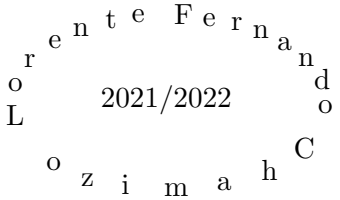
Prácticas de Cálculo Numérico I en la UAM

Fernando Chamizo Lorente

Curso 2021/22



Prefacio

Estas notas corresponden a las prácticas de la asignatura de Cálculo Numérico I tal como se impartió en el curso 2021/22 a los alumnos del doble grado Informática-Matemáticas. Van acompañadas de un apéndice que contiene posibles soluciones de todas las actividades planteadas. Además, todos los programas se pueden descargar de `http://matematicas.uam.es/~fernando.chamizo/asignaturas/2122cni/2122cni.html` como un fichero de texto.

Madrid 30 de julio de 2022

Fernando Chamizo

Índice general

Capítulo 1

Introducción a `matlab/octave`

1.1. Instalación y primer uso de `matlab/octave`

Entre el *software* empleado en cálculo numérico con fines académicos, sobre todo dentro del orientado a ingeniería, el que ha alcanzado una mayor difusión es `matlab` cuyo nombre abrevia *matrix* y *laboratory*. El nombre da una idea de que, al menos originariamente (ya tiene casi 40 años), estaba orientado a cálculos con matrices. Actualmente su ámbito es más amplio, especialmente cuando se instalan paquetes adicionales, sin embargo las matrices se siguen reflejando en la propia sintaxis y en mi experiencia (quizá sesgada) son los cálculos con ellas lo que mejor hace.

A pesar de que internamente esos cálculos se llevan a cabo con una adaptación de rutinas que hoy son de dominio público, `matlab` es *software* comercial propietario. Un punto a favor es que las licencias para estudiantes no tienen un precio astronómico y muchas universidades, como la UAM, tienen licencias generales. De todas formas, una alternativa libre y gratuita es (GNU) `octave`. Los comandos y programas básicos son intercambiables hasta el punto de que fuera de un ámbito relativamente avanzado, te parecerá que `octave` es un emulador perfecto de `matlab` (salvo alguna cuestión fina en las representaciones gráficas). En particular, es de prever que todo lo que veamos en este curso no establecerá ninguna diferencia. Más adelante habrá algunos comentarios dedicados `octave`, a pesar de que con seguridad la mayor parte de vosotros utilizaréis `matlab`. Un uso más avanzado muestra ventajas a favor de `matlab`, sobre todo con el uso de librerías, llamadas *toolboxes*, con propósitos especiales.

Para instalar `matlab` necesitamos una licencia. La UAM tiene una de campus disponible para todos los estudiantes. Las instrucciones para emplearla, por supuesto teniendo una dirección de correo institucional, están descritas siguiendo el enlace correspondiente en:

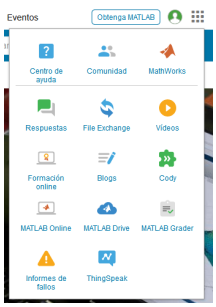
`https://faq.uam.es/index.php?action=show&cat=38`

Es bastante posible que algunos no deseéis ocupar disco duro en vuestro ordenador con programas grandes si hay alternativa. En este caso sí la hay, y es emplear `matlab` *online*. Siendo las clases presenciales y en el laboratorio, no haré uso remoto de `matlab` más allá de alguna demostración. Por cierto, el rendimiento de `matlab` *online* es muy bueno, por tanto resulta una alternativa conveniente si tu ordenador se acerca a la jubilación.

