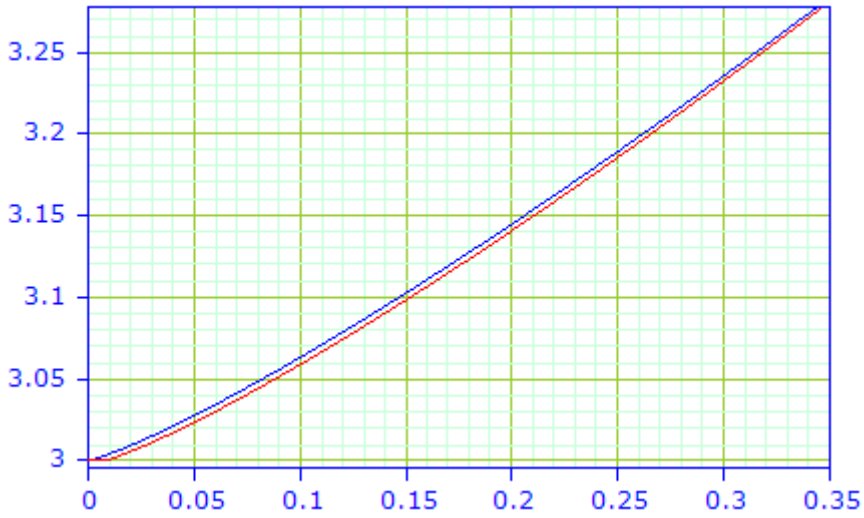
```
-3.9289, -6.8713], [3.2055, -3.9241, -6.8647], [3.2137, -3.9191, -6.8579],
[3.222, -3.9139, -6.8511], [3.2303, -3.9086, -6.8441], [3.2387, -3.9031,
-6.837], [3.2471, -3.8974, -6.8298], [3.2556, -3.8916, -6.8225], [3.2641,
-3.8857, -6.8152], [3.2727, -3.8796, -6.8077], [3.2813, -3.8733, -6.8001]]
```

Con estos valores y uno de los métodos de interpolación, en este caso el de Lagrange, se crean las tres funciones de interpolación. Con ese fin primero se transpone la matriz "vy" (para que los valores de las variables dependientes se encuentre en las filas y no en las columnas):

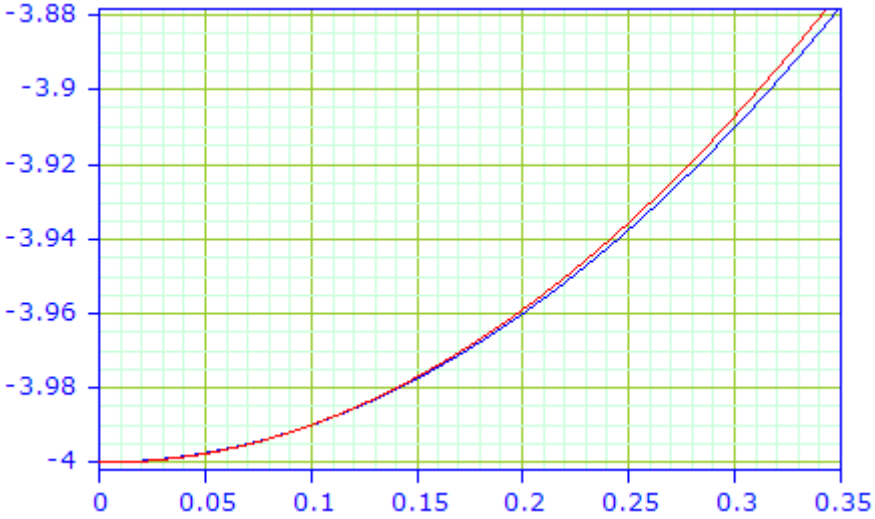
```
>>vy=transpose(vy);
>>fi=[];
>>fi[0]=geninlag(vt,vy[0]);
>>fi[1]=geninlag(vt,vy[1]);
>>fi[2]=geninlag(vt,vy[2]);
```

Para determinar visualmente cuan confiables son las soluciones encontradas se las dibuja conjuntamente las soluciones conocidas (para mayor claridad se dibujan las tres funciones por separado):

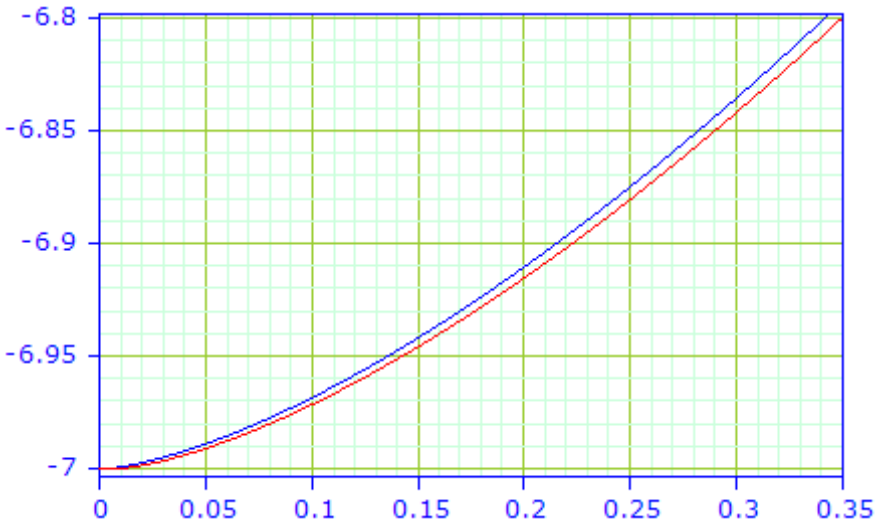
```
>>plot([fc[0],fi[0]],0,0.35)
```




```
>>plot([fc[1],fi[1]],0,0.35)
```




```
>>plot([fc[0],fi[0]],0,0.35)
```



Como era de esperar, al haberse empleado un número pequeño de divisiones, las funciones interpoladas difieren, aunque no demasiado, de las soluciones correctas.

Los valores requeridos para la primera variable dependiente ($u=y_0$), redondeados al cuarto dígito, son:

```
>>round(map(fi[0],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),4)
[3.0233, 3.0587, 3.0901, 3.1409, 3.1676, 3.1951, 3.2327, 3.2617, 3.2813]
```

Para la segunda variable independiente ($v=y_1$), se obtiene:

```
>>round(map(fi[1],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),4)
[-3.9976, -3.99, -3.9801, -3.9589, -3.9455, -3.9303, -3.907, -3.8874,
-3.8733]
```

Y para la tercera ($w=y_2$), se tiene:

```
>>round(map(fi[2],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),4)
[-6.9908, -6.9714, -6.9514, -6.9154, -6.895, -6.8732, -6.8421, -6.8173,
-6.8001]
```

Como segundo ejemplo se resolverá el siguiente sistema de 2 ecuaciones diferenciales y con la solución se calcularán valores de las variables dependientes "x" y "y" para valores de "t" iguales a 0.2, 0.5, 0.7, 0.9, 1.2, 1.3.

$$\left.\begin{aligned}\frac{dx}{dt}&=x'=2x+3y\\ \frac{dy}{dt}&=y'=2x+y\end{aligned}\right\}\ x=2.7;\ \ y=2\ \ para\ \ t=0$$

Primero se reescribe el sistema en función de la variable "t" y el vector de variables "y":

$$y'_0=f_0(t,y)=2y_0+3y_1$$
$$y'_1=f_1(t,y)=2y_0+y_1$$

Luego, la solución es prácticamente la misma que en el anterior ejemplo, excepto que ahora se tienen solo dos ecuaciones. Se guardan los valores iniciales y se calcula el incremento "h", en este caso para 200 segmentos (n=200):

```
>>t0=0; tn=1.3; y0=[2.7,2]; m=y0.length; n=200; h=(tn-t0)/n;
```

El último valor de la variable independiente ("tn") se fija en 1.3 porque ese es el último valor requerido.

Se programan ahora las funciones correspondientes a las derivadas primeras del sistema de ecuaciones diferenciales:

```
>>f=[];
>>f[0]=function(t,y){return 2*y[0]+3*y[1];};
>>f[1]=function(t,y){return 2*y[0]+y[1];};
```

Los valores iniciales para este sistema son:

```
>>k=0; vt=[t0]; vy=[y0]; t=t0; y=copy(y0); y_=copy(y0);
```

Ahora se calculan los puntos aplicando la ecuación de Euler (ecuaciones 10.23) a cada uno de los segmentos (guardando los puntos calculados en los vectores "vt" y "vy"):

```
>>for(j=0;j<n;j++){for(i=0;i<m;i++) y[i]+=f[i](t,y_)*h; t=round(t+h,15);
vt[++k]=t; vy[k]=y_=copy(y);};
```

Con los puntos (guardados en los vectores "vt" y "vy"), se generan las funciones de interpolación (en este caso 2):

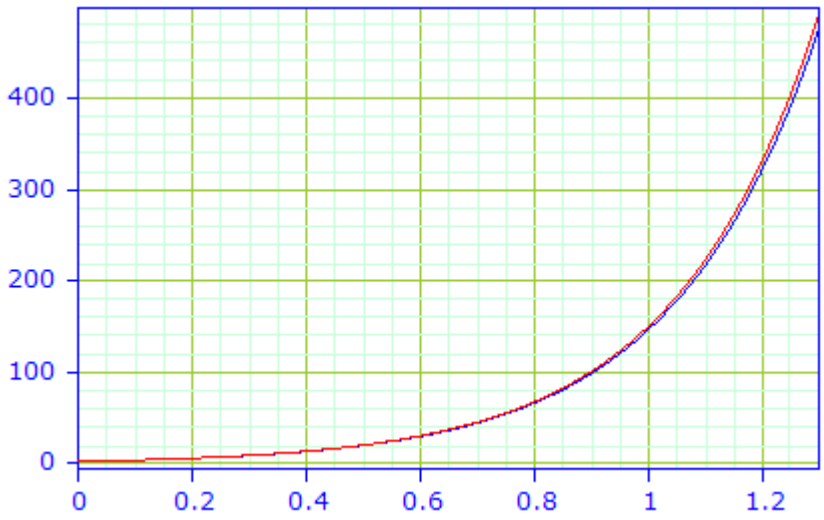
```
>>vy=transpose(vy); fi=[];
>>fi[0]=geninlin(vt,vy[0]);
>>fi[1]=geninlin(vt,vy[1]);
```

Como no se tiene la solución analítica, se generan otras funciones de interpolación para el doble de segmentos (n=400):

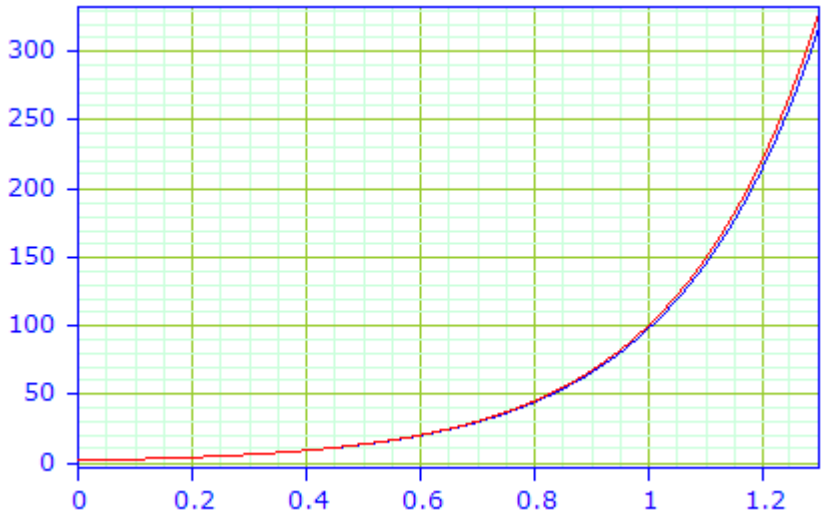
```
>>t0=0; tn=1.3; y0=[2.7,2]; m=y0.length; n=400; h=(tn-t0)/n;
>>k=0; vt=[t0]; vy=[y0]; t=t0; y=copy(y0); y_=copy(y0);
>>for(j=0;j<n;j++){for(i=0;i<m;i++) y[i]+=f[i](t,y_)*h; t=round(t+h,15);
vt[++k]=t; vy[k]=y_=copy(y);};
>>vy=transpose(vy);
>>fi2=[];
>>fi2[0]=geninlin(vt,vy[0]);
>>fi2[1]=geninlin(vt,vy[1]);
```

Y para determinar visualmente si los valores calculados son confiables se dibuja la gráfica de las funciones de interpolación para 200 y 400 segmentos:

```
>>plot([fi[0],fi2[0]],t0,tn)
```



```
>>plot([fi[1],fi2[1]],t0,tn)
```



En ambos casos la variación es pequeña, por lo que se concluye que los resultados calculados con 200 segmentos son confiables (por supuesto que los calculados con 400 lo son aún más).

Los valores pedidos para la primera variable ($x=y_0$), redondeados al 4 dígito después del punto (obtenidos con la última función de interpolación) son:

```
>>round(map(fi2[0],[0.2,0.5,0.7,0.9,1.2,1.3]),4)
[6.1457, 20.4978, 45.486, 100.7921, 332.1913, 494.3091]
```

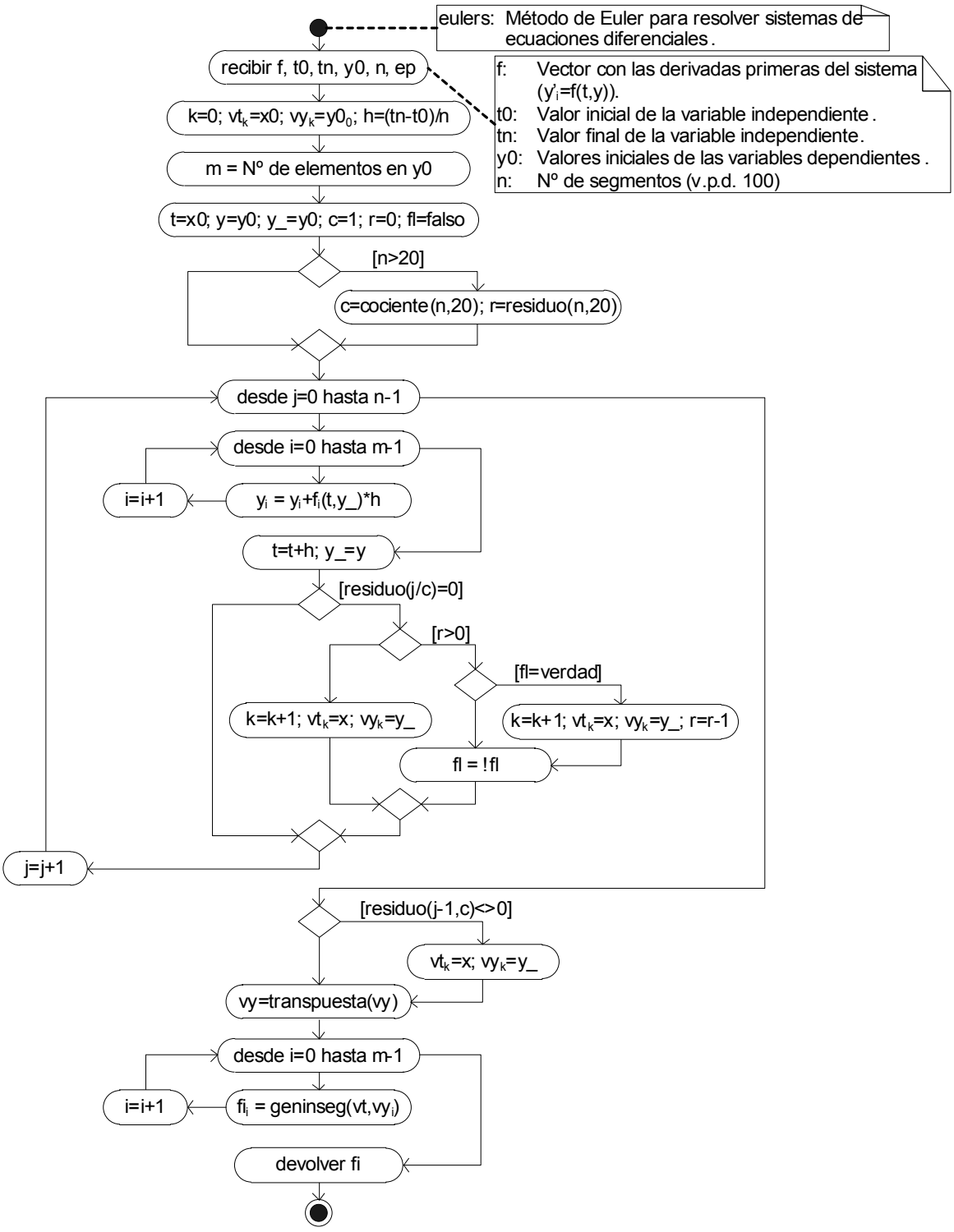
Y para la segunda ($y=y_1$), son:

```
>>round(map(fi2[1],[0.2,0.5,0.7,0.9,1.2,1.3]),4)
[4.2608, 13.7864, 30.4232, 67.276, 221.521, 329.5938]
```

10.2.3.2. Algoritmo y código

El algoritmo del método, para un sistema de ecuaciones diferenciales, se presenta en gráfica de la siguiente página y el código respectivo, añadido a la librería "hpvlib.js", es:

```
function eulers(f,t0,tn,y0,n,np,mi){
  if (n==null) n=100;
  if (np==null) np=20;
  if (mi==null) mi="segmentaria";
  mi.toLowerCase();
  var k=0,vt=[t0],vy=[y0],h=(tn-t0)/n,m=y0.length;
  var t=t0,y=copy(y0),y_=copy(y0),j,c=1,r=0,fl=false;
  var fi=[];
  if (n>np) {c=quot(n,20); r=n%20;};
  for (j=0;j<n;j++){
    for (i=0;i<m;i++) y[i]+=f[i](t,y_)*h;
    y_=copy(y);
    t=round(t+h,15);
    if (j%c == 0)
      if (r>0) {
        if (fl) {vt[++k]=t; vy[k]=y_; r--};
        fl=!fl;}
      else
        {vt[++k]=t; vy[k]=y_;};
  }
```




```
if (--j%c != 0) {vt[k]=t; vy[k]=y_;}
vy=transpose(vy);
switch(mi) {
  case "lagrange": for (i=0;i<m;i++) fi[i]=geninlag(vt,vy[i]); break;
  case "newton": for (i=0;i<m;i++) fi[i]=geninnew(vt,vy[i]); break;
  case "lineal": for (i=0;i<m;i++) fi[i]=geninlin(vt,vy[i]); break;
  default: for (i=0;i<m;i++) fi[i]=geninseg(vt,vy[i]);
}
```

```
    return fi;
}
```

Haciendo correr el programa con los datos del primer ejemplo manual, se obtiene:

```
>>f=[];
>>f[0]=function(t,y){return y[1]-2*y[2]-sqr(t)+2*pow(t,1.5)+1.2*pow(t,0.2)-
10;};
>>f[1]=function(t,y){return -5*y[2]+5*pow(t,1.5)+2*t-35;};
>>f[2]=function(t,y){return 2*y[0]-y[1]+sqr(t)-2*pow(t,1.2)+1.5*sqrt(t)-
10;};
>>fi=eulers(f,0,0.35,[3,-4,-7],40);
>>round(map(fi[0],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),4)
[3.0232, 3.0587, 3.0901, 3.1409, 3.1676, 3.195, 3.2327, 3.2617, 3.2813]
>>round(map(fi[1],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),4)
[-3.9977, -3.99, -3.9801, -3.9589, -3.9455, -3.9303, -3.907, -3.8874,
-3.8733]
>>round(map(fi[2],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),4)
[-6.9908, -6.9714, -6.9514, -6.9154, -6.895, -6.8732, -6.8421, -6.8173,
-6.8001]
```

Que esencialmente son los mismos resultados encontrados en el mencionado ejemplo (no obstante que se han empleado sólo 20 puntos para generar la función de interpolación).

Con los datos del segundo ejemplo manual se obtiene:

```
>>f=[];
>>f[0]=function(t,y){return 2*y[0]+3*y[1];};
>>f[1]=function(t,y){return 2*y[0]+y[1];};
>>fi=eulers(f,0,1.3,[2.7,2],400);
>>round(map(fi[0],[0.2,0.5,0.7,0.9,1.2,1.3]),4)
[6.1455, 20.4974, 45.4849, 100.7889, 334.1582, 494.3091]
>>round(map(fi[1],[0.2,0.5,0.7,0.9,1.2,1.3]),4)
[4.2607, 13.7861, 30.4225, 67.2738, 222.8322, 329.5938]
```

Que en este caso difieren hasta en los tres últimos dígitos con relación a los resultados obtenidos en el ejemplo manual.

10.3. MÉTODOS DE RUNGE – KUTTA

Los métodos de *Runge-Kutta*, son aquellos que tienen la forma general:

$$y_{i+1}=y_i+a_1k_1+a_2k_2+a_3k_3+\ldots+a_nk_n \tag{10.24}$$

Donde los valores de k tienen la forma:

$$\begin{aligned}
k_1&=h\,f(x_i,y_i)\\
k_2&=h\,f(x_i,p_1h,y_i+q_{1,1}k_1)\\
k_3&=h\,f(x_i+p_2h,y_i+q_{2,1}k_1+q_{2,2}k_2)\\
k_4&=h\,f(x_i+p_3h,y_i+q_{3,1}k_1+q_{3,2}k_2+q_{3,3}k_3)\\
&\ldots\\
k_n&=h\,f(x_i+p_{n-1}h,y_i+q_{n-1,1}k_1+q_{n-1,2}k_2+\ldots+q_{n-1,n-1}k_n-1)
\end{aligned} \tag{10.25}$$

Donde $f(x,y)$ es la función correspondiente a la derivada primera de la variable dependiente, x_i, y_i son los valores conocidos de las variables independiente y dependiente para un punto dado i, y_{i+1} es el valor de la varia-

ble dependiente en el siguiente punto y *h* es el incremento de la variable independiente.

Para todas las formas, el valor de la variable independiente del siguiente punto se calcula igual que en el método de *Euler*, es decir:

$$x_{j+1}=x_j+h \tag{10.26}$$

En base a la ecuación (10.24), estableciendo el número de evaluaciones funcionales (valores de k) y calculando los parámetros *"a"*, *"p"* y *"q"*, se puede deducir un número prácticamente infinito de ecuaciones de *Runke-Kutta*.

No todas las formas de Runge-Kutta predicen correctamente los valores de la variable dependiente, razón por la cual una vez deducidas deben ser probadas para determinar su confiabilidad.

Muchos investigadores han deducido y puesto a prueba varias formas de la ecuación de *Runge-Kutta*. Estas ecuaciones se clasifican generalmente como métodos *(m,n)*, donde *"m"* es el número de parámetros *"a_i"* y *"n"* es el número de valores *"k"* que deben ser calculados.