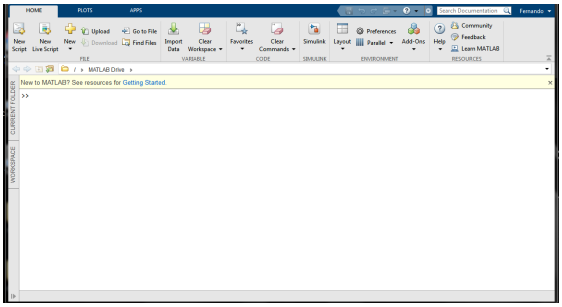
En realidad dispones de dos formas de usar `matlab` a través de la red. Una es con el servidor jupyter del departamento de matemáticas (`https://jupyter.mat.uam.es/hub/login`), pero esta no es la mejor idea porque requiere que sepas algo de jupyter y mi experiencia es que hay algunos problemas (sobre todo con los gráficos). Más conveniente es utilizar la web oficial de `matlab`. Una vez registrados, accederemos a través de

`https://matlab.mathworks.com/`

y después de entrar en nuestra cuenta, en el menú de la derecha, elegimos MATLAB Online que nos lleva al interfaz principal.



MATLAB Online



Aspecto inicial de la interfaz

Una posibilidad un poco más directa es acceder con `https://www.mathworks.com/products/matlab-online.html` y pulsar *Start using MATLAB Online*.

Si deseamos instalar `octave`, la página del proyecto es:

`https://www.gnu.org/software/octave/`

Allí están los instaladores para diferentes sistemas operativos. La instalación básica es más ligera que la de de `matlab`, la última versión para Windows ocupa 1.8 GB en el disco duro. Al parecer, `octave` también se puede usar *online* en `https://octave-online.net/`, pero no tengo experiencia al respecto y algún intento de prueba me ha hecho desistir.

Al ejecutar `octave` obtenemos una interfaz parecida a la de `matlab`, pero más espartana. Insisto, no obstante, que es difícil notar diferencias moviéndose en un nivel básico.

1.2. Uso como una calculadora avanzada

El *prompt* está pidiendo que escribamos algo y ahí podemos teclear operaciones como en una calculadora. Ni que decir tiene que el producto se indica con `*` y, por otro lado, las potencias se indican con `^`, como en `sagemath` de la asignatura de Laboratorio. Por ejemplo, tecleando `1+1` y pulsando *enter*, obtendremos nuestra primera sesión:

```
>> 1+1
ans = 2
```

Aquí `ans` es la abreviatura de *answer* (para decir toda la verdad, en realidad en la versión *online* la respuesta sale menos compacta, ocupando varias líneas). Por supuesto, existen todas las funciones que habitualmente pueblan las teclas de nuestras calculadoras y que incorporan los principales lenguajes de programación. Sin ánimo de ser exhaustivo, se tienen `sqrt` (raíz cuadrada), `exp`, `log` (logaritmo neperiano), las funciones trigonométricas `sin`, `cos`, `tan` y sus inversas `asin`, `acos`, `atan`. Además tenemos el número π denotado por `pi`. Por ejemplo:

```
>> ( log(pi) + exp(1) )*( (sin(1))^2 + sqrt(6) )
ans = 12.1977
```

Los espacios son irrelevantes, solo se han introducido para dar legibilidad al código.

Si te sientes defraudado por este ejemplo, ya que tu calculadora te da mayor precisión, es el momento de aprender que hay varios formatos posibles y los dos más usados son `format short` y `format long`. Por defecto, estamos en el primero que da solo precisión simple. Si tecleamos `format long` y después nuestro ejemplo anterior, el resultado pasa a ser `12.197703479831025` que posiblemente mejore lo que muestra en la pantalla tu calculadora de bolsillo. Estos dos formatos admiten algunas variantes que no examinaremos aquí. Solo mencionaré `format rat` que aproxima por una fracción y en el caso anterior daría `5306/435`.

Una constante que no está en las calculadoras más básicas es `i`, la unidad imaginaria. Sin nada especial, podemos hacer operaciones con números imaginarios, incluso con funciones trascendentes. Por ejemplo, la famosísima fórmula de Euler $e^{i\pi} + 1 = 0$, es coherente con que `exp(i*pi)` resulte `-1.0000+0.000i` y con que `log(-1)`, que no nos dejaban hacer en secundaria, resulte `0.0000+3.1416i`. Por cierto, dependiendo de la versión de `matlab/octave` puede que la parte imaginaria de `exp(i*pi)` sea un número del orden de 10^{-16} en vez de ver `0.000`. De hecho, en la versión actual de `matlab` (*online*), el resultado antes mencionado contrasta con que `exp(pi*i) + 1` dé `0.0000e+00 + 1.2246e-16i` (recuerda que la notación científica *a*e*b* abrevia $a \cdot 10^b$). Seguro que estás sospechando, acertadamente, que esto se debe a que `matlab` (a diferencia de `sagemath`) no está haciendo cálcu-