Algunas de las ecuaciones de *Runke-Kutta* que más se utilizan en la práctica son las siguientes:

La ecuación (2,2) de *Runge-Kutta* con una exactitud comparable con la del método de *Euler* modificado:

$$\begin{aligned} y_{j+1}&=y_j+\frac{1}{2}\left(k_1+k_2\right)\\ k_1&=h\,f\left(x_j,y_j\right)\\ k_2&=h\,f\left(x_j+h,y_j+k_1\right)\end{aligned} \tag{10.27}$$

La ecuación (3,3) de *Runge-Kutta* con un error de orden h^4:

$$\begin{aligned} y_{j+1}&=y_j+\frac{1}{6}\left(k_1+4\,k_2+k_3\right)\\ k_1&=h\,f\left(x_j,y_j\right)\\ k_2&=h\,f\left(x_j+\frac{h}{2},y_j+\frac{k_1}{2}\right)\\ k_3&=h\,f\left(x_j+h,y_j-k_1+2\,k_2\right)\end{aligned} \tag{10.28}$$

La ecuación (4,4) de *Runge-Kutta*, conocida como la forma *clásica*:

$$\begin{aligned} y_{j+1}&=y_j+\frac{1}{6}\left(k_1+2k_2+2k_3+k_4\right)\\ k_1&=h\,f\left(x_j,y_j\right)\\ k_2&=h\,f\left(x_j+\frac{h}{2},y_j+\frac{k_1}{2}\right)\\ k_3&=h\,f\left(x_j+\frac{h}{2},y_j+\frac{k_2}{2}\right)\\ k_4&=h\,f\left(x_j+h,y_j+k_3\right)\end{aligned} \tag{10.29}$$

La ecuación (4,4) de *Runge-Kutta* conocida como la forma de *Gil*, la cual minimiza los errores debido a redondeos:

$$
\begin{aligned}
y_{j+1} &= y_j + \frac{1}{6}\left(k_1 + \left(2-\sqrt{2}\right)k_2 + \left(2+\sqrt{2}\right)k_3 + k_4\right) \\
k_1 &= h\,f\left(x_j, y_j\right) \\
k_2 &= h\,f\left(x_j + \frac{h}{2}, y_j + \frac{k_1}{2}\right) \\
k_3 &= h\,f\left(x_j + \frac{h}{2}, y_j + \left(\sqrt{2}-1\right)\frac{k_1}{2} + \left(2-\sqrt{2}\right)\frac{k_2}{2}\right) \\
k_4 &= h\,f\left(x_j + h, y_j - \sqrt{2}\frac{k_2}{2} + \left(1+\frac{\sqrt{2}}{2}\right)k_2\right)
\end{aligned}
\qquad (10.30)
$$

Y una ecuación (5,6) conocida como la forma de *Butcher*, que suele ser empleada cuando se requiere gran exactitud en los cálculos:

$$
\begin{aligned}
y_{j+1} &= y_j + \frac{1}{90}\left(7k_1 + 32k_3 + 12k_4 + 35k_5 + 7k_6\right) \\
k_1 &= h\,f\left(x_j, y_j\right) \\
k_2 &= h\,f\left(x_j + \frac{h}{4}, y_j + \frac{k_1}{4}\right) \\
k_3 &= h\,f\left(x_j + \frac{h}{4}, y_j + \frac{k_1}{8} + \frac{k_2}{8}\right) \\
k_4 &= h\,f\left(x_j + \frac{h}{2}, y_j + \frac{k_2}{2} + k_3\right) \\
k_5 &= h\,f\left(x_j + \frac{3}{4}h, y_j + \frac{3}{16}k_1 + \frac{9}{16}k_4\right) \\
k_6 &= h\,f\left(x_j + h, y_j - \frac{3}{7}k_1 + \frac{2}{7}k_2 + \frac{12}{7}k_3 - \frac{12}{7}k_4 + \frac{8}{7}k_5\right)
\end{aligned}
\qquad (10.31)
$$

En este tema se empleará la forma clásica (ecuación 10.29), por ser la forma que más se emplea en la práctica. En lo sucesivo, cuando se haga referencia al método de *Runge Kutta* se hará referencia a dicha forma.

10.3.1. Resolución de una ecuación diferencial de primer orden

Todas las ecuaciones de *Runge Kutta* permiten resolver sólo ecuaciones diferenciales de primer orden y el procedimiento de cálculo es esencialmente el mismo que en el método de *Euler*, sólo que para el cálculo de la variable dependiente se emplea la ecuación de Runge Kutta (10.29).

10.3.1.1. Ejemplo Manual

Como primer ejemplo se volverá a resolver la ecuación diferencial resuelta con el método de Euler:

$$
y' = f(x,y) = \frac{8x^3 - 4xy - 4x}{3} \quad \left\{ y = 4 \quad para \quad x = 2 \right.
$$

Donde se quiere calcular el valor de "y" para "x=6" y además, calcular valore de "y" para valores de "x" iguales a 2.3, 3.7, 4.6 y 5.7

Primero se crea la función correspondiente a la derivada primera:

```
>>f=function(x,y){return (8*x*x*x-4*x*y-4*x)/3;};
```

Se fija el número de segmentos, en este caso 50 (n=50), se guardan los valores conocidos y se calcula el ancho ("h") de cada segmento:

```
>>n=50; x0=2; xn=6; y0=4; h=(xn-x0)/n;
```

Ahora se aplican las ecuaciones de Runke-Kutta (10.29), dentro de un ciclo for, guardando los puntos calculados en los vectores "vx" y "vy" (para crear la función de interpolación):

```
>>vx=[x0]; vy=[y0]; x=x0; y=y0;
>>for (j=0;j<n;j++){ k1=h*f(x,y); k2=h*f(x+h/2,y+k1/2);
k3=h*f(x+h/2,y+k2/2); k4=h*f(x+h,y+k3); y+=1/6*(k1+2*k2+2*k3+k4);
vy[j+1]=y; x=round(x+h,15); vx[j+1]=x;};
```

Como el último valor de "x" es 6 ("xn"), el valor de "y" para "x=6" es el último valor calculado en el ciclo, es decir el último elemento guardado en el vector "vy":

```
>>vy[n]
68.00014489535796
```

Resultado que sólo tiene un error del 0.0002% con relación al resultado exacto (68). Los otros valores se calculan creando la función de interpolación (en este caso con el método de Lagrange):

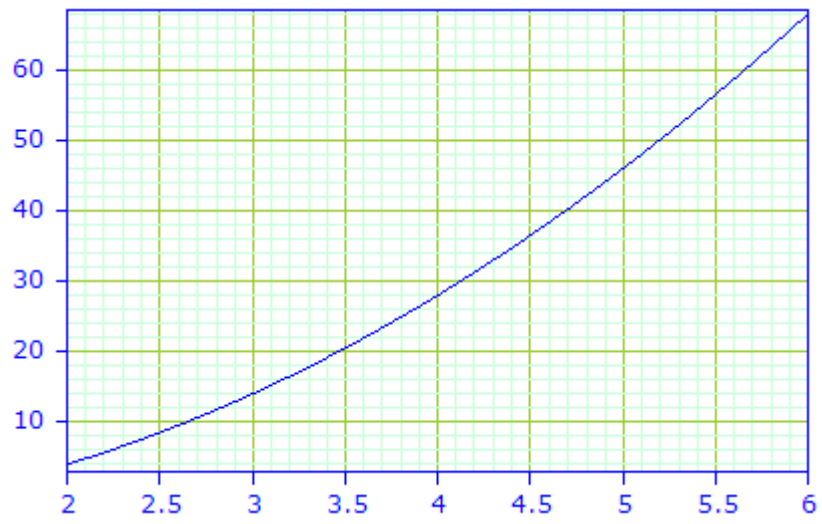
```
>>fi=geninlag(vx,vy) ;
```

Los otros valores pedidos se calculan con esta función y redondeados al tercer dígito después del punto son:

```
>>round(map(fi,[2.3,3.7,4.6,5.7]),3)
[6.58, 23.38, 38.32, 60.98]
```

La curva correspondiente a la solución (entre x=0 y x=6), es:

```
>>plot([fi],x0,xn)
```



Como segundo ejemplo se resolverá la siguiente ecuación diferencial y se calcularán los valores de "y" para valores de "x" iguales a 0.3, 0.7, 0.9, 1.2, 1.5, 1.7 y 2.0:

$$y' = 4e^{0.8x} - 0.5y \quad \lfloor y = 2 \;\; para \;\; x = 0$$

Como es usual, primero se programa la función de la derivada primera:

```
>>f=function(x,y){return 4*exp(0.8*x)-0.5*y;};
```

Se fija el número de segmentos, en este caso 60 (n=60), se guardan los valores conocidos y se calcula el ancho ("h") de cada segmento:

```
>>n=60; x0=0; xn=2; y0=2; h=(xn-x0)/n;
```

Y se aplican las ecuaciones de Runke-Kutta (10.29):

```
>>vx=[x0]; vy=[y0]; x=x0; y=y0;
>>for (j=0;j<n;j++){ k1=h*f(x,y); k2=h*f(x+h/2,y+k1/2);
 k3=h*f(x+h/2,y+k2/2); k4=h*f(x+h,y+k3); y+=1/6*(k1+2*k2+2*k3+k4);
vy[j+1]=y; x=round(x+h,15); vx[j+1]=x;};
```

Con los puntos calculados, guardados en los vectores "vx" y "vy", se crea la función de interpolación (empleando el método segmentario):

```
>>fi=geninseg(vx,vy);
```

Y con esta función se calculan los valores pedidos:

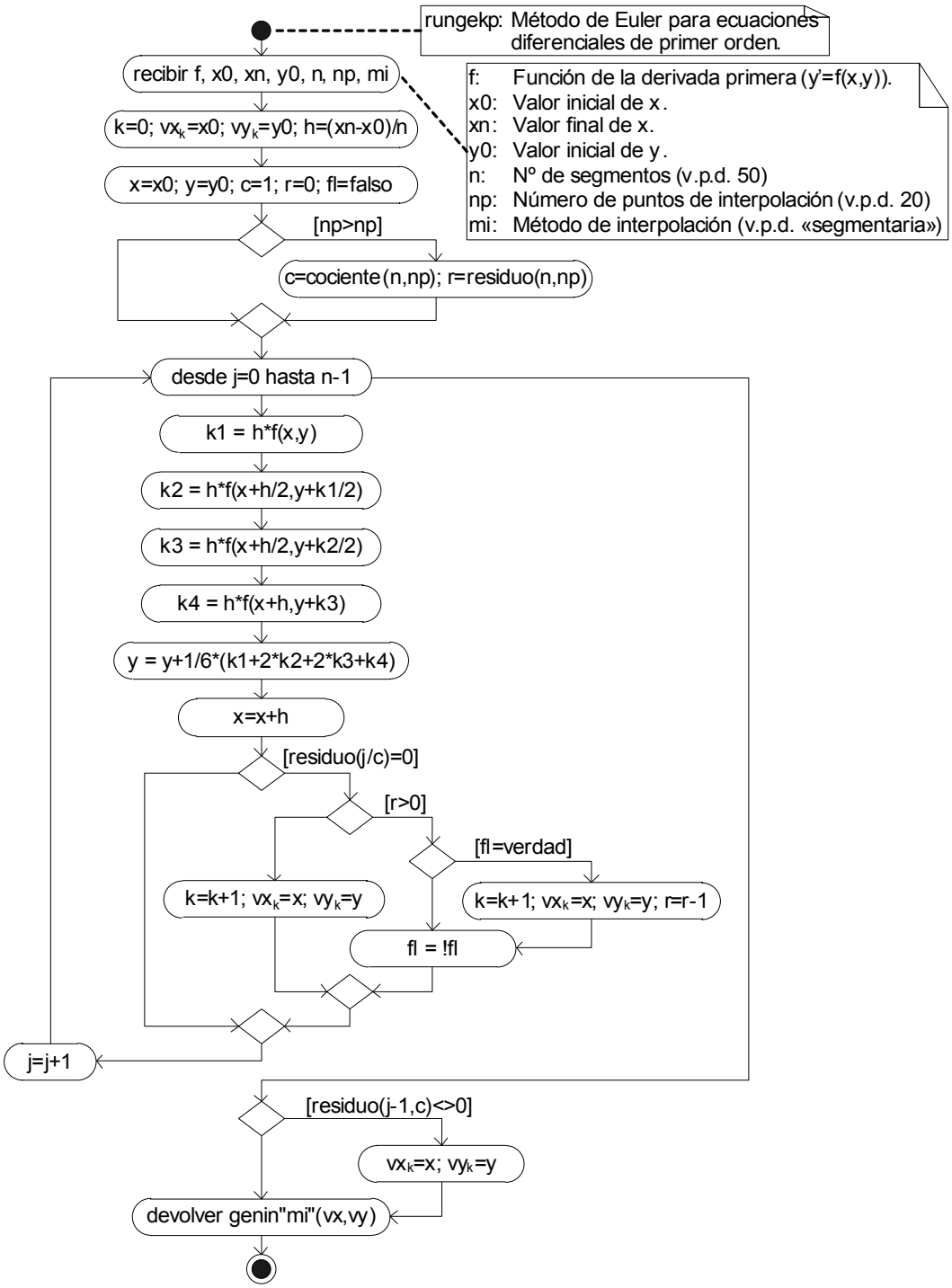
```
>>round(map(fi,[0.3,0.7,0.9,1.2,1.5,1.7,2.0]),3)
[2.985, 4.628, 5.635, 7.445, 9.707, 11.528, 14.844]
```

10.3.1.2. Algoritmo y código

El algoritmo del método se presenta en la siguiente página. Con excepción de las ecuaciones, el algoritmo del método de Runge-Kutta, es esencialmente el mismo que el de Euler.

El algoritmo elaborado en base al mismo y añadido a la librería "hpv-lib.js" es:

```
function rungekp(f,x0,xn,y0,n,np,mi){
  if (n==null) n=100;
  if (np==null) np=20;
  if (mi==null) mi="segmentaria";
  mi=mi.toLowerCase();
  var vx=[x0],vy=[y0],h=(xn-x0)/n,x=x0,y=y0,j,k=0,c=1,r=0,fl=false;
  var k1,k2,k3,k4;
  if (n>np) {c=quot(n,np); r=n%np;};
  for (j=0;j<n;j++){
    k1=h*f(x,y);
    k2=h*f(x+h/2,y+k1/2);
    k3=h*f(x+h/2,y+k2/2);
    k4=h*f(x+h,y+k3);
    y+=1/6*(k1+2*k2+2*k3+k4);
    x=round(x+h,15);
    if (j%c == 0)
      if (r>0) {
        if (fl) {vx[++k]=x; vy[k]=y; r--};
        fl=!fl;}
      else
        {vx[++k]=x; vy[k]=y;};
  }
  if (--j%c != 0) {vx[k]=x; vy[k]=y;}
  switch(mi) {
    case "lagrange": return geninlag(vx,vy);
    case "newton": return geninnew(vx,vy);
    case "lineal": return geninlin(vx,vy);
    default: return geninseg(vx,vy);
  }
}
```



Haciendo correr el programa con los datos del primer ejemplo manual:

```
>>f=function(x,y){return (8*x*x*x-4*x*y-4*x)/3;};
>>fi=rungekp(f,2,6,4,50,50,"lagrange");
```

Se obtienen los mismos resultados que en dicho ejemplo:

```
>>round(map(fi,[2.3,3.7,4.6,5.7,6]),3)
[6.58, 23.38, 38.32, 60.98, 68]
```

Igualmente con los datos del segundo ejemplo manual:

```
>>f=function(x,y){return 4*exp(0.8*x)-0.5*y;};
```

```
>>fi=rungekp(f,0,2,2,60,60);
```

Se obtienen los mismos resultados:

```
>>round(map(fi,[0.3,0.7,0.9,1.2,1.5,1.7,2.0]),3)
[2.985, 4.628, 5.635, 7.445, 9.707, 11.528, 14.844]
```

10.3.2. Ecuaciones diferenciales de enésimo orden

Al igual que con el método de *Euler*, las ecuaciones de *Runge-Kutta* pueden ser empleadas para resolver ecuaciones diferenciales de enésimo orden. Para ello se sigue el mismo procedimiento que en el método de *Euler*, es decir se transforma la ecuación diferencial en un sistema de ecuaciones diferenciales de primer orden y se aplica la ecuación de *Runge-Kutta* (ecuación 10.29) a cada una de las ecuaciones resultantes.

En el caso de la forma clásica (que como se dijo es la forma a emplear en este tema) implica el cálculo de los cuatro parámetros de la ecuación de *Runge-Kutta* para cada una de las ecuaciones resultantes de la transformación. Así, como ya se vio en el método de Euler, la siguiente ecuación:

$$3x\frac{d^4y}{dx^4}+(x-y)\frac{d^3y}{dx^3}+5\frac{d^2y}{dx^2}-\frac{dy}{dx}-\frac{x^2-y^2}{xy}$$
$$=3xy''''+(x-y)y'''+5y''-y'-\frac{x^2-y^2}{xy}=0$$

Puede ser transformada en el siguiente sistema de ecuaciones lineales:

$$y'_3=y_0''''=f\left(x,y_0,y_1,y_2,y_3\right)=\frac{\left(y_0-x\right)y_3-5y_2+y_1+\frac{\left(x^2-y_0^2\right)}{xy_0}}{3x}=y_4$$
$$y'_0=y_1$$
$$y'_1=y_0''=y_2$$
$$y'_2=y_0'''=y_3$$

Donde "y_0" es el nuevo nombre que se da a la variable dependiente "y", pues ahora "y" es el vector formado por la variable dependiente "y_0" y las variables correspondientes a las derivadas (y_1, y_2, y_3). Aplicando las ecuaciones de Runge-Kutta (10.29) a este sistema se tiene:

$$k_{1,0}=hy_{1,j}$$
$$k_{1,1}=hy_{2,j}$$
$$k_{1,2}=hy_{3,j}$$
$$k_{1,3}=hf\left(x_j,y_{0,j},y_{1,j},y_{2,j},y_{3,j}\right)$$
$$k_{2,0}=h\left(y_{1,j}+\frac{k_{1,1}}{2}\right)$$
$$k_{2,1}=h\left(y_{2,j}+\frac{k_{1,2}}{2}\right)$$
$$k_{2,2}=h\left(y_{3,j}+\frac{k_{1,3}}{2}\right)$$

$$k_{2,3}=h\,f\left(x_j+\frac{h}{2},\,y_{0,j}+\frac{k_{1,0}}{2},\,y_{1,j}+\frac{k_{1,1}}{2},\,y_{2,j}+\frac{k_{1,2}}{2},\,y_{3,j}+\frac{k_{1,3}}{2}\right)$$
$$k_{3,0}=h\left(y_{1,j}+\frac{k_{2,1}}{2}\right)$$
$$k_{3,1}=h\left(y_{2,j}+\frac{k_{2,2}}{2}\right)$$
$$k_{3,2}=h\left(y_{3,j}+\frac{k_{2,3}}{2}\right)$$
$$k_{3,3}=h\,f\left(x_j+\frac{h}{2},\,y_{0,j}+\frac{k_{2,0}}{2},\,y_{1,j}+\frac{k_{2,1}}{2},\,y_{2,j}+\frac{k_{2,2}}{2},\,y_{3,j}+\frac{k_{2,3}}{2}\right)$$
$$k_{4,0}=h\left(y_{1,j}+k_{3,1}\right)$$
$$k_{4,1}=h\left(y_{2,j}+k_{3,2}\right)$$
$$k_{4,2}=h\left(y_{3,j}+k_{3,3}\right)$$
$$k_{4,3}=h\,f\left(x_j+h,\,y_{0,j}+k_{3,0},\,y_{1,j}+k_{3,1},\,y_{2,j}+k_{3,2},\,y_{3,j}+k_{2,3}\right)$$
$$y_{0,j+1}=y_{0,j}+\frac{1}{6}\left(k_{1,0}+2k_{2,0}+2k_{3,0}+k_{4,0}\right)$$
$$y_{1,j+1}=y_{1,j}+\frac{1}{6}\left(k_{1,1}+2k_{2,1}+2k_{3,1}+k_{4,1}\right)$$
$$y_{2,j+1}=y_{2,j}+\frac{1}{6}\left(k_{1,2}+2k_{2,2}+2k_{3,2}+k_{4,2}\right)$$
$$y_{3,j+1}=y_{3,j}+\frac{1}{6}\left(k_{1,3}+2k_{2,3}+2k_{3,3}+k_{4,3}\right)$$
$$x_j=x_j+h$$

De este ejemplo, se pueden deducir las ecuaciones generales para una ecuación diferencial general de orden "m" (en un segmento "j" cualquiera):

$$\begin{aligned}
&k_{1,i}=h\,y_{i+1,j}\quad\{i=0\rightarrow m-2\\
&k_{1,m-1}=h\,f\left(x_j,\,y_{0,j},\,y_{1,j},\ldots,\,y_{m-1,j}\right)\\
&k_{2,i}=h\left(y_{i+1,j}+\frac{k_{1,i+1}}{2}\right)\quad\{i=0\rightarrow m-2\\
&k_{2,m-1}=h\,f\left(x_j+\frac{h}{2},\,y_{0,j}+\frac{k_{1,0}}{2},\,y_{1,j}+\frac{k_{1,1}}{2},\ldots,\,y_{m-1,j}+\frac{k_{1,m-1}}{2}\right)\\
&k_{3,i}=h\left(y_{i+1,j}+\frac{k_{2,i+1}}{2}\right)\quad\{i=0\rightarrow m-2\\
&k_{3,m-1}=h\,f\left(x_j+\frac{h}{2},\,y_{0,j}+\frac{k_{2,0}}{2},\,y_{1,j}+\frac{k_{2,1}}{2},\ldots,\,y_{m-1,j}+\frac{k_{2,m-1}}{2}\right)\\
&k_{4,i}=h\left(y_{i+1,j}+k_{3,i+1}\right)\quad\{i=0\rightarrow m-2\\
&k_{4,m-1}=h\,f\left(x_j+h,\,y_{0,j}+k_{3,0},\,y_{1,j}+k_{2,1},\ldots,\,y_{m-1,j}+k_{3,m-1}\right)
\end{aligned}\qquad(10.32)$$

$$y_{i,j+1}=y_{i,j}+\frac{1}{6}\left(k_{1,i}+2k_{2,i}+2k_{3,i}+k_{4,i}\right) \quad \left\{ i=0 \rightarrow m-1$$
$$x_j=x_j+h$$

10.3.2.1. Ejemplo manual

Para comprender mejor el procedimiento que se sigue en el método de Runge-Kutta, se volverá a resolver la siguiente ecuación diferencial, encontrando el valor de la variable dependiente "y" para "x" igual a 1.3, 1.5, 1.8, 2, 2.4, 2.9, 3.3, 3.7, 3.8 y 4.

$$3y''''-3xy''+y'+1.8711x^{-1.9}+14.49x^{1.1}-5=0$$
$$y=4,\ y'=11.3,\ y''=6.93,\ y'''=0.693 \quad para \quad x=1$$

Se sabe que la solución analítica de esta ecuación diferencial es:

$$y=3x^{2.1}+5x-4$$

Para efectos de comparación se programa la misma:

```
>>fc=function(x){return 3*pow(x,2.1)+5*x-4;};
```