lo simbólico. Hay cierto error interno con números que no son "exactos". La falta de precisión se cuantifica con el llamado *épsilon máquina*. Este es el menor error relativo que es posible distinguir. En `matlab` el épsilon máquina se obtiene con el comando `eps`. Si lo tecleamos, obtendremos `2.2204e-16`. Parece un número que no tiene nada de particular, pero si calculas `2^-52` quizá cambies de opinión. La inmensa mayoría de los instrumentos digitales de cálculo que hayas utilizado en tu vida están afectados exactamente por este épsilon máquina. La razón es que el formato empleado habitualmente para almacenar números con 64 bits no permite un error relativo menor.

Si todo esto del error relativo más pequeño te resulta incomprensible, el siguiente ejemplo te iluminará:

```
>> (0.1+eps)-0.1
ans = 2.2204e-16

>> (10+eps)-10
ans = 0
```

La moraleja es que aunque nuestra calculadora u ordenador sea capaz de tratar con números tan pequeños como 10^{-100} y distinguirlos de cero, eso no significa que sea capaz de discriminar cualquier par de números que difieran en 10^{-100}. El límite real es del orden de 10^{-16} y este problema es bastante universal, no una manía de `matlab`.

Volviendo a la comparación con la calculadora, algo en lo que `matlab` va más allá es que está permitido el uso de variables como en cualquier lenguaje de programación típico. Además, al escribir la variable recuperamos su valor. Por ejemplo, digamos que queremos saber cuántos minutos tuvo 2020, que fue bisiesto. Para ello creamos las variables `d` de días, `hd` de horas por día y `mh` de minutos por hora. La cuenta sería

```
>> d = 366
d =  366
>> hd = 24
hd =  24
>> mh = 60
mh =  60
>> d*hd*mh
ans =  527040
```

Es un poco pesado ver el mismo resultado que tecleamos. Una solución, que a la hora de programar será casi necesidad, es añadir al final de la línea un punto y coma, lo cual omite la salida. Es decir, teclearíamos `d = 366;` etc. excepto en la última línea de la que queremos ver el resultado. Digamos que ahora queremos hacer el cálculo para 2022, entonces `d` se reduce en uno porque no es bisiesto y podríamos proceder con

```
>> d = d - 1;
>> d*hd*mh
ans =  525600
```

Si pusiéramos un punto coma también tras `d*hd*mh`, la salida sería vacía aunque, naturalmente, la cuenta se sigue haciendo de manera interna.

1.3. Organizando los cálculos en ficheros

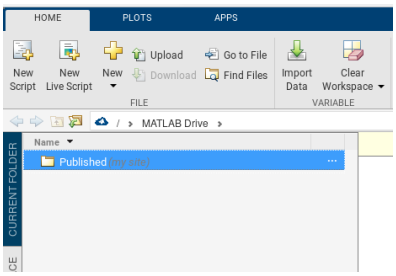
En cuanto queramos hacer algo más que un cálculo inmediato, es mejor utilizar un fichero en vez de teclear separadamente cada cálculo. Esto nos permitirá hacer modificaciones de manera más eficiente y se convertirá en una necesidad para programar.

Los ficheros con instrucciones matlab tienen la extensión `.m` y esencialmente son de dos tipos: *scripts* y funciones. En realidad, la terminología varía según los autores. En un contexto informático general, los primeros son lo que llamaríamos programas y las segundas, subrutinas a las que llaman los programas.

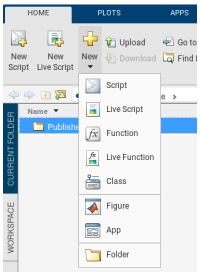
Por ahora nos centramos en los *scripts*. Imaginemos que queremos crear uno llamado `ex01.m`. Distingamos dos situaciones, en la primera teniendo `matlab/octave` instalado y en la segunda con `matlab` *online*.