Se despeja el término de mayor orden:

$$y'_3=y_4=f(x,y)=\frac{3xy_2-y_1-1.8711x^{-1.9}-14.49x^{1.1}+5}{3}$$

Y se programa la función resultante:

```
>>f=function(x,y){return (3*x*y[2]-y[1]-1.8711*pow(x,-1.9)-
14.49*pow(x,1.1)+5)/3;};
```

Se fija el número de segmentos (en este ejemplo 10), se guardan los valores conocidos (el último valor de "x" (xn) es 4, porque es el último valor requerido) y se calcula el ancho de los segmentos ("h"):

```
>>n=10; x0=1; xn=4; y0=[4,11.3,6.93,0.693]; m=y0.length; h=(xn-x0)/n;
```

Ahora se pueden aplicar la ecuaciones de Runge-Kutta (ecuaciones 10.32), para calcular los 10 puntos de los 10 segmentos (n=10), inicializando las variables que forma parte del proceso:

```
>>vx=[x0]; vy=[y0]; k=0; x=x0; y=copy(y0); k1=new Array(m-1); k2=new
Array(m-1); k3=new Array(m-1); k4=new Array(m-1);
```

Donde "vx" y "vy" son los vectores donde se guardan los puntos calculados (para generar la función de interplación). El cálculo propiamente se lleva a cabo dentro de un ciclo "for" que va desde "j=0" hasta "n-1" (para los 10 puntos) y dentro ciclos "for" para el cálculo de los valores de "k".

```
>>for(j=0;j<n;j++){for(i=0;i<m-1;i++) k1[i]=h*y[i+1]; k1[m-1]=h*f(x,y);
for(i=0;i<m-1;i++) k2[i]=h*(y[i+1]+k1[i+1]/2); k2[m-
1]=h*f(x+h/2,add(y,div(k1,2))); for(i=0;i<m-1;i++)
k3[i]=h*(y[i+1]+k2[i+1]/2); k3[m-1]=h*f(x+h/2,add(y,div(k2,2)));
for(i=0;i<m-1;i++) k4[i]=h*(y[i+1]+k3[i+1]); k4[m-1]=h*f(x+h,add(y,k3));
for(i=0;i<m;i++) y[i]+=(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6; x=round(x+h,15);
vy[++k]=copy(y); vx[k]=x;};
```

Para calcular los valores requeridos se genera la función de interpolación empleando en este caso el método de Newton:

```
>>vy=transpose(vy);
>>fi=geninnew(vx,vy[0]);
```

Siendo los valores pedidos (redondeados al cuarto dígito):

```
>>round(map(fi,[1.3,1.5,1.8,2,2.4,2.9,3.3,3.7,3.8,4]),4)
[7.7048, 10.5293, 15.3084, 18.8612, 26.861, 38.5644, 49.3128, 61.3104,
64.5067, 71.1372]
```

Las soluciones exactas, obtenidas con la función de solución, son:

```
>>round(map(fc,[1.3,1.5,1.8,2,2.4,2.9,3.3,3.7,3.8,4]),4)
[7.7048, 10.5293, 15.3085, 18.8613, 26.861, 38.5645, 49.3129, 61.3107,
64.507, 71.1375]
```

A pesar de haberse empleado sólo 10 segmentos, los valores calculados con el método de Runge-Kutta son muy cercanos a las soluciones correctas, lo que corrobora la mayor precisión del método.

Como un segundo ejemplo se resolverá la siguiente ecuación diferencial, encontrando los valores de "y" para valores de "t" iguales a 0.2, 0.3, 0.4, 0.8, 1.1, 1.4 y 1.5:

$$y'''+t\,y''-t\,y'-2\,y=t \quad \{y=0;\ y'=0;\ y''=0 \ \ para \ \ t=0$$

Al igual que en el ejemplo anterior se despeja la derivada de mayor orden:

$$y'_2=y_3=t+2\,y_0+t\,y_1-t\,y_2$$

Y se programa la función resultante:

```
>>f=function(t,y){return t+2*y[0]+t*y[1]-t*y[2];};
```

Puesto que los valores a calcular son menores a 1.5, ese será el valor final de "t" (tn). Los valores iniciales vienen dados en la definición de la ecuación. Asumiendo 25 segmentos (n=25), los valores iniciales y el incremento "h" para este ejemplo son:

```
>>t0=0; tn=1.5; y0=[0,0,0]; m=y0.length; n=25; h=(tn-t0)/n;
```

Se inician las variables que forman parte del proceso y se aplican las ecuaciones de Runge-Kutta (10.32) (guardando los puntos calculados en los vectores *"vt"* y *"vy"*):

```
>>vt=[t0]; vy=[y0]; k=0; t=t0; y=copy(y0); k1=new Array(m-1); k2=new
Array(m-1); k3=new Array(m-1); k4=new Array(m-1);
>>for(j=0;j<n;j++){for(i=0;i<m-1;i++) k1[i]=h*y[i+1]; k1[m-1]=h*f(t,y);
for(i=0;i<m-1;i++) k2[i]=h*(y[i+1]+k1[i+1]/2); k2[m-
1]=h*f(t+h/2,add(y,div(k1,2))); for(i=0;i<m-1;i++)
k3[i]=h*(y[i+1]+k2[i+1]/2); k3[m-1]=h*f(t+h/2,add(y,div(k2,2)));
for(i=0;i<m-1;i++) k4[i]=h*(y[i+1]+k3[i+1]); k4[m-1]=h*f(t+h,add(y,k3));
for(i=0;i<m;i++) y[i]+=(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6; t=round(t+h,15);
vy[++k]=copy(y); vt[k]=t;};
```

Con los vectores "vt" y "vy" se crea la función de interpolación (en este caso con el método de Lagrange):

```
>>vy=transpose(vy);
>>fi=geninlag(vt,vy[0]);
```

Y con esta función se calculan los valores de "y" requeridos (redondeados al quinto dígito después del punto):

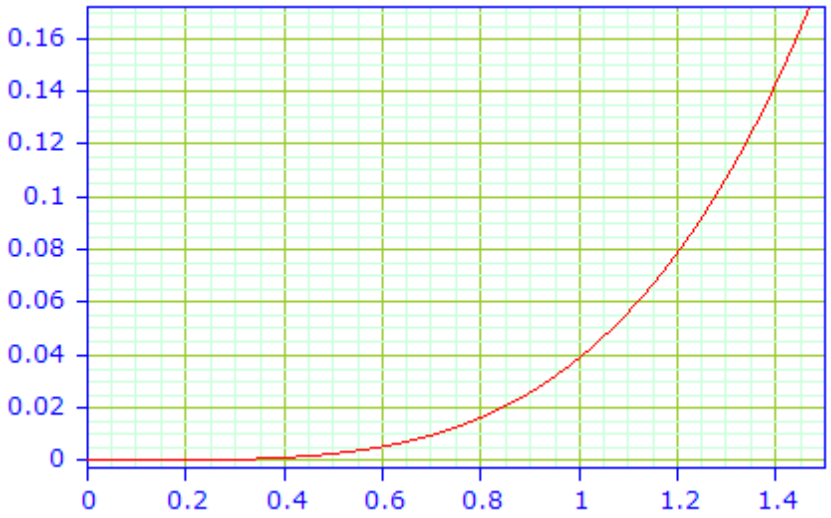
```
>>round(map(fi,[0.2,0.3,0.4,0.8,1.1,1.4,1.5]),5)
[0.00007, 0.00033, 0.00105, 0.01626, 0.05635, 0.14346, 0.18733]
```

Como no se cuenta con la solución analítica, para determinar si los resultados obtenidos son confiables, se repite el cálculo pero empleando un número de segmentos igual al doble del anterior (n=50):

```
>>t0=0; tn=1.5; y0=[0,0,0]; m=y0.length; n=50; h=(tn-t0)/n;
>>vt=[t0]; vy=[y0]; k=0; t=t0; y=copy(y0); k1=new Array(m-1); k2=new
Array(m-1); k3=new Array(m-1); k4=new Array(m-1);
>>for(j=0;j<n;j++){for(i=0;i<m-1;i++) k1[i]=h*y[i+1]; k1[m-1]=h*f(t,y);
for(i=0;i<m-1;i++) k2[i]=h*(y[i+1]+k1[i+1]/2); k2[m-
1]=h*f(t+h/2,add(y,div(k1,2))); for(i=0;i<m-1;i++)
k3[i]=h*(y[i+1]+k2[i+1]/2); k3[m-1]=h*f(t+h/2,add(y,div(k2,2)));
for(i=0;i<m-1;i++) k4[i]=h*(y[i+1]+k3[i+1]); k4[m-1]=h*f(t+h,add(y,k3));
for(i=0;i<m;i++) y[i]+=(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6; t=round(t+h,15);
vy[++k]=copy(y); vt[k]=t;};
>>vy=transpose(vy);
>>fi2=geninlag(vt,vy[0]);
```

Y se grafican las dos funciones de interpolación:

```
>>plot([fi,fi2],t0,tn)
```



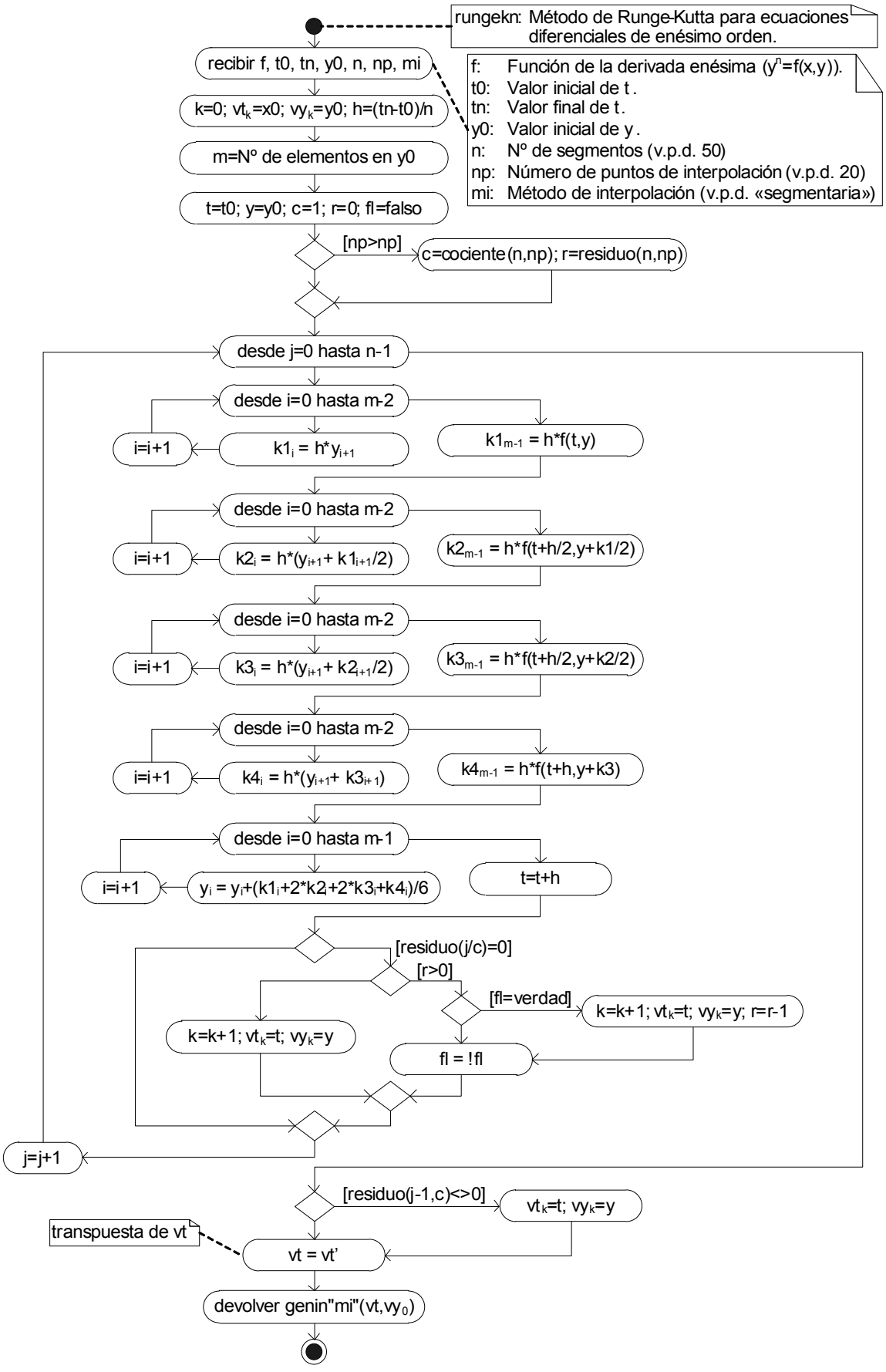
Como se puede ver en este caso las dos curvas son iguales (se superponen), por lo que se sabe que los datos calculados son confiables, corroborando una vez más la precisión del método.