En el primer caso, las versiones modernas de `matlab/octave` llevan incorporado un editor, pero si nos gusta más el nuestro (esto se aplica sobre todo a los maniáticos de Linux), nada impide que usemos el que queramos (que es lo que hago yo, sobre todo con `octave`). Así que basta crear con nuestro editor o con el editor incorporado, un fichero de texto `ex01.m` de la manera habitual (posiblemente File, New,...). Para que `matlab/octave` tenga acceso al fichero debe estar en el `path` correspondiente. Existen los comandos de Linux `cd`, `ls`, `pwd` que nos permitirán saber dónde estamos y movernos donde esté el fichero. También es posible hacerlo con la interfaz. Soy consciente de que la explicación es un poco críptica, pero al buen entendedor... y prefiero centrarme en el la segunda posibilidad, que posiblemente sea la mayoritaria.

Veamos ahora con todo detalle cómo crear un *script* en `matlab` *online*, que quedará almacenado en la nube. En `matlab` instalado es similar. De hecho, en la práctica, más fácil porque nuestro ordenador nos es más familiar que la nube. Para hacerlo un poco más completo vamos a meterlo dentro de un directorio llamado `week01`. Si desplegamos la pestaña `Current folder` veremos que `MATLAB Drive` solo contiene una carpeta llamada `Published`. Para crear `week01` pulsamos en `New` y en el desplegable escogemos `Folder`. Nos pedirá el nombre, `week01` en nuestro caso.

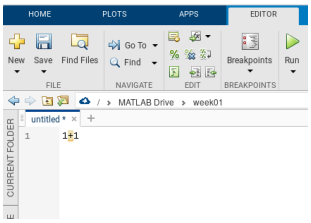


Pestaña Current folder

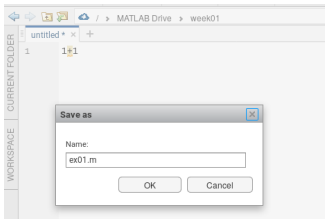


Desplegable de New

Con ello, en `Current folder` ya tenemos el nuevo directorio y nos metemos en él simplemente pinchando encima. Notemos que en la línea bajo el menú gráfico, `>MATLAB Drive` se ha completado con un `>week01` a la derecha. Podemos usar esa línea con el ratón para navegar entre los directorios. O bien pulsando de nuevo `New`, pero ahora escogiendo `script`, o bien haciendo lo mismo pulsando el botón derecho del ratón dentro de la pestaña `Current folder`, creamos un *script*. De la segunda forma podremos ajustar directamente el nombre, de la primera, cuando pulsemos `Save` nos pedirá un nuevo nombre (por defecto es `untitled.m`).



Nuevo script



Grabar el nuevo script

En las imágenes, `ex01.m` tiene un contenido bastante anodino, solo `1+1`. Si lo ejecutamos pulsando el botón `Run` (el triángulo verde) o escribiendo en la ventana de comandos `ex01` (sin el `.m`), obtendremos la consabida respuesta `ans = 2`.

Iremos viendo a lo largo del curso muchos comandos. Por ahora, juguemos a hacer nuestro uso tipo calculadora, pero con más comodidad. Los comentarios se escriben precediendo las líneas con `%`.

```
1  % Aproximando el número e
2  n = 50;
3  (1+1/n)^n
4  % Con un n doble del anterior
5  n = 2*n;
6  (1+1/n)^n
7  % sin(1) y su aproximación de Taylor de orden 7
8  sin(1)
9  1-1/factorial(3)+1/factorial(5)-1/factorial(7)
10 % Diferencia de sin(1) y Taylor de orden 7
11 sin(1) - (1-1/factorial(3)+1/factorial(5)-1/factorial(7) )
```

Los puntos y comas 2 y 5 impiden que se vean los valores 50 y 100 de `n`. Por cierto, indico los números de línea para hacer referencia a ellos y facilitar la copia, por supuesto no hay que teclearlos.