10.3.2.2. Algoritmo y código

El algoritmo del método de Runge-Kutta se presenta en la figura de la siguiente página. Práctiamente sólo difiere del método de Euler en las ecuaciones empleadas.

El código elaborado en base al mismo y añadido a la librería "hpvlib.js" es:

```
function rungekn(f,t0,tn,y0,n,np,mi){
  if (n==null) n=50;
  if (np==null) np=20;
  if (mi==null) mi="segmentaria";
  mi.toLowerCase();
  var k=0,vt=[t0],vy=[y0],h=(tn-t0)/n,m=y0.length;
  var t=t0,y=copy(y0),j,c=1,r=0,fl=false,k1=new Array(m-1);
  var k2=new Array(m-1), k3=new Array(m-1), k4=new Array(m-1);
  if (n>np) {c=quot(n,np); r=np;};
  for (j=0;j<n;j++){
    for(i=0;i<m-1;i++) k1[i]=h*y[i+1];
    k1[m-1]=h*f(t,y);
    for(i=0;i<m-1;i++) k2[i]=h*(y[i+1]+k1[i+1]/2);
    k2[m-1]=h*f(t+h/2,add(y,div(k1,2)));
```


rungekn: Método de Runge-Kutta para ecuaciones diferenciales de enésimo orden.
recibir f, t0, tn, y0, n, np, mi
f: Función de la derivada enésima ($y^n=f(x,y)$).
t0: Valor inicial de t.
tn: Valor final de t.
y0: Valor inicial de y.
n: Nº de segmentos (v.p.d. 50)
np: Número de puntos de interpolación (v.p.d. 20)
mi: Método de interpolación (v.p.d. «segmentaria»)
k=0; vt_k=x0; vy_k=y0; h=(tn-t0)/n
m=Nº de elementos en y0
t=t0; y=y0; c=1; r=0; fl=falso
[np>np]
c=cociente(n,np); r=residuo(n,np)
desde j=0 hasta n-1
desde i=0 hasta m-2
$k1_i = h*y_{i+1}$
i=i+1
$k1_{m-1} = h*f(t,y)$
desde i=0 hasta m-2
$k2_i = h*(y_{i+1}+ k1_{i+1}/2)$
i=i+1
$k2_{m-1} = h*f(t+h/2,y+k1/2)$
desde i=0 hasta m-2
$k3_i = h*(y_{i+1}+ k2_{i+1}/2)$
i=i+1
$k3_{m-1} = h*f(t+h/2,y+k2/2)$
desde i=0 hasta m-2
$k4_i = h*(y_{i+1}+ k3_{i+1})$
i=i+1
$k4_{m-1} = h*f(t+h,y+k3)$
desde i=0 hasta m-1
$y_i = y_i+(k1_i+2*k2_i+2*k3_i+k4_i)/6$
i=i+1
t=t+h
[residuo(j/c)=0]
[r>0]
[fl=verdad]
k=k+1; vt_k=t; vy_k=y; r=r-1
k=k+1; vt_k=t; vy_k=y
fl = !fl
j=j+1
[residuo(j-1,c)<>0]
vt_k=t; vy_k=y
transpuesta de vt
vt = vt'
devolver genin"mi"(vt,vy_0)

```
      for(i=0;i<m-1;i++) k3[i]=h*(y[i+1]+k2[i+1]/2);
      k3[m-1]=h*f(t+h/2,add(y,div(k2,2)));
      for(i=0;i<m-1;i++) k4[i]=h*(y[i+1]+k3[i+1]);
      k4[m-1]=h*f(t+h,add(y,k3));
      for(i=0;i<m;i++) y[i]+=(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6;
      t=round(t+h,15);
      if (j%c == 0)
        if (r>0) {
          if (fl) {vt[++k]=t; vy[k]=copy(y); r--};
          fl=!fl;}
        else
          {vt[++k]=t; vy[k]=copy(y);};
    }
    if (--j%c != 0) {vt[k]=t; vy[k]=copy(y);}
    vy=transpose(vy);
    switch(mi) {
      case "lagrange": return geninlag(vt,vy[0]);
      case "newton": return geninnew(vt,vy[0]);
      case "lineal": return geninlin(vt,vy[0]);
      default: return geninseg(vt,vy[0]);
    }
  }
```

Haciendo correr el programa con los datos del primer ejemplo manual, se obtienen los mismos resultados que en dicho ejemplo:

```
>>f=function(x,y){return (3*x*y[2]-y[1]-1.8711*pow(x,-1.9)-
14.49*pow(x,1.1)+5)/3;};
>>fi=rungekn(f,1,4,[4,11.3,6.93,0.693],10,10,"newton");
>>round(map(fi,[1.3,1.5,1.8,2,2.4,2.9,3.3,3.7,3.8,4]),4)
[7.7048, 10.5293, 15.3084, 18.8612, 26.861, 38.5644, 49.3128, 61.3104,
64.5067, 71.1372]
```

Igualmente, haciendo correr el programa con los datos del segundo ejemplo manual, se obtienen los mismos resultados que en dicho ejemplo:

```
>>f=function(t,y){return t+2*y[0]+t*y[1]-t*y[2];};
>>fi=rungekn(f,0,1.5,[0,0,0],25,25,"lagrange");
>>round(map(fi,[0.2,0.3,0.4,0.8,1.1,1.4,1.5]),5)
[0.00007, 0.00033, 0.00105, 0.01626, 0.05635, 0.14346, 0.18733]
```

10.3.3. Sistemas de ecuaciones diferenciales

Al igual que el método de *Euler*, los métodos de *Runge-Kutta* pueden ser empleados para resolver sistemas de ecuaciones diferenciales de primer orden:

$$\begin{aligned} y'_0 &= f_0(t, y_0, y_1, y_2, \dots, y_{m-1}) \\ y'_1 &= f_1(t, y_0, y_1, y_2, \dots, y_{m-1}) \\ &\dots \\ y'_{m-1} &= f_{m-1}(t, y_0, y_1, y_2, \dots, y_{m-1}) \end{aligned} \qquad (10.22)$$

En este sistema "*t*" es la variable independiente, y_0, y_1, y_2, ... , y_{m-1} es el vector de variables dependientes "y".

La ecuación general para resolver un sistema de ecuaciones diferenciales por el método de Runge-Kutta, se deduce simplemente aplicando las ecuaciones de *Runge-Kutta* a cada una de las ecuaciones en un segmento "j" cualquiera:

$$
\begin{aligned}
k_{1,i} &= h f_i\left(t_j, y_{0,j}, y_{1,j}, \ldots, y_{m,j}\right) \quad \left\{ i=0 \rightarrow m-1 \right. \\
k_{2,i} &= h f_i\left(t_j+\frac{h}{2}, y_{0,j}+\frac{k_{1,0}}{2}, y_{1,j}+\frac{k_{1,1}}{2}, \ldots, y_{m-1,j}+\frac{k_{1,m-1}}{2}\right) \quad \left\{ i=0 \rightarrow m-1 \right. \\
k_{3,i} &= h f_i\left(t_j+\frac{h}{2}, y_{0,j}+\frac{k_{2,0}}{2}, y_{1,j}+\frac{k_{2,1}}{2}, \ldots, y_{m-1,j}+\frac{k_{2,m-1}}{2}\right) \quad \left\{ i=0 \rightarrow m-1 \right. \qquad (10.33) \\
k_{4,i} &= h f_i\left(t_j+h, y_{0,j}+k_{3,0}, y_{1,j}+k_{2,1}, \ldots, y_{m-1,j}+k_{3,m-1}\right) \quad \left\{ i=0 \rightarrow m-1 \right. \\
y_{i,j+1} &= y_{i,j}+\frac{1}{6}\left(k_{1,i}+2k_{2,i}+2k_{3,i}+k_{4,i}\right) \quad \left\{ i=0 \rightarrow m-1 \right. \\
t_j &= t_j+h
\end{aligned}
$$

Al igual que sucede con una ecuación diferencial de enésimo orden, primero se calculan los valores del vector "k_1", luego los de "k_2", "k_3", "k_4" y finalmente los valores de las variables dependientes y el de la independiente.

10.3.3.1. Ejemplo Manual

Para comprender mejor el proceso que se sigue al resolver sistemas de ecuaciones diferenciales, se resolverá el siguiente sistema, calculando los valores de las variables dependientes ("u", "v" y "w") para valores de "t" iguales a: 0.05, 0.1, 0.14, 0.20, 0.23, 0.26, 0.30, 0.33 y 0.35.

$$
\left.
\begin{aligned}
u' - v + 2w + t^2 - 2t^{1.5} - 1.2t^{0.2} + 10 = 0 \\
v' + 5w - 5t^{1.5} - 2t + 35 = 0 \\
w' - 2u + v - t^2 + 2t^{1.2} - 1.5t^{0.5} + 10 = 0
\end{aligned}
\right\} \; u=3, \; v=-4, \; w=-7 \;\; para \;\; t=0
$$

Se sabe que las soluciones del sistema son:

$$
\begin{aligned}
u &= t^{1.2}+3 \\
v &= t^2-4 \\
w &= t^{1.5}-7
\end{aligned}
$$

Primero, se reescriben las ecuaciones en función de las variables y_0, y_1 y y_2 (en lugar de "u", "v" y "w"):

$$
\left.
\begin{aligned}
y_0' &= f_0(t,y) = y_1 - 2y_2 - t^2 + 2t^{1.5} + 1.2t^{0.2} - 10 \\
y_1' &= f_1(t,y) = -5y_2 + 5t^{1.5} + 2t - 35 \\
y_2' &= f_2(t,y) = 2y_0 - y_1 + t^2 - 2t^{1.2} + 1.5t^{0.5} - 10 = 0
\end{aligned}
\right\} \; y_0=3, y_1=-4, y_2=-7 \;\; para \;\; t=0
$$

Siendo las soluciones:

$$
\begin{aligned}
y_0 &= t^{1.2}+3 \\
y_1 &= t^2-4 \\
y_2 &= t^{1.5}-7
\end{aligned}
$$

Para efectos de comparación, se programa primero estas soluciones:

```
>>fc=[];
```

```
>>fc[0]=function(t){return pow(t,1.2)+3;};
>>fc[1]=function(t){return sqr(t)-4;};
>>fc[2]=function(t){return pow(t,1.5)-7;};
```

Y se programan las funciones correspondientes a las derivadas primeras:

```
>>f=[];
>>f[0]=function(t,y){return y[1]-2*y[2]-sqr(t)+2*pow(t,1.5)+1.2*pow(t,0.2)-
10;};
>>f[1]=function(t,y){return -5*y[2]+5*pow(t,1.5)+2*t-35;};
>>f[2]=function(t,y){return 2*y[0]-y[1]+sqr(t)-2*pow(t,1.2)+1.5*sqrt(t)-
10;};
```

Y para iniciar el proceso, se guardan los valores iniciales y se calcula el valor de "h", fijando en este caso el número de segmentos en 40:

```
>>t0=0; tn=0.35; y0=[3,-4,-7]; m=y0.length; n=40; h=(tn-t0)/n;
```

Ahora se deben generar los puntos para cada una de las ecuaciones del sistema aplicando las ecuaciones de Euler (ecuaciones 10.33) a cada uno de los segmentos (en un ciclo "for").

Los valores de la variable independiente ("t") y dependiente ("y_0") se guardan (para efectos de interpolación) en los vector "vt" y "vy". Los valores iniciales de las variables que forman parte del proceso son:

```
>>vt=[t0]; vy=[y0]; k=0; t=t0; y=copy(y0); k1=new Array(m-1); k2=new
Array(m-1); k3=new Array(m-1); k4=new Array(m-1);
```

Ahora se generan los 40 puntos (para los 40 segmentos):

```
>>for(j=0;j<n;j++){for(i=0;i<m;i++) k1[i]=h*f[i](t,y); t_=t+h/2;
y_=add(y,div(k1,2)); for(i=0;i<m;i++) k2[i]=h*f[i](t_,y_);
y_=add(y,div(k2,2)); for(i=0;i<m;i++) k3[i]=h*f[i](t_,y_); t_=t+h;
y_=add(y,k3); for(i=0;i<m;i++) k4[i]=h*f[i](t_,y_); for(i=0;i<m;i++) y[i]
+=(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6; t=round(t+h,15); vy[++k]=copy(y);
vt[k]=t;};
```

Entonces se crean las funciones de interpolación (empleando en este caso el método de Lagrange):

```
>>vy=transpose(vy);
>>fi=[];
>>fi[0]=geninlag(vt,vy[0]);
>>fi[1]=geninlag(vt,vy[1]);
>>fi[2]=geninlag(vt,vy[2]);
```

Entonces se calculan los valores pedidos empleando tanto la función de interpolación como las soluciones conocidas, siendo los resultados para la primera ecuación:

```
>>round(map(fi[0],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),3)
[3.027, 3.063, 3.094, 3.145, 3.171, 3.198, 3.236, 3.264, 3.284]
>>round(map(fc[0],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),3)
[3.027, 3.063, 3.094, 3.145, 3.171, 3.199, 3.236, 3.264, 3.284]
```

Para la segunda:

```
>>round(map(fi[1],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),3)
[-3.997, -3.99, -3.98, -3.96, -3.947, -3.932, -3.91, -3.891, -3.877]
>>round(map(fc[1],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),3)
[-3.998, -3.99, -3.98, -3.96, -3.947, -3.932, -3.91, -3.891, -3.877]
```

Y para la tercera:

```
>>round(map(fi[2],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),3)
[-6.989, -6.968, -6.948, -6.911, -6.89, -6.868, -6.836, -6.811, -6.793]
```

```
>>round(map(fc[2],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),3)
[-6.989, -6.968, -6.948, -6.911, -6.89, -6.867, -6.836, -6.81, -6.793]
```

En todos los casos (a pesar de haber empleado sólo 40 segmentos) los valores calculados con el método de Runge-Kutta son bastante próximos a soluciones conocidas, lo que una vez más corrobora la precisión del método.

Como segundo ejemplo se resolverá el siguiente sistema de 2 ecuaciones diferenciales calculando valores de las variables dependientes "x" y "y" para valores de "t" iguales a 0.2, 0.5, 0.7, 0.9, 1.2, 1.3.

$$\left.\begin{aligned} \frac{dx}{dt} &= x' = 2x + 3y \\ \frac{dy}{dt} &= y' = 2x + y \end{aligned}\right\} \quad x = 2.7\,;\ y = 2 \ \ para \ \ t = 0$$

Primero se reescribe el sistema en función de la variable "t" y el vector de variables "y":

$$\left.\begin{aligned} y'_0 &= f_0(t, y) = 2y_0 + 3y_1 \\ y'_1 &= f_1(t, y) = 2y_0 + y_1 \end{aligned}\right\} \quad y_0 = 2.7\,;\ y_1 = 2 \ \ para \ \ t = 0$$

Entonces se procede igual que en el anterior ejemplo, es decir primero se crea el vector de funciones:

```
>>f=[];
>>f[0]=function(t,y){return 2*y[0]+3*y[1];};
>>f[1]=function(t,y){return 2*y[0]+y[1];};
```

En este caso se emplearán 50 segmentos (n=50), el último valor de la variable independiente ("tn") es 1.3 porque ese es el último valor requerido:

```
>>t0=0; tn=1.3; y0=[2.7,2]; m=y0.length; n=50; h=(tn-t0)/n;
```

Los valores iniciales de las variables que forman parte del proceso son:

```
>>vt=[t0]; vy=[y0]; k=0; t=t0; y=copy(y0); k1=new Array(m-1); k2=new
Array(m-1); k3=new Array(m-1); k4=new Array(m-1);
```

Los 50 puntos (n=50) se calculan igual que el ejemplo anterior:

```
>>for(j=0;j<n;j++){for(i=0;i<m;i++) k1[i]=h*f[i](t,y); t_=t+h/2;
y_=add(y,div(k1,2)); for(i=0;i<m;i++) k2[i]=h*f[i](t_,y_);
y_=add(y,div(k2,2)); for(i=0;i<m;i++) k3[i]=h*f[i](t_,y_); t_=t+h;
y_=add(y,k3); for(i=0;i<m;i++) k4[i]=h*f[i](t_,y_); for(i=0;i<m;i++) y[i]
+=(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6; t=round(t+h,15); vy[++k]=copy(y);
vt[k]=t;};
```

Con los puntos (guardados en los vectores "vt" y "vy"), se generan las funciones de interpolación (en este caso empleando el método segmentario):

```
>>vy=transpose(vy);
>>fi=[];
>>fi[0]=geninseg(vt,vy[0]);
>>fi[1]=geninseg(vt,vy[1]);
```

Como ahora no se tiene la solución analítica, se generan otras funciones de interpolación para el doble de segmentos (n=100):

```
>>t0=0; tn=1.3; y0=[2.7,2]; m=y0.length; n=100; h=(tn-t0)/n;
>>vt=[t0]; vy=[y0]; k=0; t=t0; y=copy(y0); k1=new Array(m-1); k2=new
Array(m-1); k3=new Array(m-1); k4=new Array(m-1);
>>for(j=0;j<n;j++){for(i=0;i<m;i++) k1[i]=h*f[i](t,y); t_=t+h/2;
y_=add(y,div(k1,2)); for(i=0;i<m;i++) k2[i]=h*f[i](t_,y_);
```

```
y_=add(y,div(k2,2)); for(i=0;i<m;i++) k3[i]=h*f[i](t_,y_); t_=t+h;
y_=add(y,k3); for(i=0;i<m;i++) k4[i]=h*f[i](t_,y_); for(i=0;i<m;i++) y[i]
+=(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6; t=round(t+h,15); vy[++k]=copy(y);
vt[k]=t;};
>>vy=transpose(vy);
>>fi2=[];
>>fi2[0]=geninseg(vt,vy[0]);
>>fi2[1]=geninseg(vt,vy[1]);
```

Para determinar visualmente si los valores calculados son o no confiables se dibujan las gráficas de las dos funciones de interpolación:

```
>>plot([fi[0],fi2[0]],t0,tn)
```


```
>>plot([fi[1],fi2[1]],t0,tn)
```

Como se puede ver las dos funciones concuerdan a tal punto que se superponen completamente, por lo tanto se concluye que las funciones de interpolación generadas para 50 segmentos son confiables.

Los valores pedidos para la primera variable ($x=y_0$), redondeados al 4 dígito después del punto son:

```
>>round(map(fi[0],[0.2,0.5,0.7,0.9,1.2,1.3]),4)
[6.1778, 20.7643, 46.3142, 103.1579, 342.6202, 511.1526]
```

Y para la segunda ($y=y_1$):

```
>>round(map(fi[1],[0.2,0.5,0.7,0.9,1.2,1.3]),4)
[4.2823, 13.9642, 30.9754, 68.8532, 228.4737, 340.8229]
```

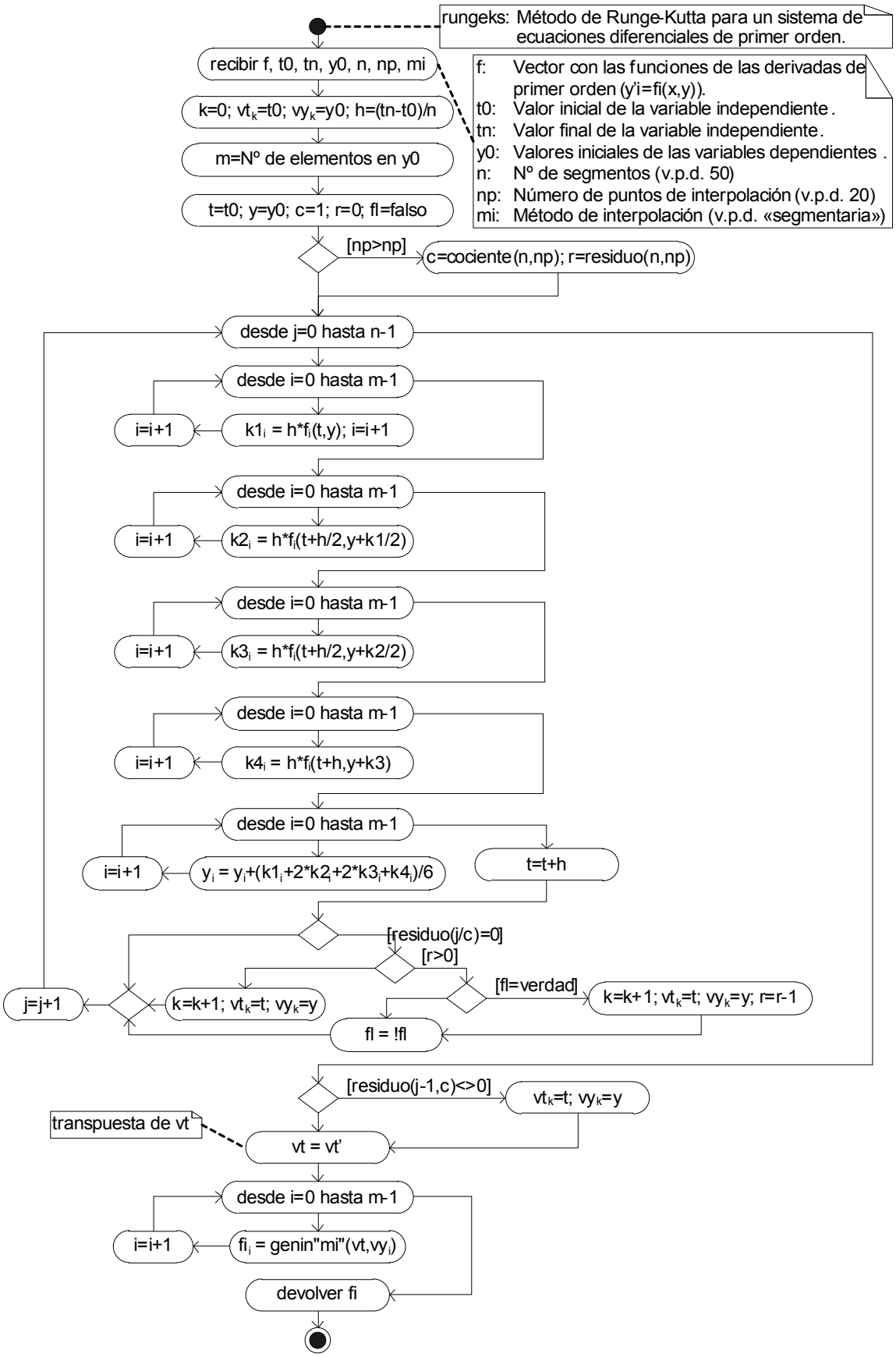
10.3.3.2. Algoritmo y código

El algoritmo para encontrar las soluciones de un sistema de ecuaciones diferenciales de primer orden, por el método de Runge-Kutta, se presenta en la siguiente página. El código elaborado en base al mismo y añadido a la librería "hpvlib.js", es:

```
function rungeks(f,t0,tn,y0,n,np,mi){
  if (n==null) n=50;
  if (np==null) np=20;
  if (mi==null) mi="segmentaria";
  mi.toLowerCase();
  var k=0,vt=[t0],vy=[y0],h=(tn-t0)/n,m=y0.length,t_,y_,fi=[];
  var t=t0,y=copy(y0),j,c=1,r=0,fl=false,k1=new Array(m-1);
  var k2=new Array(m-1), k3=new Array(m-1), k4=new Array(m-1);
  if (n>np) {c=quot(n,np); r=np;};
  for (j=0;j<n;j++){
    for(i=0;i<m;i++) k1[i]=h*f[i](t,y);
    t_=t+h/2; y_=add(y,div(k1,2));
    for(i=0;i<m;i++) k2[i]=h*f[i](t_,y_);
    y_=add(y,div(k2,2));
    for(i=0;i<m;i++) k3[i]=h*f[i](t_,y_);
    t_=t+h; y_=add(y,k3);
    for(i=0;i<m;i++) k4[i]=h*f[i](t_,y_);
    for(i=0;i<m;i++) y[i]+=(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6;
    t=round(t+h,15);
    if (j%c == 0)
      if (r>0) {
        if (fl) {vt[++k]=t; vy[k]=copy(y); r--};
        fl=!fl;}
      else
        {vt[++k]=t; vy[k]=copy(y);};
  }
  if (--j%c != 0) {vt[k]=t; vy[k]=copy(y);}
  vy=transpose(vy);
  switch(mi) {
    case "lagrange": for (i=0;i<m;i++) fi[i]=geninlag(vt,vy[i]); break;
    case "newton": for (i=0;i<m;i++) fi[i]=geninnew(vt,vy[i]); break;
    case "lineal": for (i=0;i<m;i++) fi[i]=geninlin(vt,vy[i]); break;
    default: for (i=0;i<m;i++) fi[i]=geninseg(vt,vy[i]);
  }
  return fi;
}
```

Haciendo correr el programa con los datos del primer ejemplo manual se obtienen los mismos resultados que en dicho ejemplo:

```
>>f=[];
>>f[0]=function(t,y){return y[1]-2*y[2]-sqr(t)+2*pow(t,1.5)+1.2*pow(t,0.2)-
10;};
>>f[1]=function(t,y){return -5*y[2]+5*pow(t,1.5)+2*t-35;};
>>f[2]=function(t,y){return 2*y[0]-y[1]+sqr(t)-2*pow(t,1.2)+1.5*sqrt(t)-
10;};
>>fi=rungeks(f,0,0.35,[3,-4,-7],40,40,"euler");
>>round(map(fi[0],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),3)
[3.027, 3.063, 3.094, 3.145, 3.171, 3.198, 3.236, 3.264, 3.284]
```


rungeks: Método de Runge-Kutta para un sistema de ecuaciones diferenciales de primer orden.
recibir f, t0, tn, y0, n, np, mi
f: Vector con las funciones de las derivadas de primer orden (y'i=fi(x,y)).
t0: Valor inicial de la variable independiente.
tn: Valor final de la variable independiente.
y0: Valores iniciales de las variables dependientes.
n: Nº de segmentos (v.p.d. 50)
np: Número de puntos de interpolación (v.p.d. 20)
mi: Método de interpolación (v.p.d. «segmentaria»)
k=0; vtk=t0; vyk=y0; h=(tn-t0)/n
m=Nº de elementos en y0
t=t0; y=y0; c=1; r=0; fl=falso
[np>np]
c=cociente(n,np); r=residuo(n,np)
desde j=0 hasta n-1
desde i=0 hasta m-1
i=i+1
k1i = h*fi(t,y); i=i+1
desde i=0 hasta m-1
i=i+1
k2i = h*fi(t+h/2,y+k1/2)
desde i=0 hasta m-1
i=i+1
k3i = h*fi(t+h/2,y+k2/2)
desde i=0 hasta m-1
i=i+1
k4i = h*fi(t+h,y+k3)
desde i=0 hasta m-1
i=i+1
yi = yi+(k1i+2*k2i+2*k3i+k4i)/6
t=t+h
[residuo(j/c)=0]
[r>0]
[fl=verdad]
j=j+1
k=k+1; vtk=t; vyk=y
k=k+1; vtk=t; vyk=y; r=r-1
fl = !fl
[residuo(j-1,c)<>0]
vtk=t; vyk=y
transpuesta de vt
vt = vt'
desde i=0 hasta m-1
i=i+1
fi = genin"mi"(vt,vyi)
devolver fi

```
>>round(map(fi[1],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),3)
[-3.997, -3.99, -3.98, -3.96, -3.947, -3.932, -3.91, -3.891, -3.877]
>>round(map(fi[2],[0.05,0.1,0.14,0.20,0.23,0.26,0.30,0.33,0.35]),3)
[-6.989, -6.968, -6.948, -6.911, -6.89, -6.868, -6.836, -6.811, -6.793]
```

Igualmente, se obtienen los mismos resultados con los datos del segundo ejemplo manual:

```
>>f[0]=function(t,y){return 2*y[0]+3*y[1];};
>>f[1]=function(t,y){return 2*y[0]+y[1];};
>>fi=rungeks(f,0,1.3,[2.7,2],50,50);
>>round(map(fi[0],[0.2,0.5,0.7,0.9,1.2,1.3]),4)
[6.1778, 20.7643, 46.3142, 103.1579, 342.6202, 511.1526]
>>round(map(fi[1],[0.2,0.5,0.7,0.9,1.2,1.3]),4)
[4.2823, 13.9642, 30.9754, 68.8532, 228.4737, 340.8229]
```

10.4. EJERCICIOS

Como en los anteriores temas para presentar los ejercicios que resuelva en la calculadora, copie las instrucciones en un editor de texto y luego, al momento de presentar el trabajo, vuelva a copiarlos y ejecutarlos en la calculadora. Alternativamente, puede guardar el ejercicio resuelto como una página HTML, haciendo clic derecho en página y eligiendo la opción "guardar como" (o su equivalente) y guardando la página como página web completa (o fichero único) asignándole un nombre adecuado como ejercicio1.html.

1. Encuentre los valores de "y" para los valores de "x" dados, resolviendo la siguiente ecuación diferencial con el método de Euler. Luego dibuje la función de interpolación resultante:

$$y' = y\left(x^2 - 1\right) \quad \begin{cases} y = 1 & para \quad x = 0 \\ y = ? & para \quad x = 0.3, 0.6, 0.8, 1.0, 1.3, 1.5, 1.7, 2 \end{cases}$$

2. Repita el ejercicio anterior empleando el método de Runge-Kutta.

3. Encuentre los valores de "y" para los valores de "x" dados, resolviendo la siguiente ecuación diferencial con el método de Euler. Luego dibuje la función de interpolación resultante (emplee el método de Newton para la interpolación):

$$y' = \frac{5x}{y} - x\,y \quad \begin{cases} y = 2 & para \quad x = 0 \\ y = ? & para \quad x = 0.1, 0.2, 0.3, 0.4, 0.5 \end{cases}$$

4. Repita el ejercicio anterior empleando el método de Runge-Kutta.

5. Encuentre los valores de "y" para los valores de "x" dados, resolviendo la siguiente ecuación diferencial con el método de Euler. Luego dibuje la función de interpolación resultante (emplee el método de Lagrange para la interpolación):

$$y' = x^3 + y^2 \quad \begin{cases} y = 2 & para \quad x = 0 \\ y = ? & para \quad x = 0.2, 0.5, 0.7, 1.0, 1.2, 1.3, 1.4 \end{cases}$$

6. Repita el ejercicio anterior empleando el método de Runge-Kutta.

7. Encuentre los valores de "y" para los valores de "t" dados, resolviendo la siguiente ecuación diferencial con el método de Euler. Luego dibuje

la función de interpolación resultante (emplee el método segmentario para la interpolación):

$$y' = y^2 + t^2 \quad \begin{cases} y = 2 & para \quad t = 1 \\ y = ? & para \quad t = 1.3, 1.6, 1,8\,, 2.0 \end{cases}$$

8. Repita el ejercicio anterior empleando el método de Runge-Kutta.

9. Encuentre los valores de "q" para los valores de "t" dados, resolviendo la siguiente ecuación diferencial con el método de Euler. Luego dibuje la función de interpolación resultante (emplee el método de Lagrange para la interpolación):

$$\frac{1}{2}q'' + 6q' + 50\,q = 24\sin(10\,t) \quad \begin{cases} q = 0 & para \quad t = 0 \\ q = ? & para \quad t = 0.4, 0.6, 0.8, 1.2, 1.4, 1.6, 1.9, 2.1 \end{cases}$$

10. Repita el ejercicio anterior empleando el método de Runge-Kutta.

11. Encuentre los valores de "x" para los valores de "t" dados, resolviendo la siguiente ecuación diferencial con el método de Euler. Luego dibuje la función de interpolación resultante (emplee el método de Newton para la interpolación):

$$x'' + 64\,x = 16\cos(8\,t) \quad \begin{cases} x = 0 & para \quad t = 0 \\ x = ? & para \quad t = 0.1, 0.2, 0.5, 0.6, 0.7, 0.8 \end{cases}$$

12. Repita el ejercicio anterior empleando el método de Runge-Kutta.

13. Encuentre los valores de "x" y "y" para los valores de "t" dados, resolviendo el siguiente sistema de ecuaciones diferenciales con el método de Euler. Luego dibuje las funciones de interpolación resultantes (emplee el método de Lagrange para la interpolación):

$$\begin{aligned} x' &= x\,y + t \\ y' &= x - t \end{aligned} \quad \begin{cases} x = 0, y = 1 & para \quad t = 0 \\ x\,, y = ? & para \quad t = 0.1, 0.2, 0.4, 0.6, 0.7, 0.8 \end{cases}$$

14. Repita el ejercicio anterior empleando el método de Runge-Kutta.

15. Encuentre los valores de "x" y "y" para los valores de "t" dados, resolviendo el siguiente sistema de ecuaciones diferenciales con el método de Euler. Luego dibuje las funciones de interpolación resultantes (emplee el método segmentario para la interpolación):

$$\begin{aligned} u' - 2v + 2 + t^2 - 2t^{1.5} + 1.2\,t^{0.2} + 7 = 0 \\ v' + 3w - 4t^{1.5} - 2t + 25 = 0 \\ w' - 3u + 2v - t^2 + t^{1.2} - 1.5\,t^{0.5} + 8 = 0 \end{aligned}$$

$$\begin{cases} u = 3, v = -4, w = -7 & para \quad t = 0.3 \\ u\,, v\,, w\,? & para \quad t = 0.33, 0.35, 0.4, 0.42, 0.45, 0.47, 0.5 \end{cases}$$

16. Repita el ejercicio anterior empleando el método de Runge-Kutta.