1.4. Matrices, la gran fortaleza de `matlab/octave`

Cada lenguaje de programación tiene sus peculiaridades: en C los punteros, en C++ y Java las clases y en python las listas. Como indica su nombre, en `matlab` la estructura de datos estrella son las matrices. En muchas ocasiones, sustituirán a los bucles que utilizaríamos en los lenguajes mencionados. La manera de indicar una matriz es limitándola con corchetes e indicando

los cambios de fila con punto y coma. Los elementos de una fila se pueden separar con comas, pero esto no es obligatorio, con espacios es suficiente. La suma y el producto de matrices no requieren comandos especiales, bastan `+` y `*`. La traspuesta se indica con el apóstrofo. Para ser del todo rigurosos, lo que se consigue con él es la traspuesta conjugada, pero para matrices reales no hay diferencia. La matriz nula `m` $\times$ `n` se indica con `zeros(m,n)` y la matriz identidad de dimensión `n` mediante `eye(n)`. También existe `ones(m,n)` que general una matriz `m` $\times$ `n` con todos sus elementos iguales a uno.

Como ejemplo, escribamos en un *script* algunas operaciones con las matrices

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 2 & -1 & 1 \end{pmatrix} \qquad \text{y} \qquad B = \begin{pmatrix} -2 & 1 & 0 \\ 3 & 1 & 1 \end{pmatrix}$$

que también involucren matrices especiales.

```
% Definiendo dos matrices
A = [1, 0, 1; 2, -1, 1];
B = [-2, 1, 0; 3, 1, 1];
% Calcula la suma de matrices
A + B
% La traspuesta de B es
B'
% El producto de A por la traspuesta de B es
A*B'
% Esto da A
A + zeros(2,3)
% y esto vuelve a dar A*B'
A*B'*eye(2)
```

Un comentario para los que uséis `octave` en vez de `matlab` *online*, es que cuando la salida ocupa más de una "página" debemos pasar la salida línea a línea. Para evitar esto, basta teclear en la ventana de comandos `more off`. Si queremos restablecer este comportamiento, usaremos `more on`.

El elemento en la fila `m` y columna `n` de una matriz `A` se indica con `A(m,n)`. Así, en el ejemplo anterior `A(2,1)` resultaría 2 y `B(1,1)` sería -2. A diferencia de lo que ocurre en `sagemath` y en los principales lenguajes de programación, en `matlab/octave` los índices no empiezan en cero sino en uno.

Las matrices admiten ser construidas por bloques conservando la misma sintaxis que tendrían si los bloques fueran números. Por ejemplo, en física la matriz de Dirac γ_2 es una matriz que se construye a partir de la matriz de Pauli σ_2 por medio de

$$\gamma_2 = \begin{pmatrix} O & -\sigma_2 \\ \sigma_2 & O \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & -i & 0 \\ 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix} \qquad \text{con} \qquad \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}.$$

Esta matriz es unitaria, es decir, al multiplicarla por su traspuesta conjugada da la identidad. Escribamos su construcción y la comprobación de que es unitaria en un *script*.

```
1  % Matriz de Pauli
2  s_2 = [0 -i; i 0]
3  % Matriz de Dirac
4  g_2 = [zeros(2) -s_2; s_2 zeros(2) ]
5  % Comprobamos que es unitaria
6  g_2*g_2'
```

La construcción por bloques se produce en la línea 4 y de paso aprendemos que `zeros(n)` es una abreviatura de `zeros(n,n)` (lo mismo funciona para `ones`). La matriz γ_2 es simétrica, por tanto la traspuesta conjugada es lo mismo que la conjugada. La conjugación se indica mediante `conj`, entonces sustituyendo la línea 6 por `g_2*conj(g_2)`, seguimos obteniendo la identidad.

Los vectores (fila o columna) no son otra cosa que matrices que tienen una de sus dimensiones iguales a uno. Sin embargo hay dos constructores de vectores fila que merecen una consideración especial.

Generalizando ligeramente lo que ocurre en python, con `[`a`:`b`]`, o simplemente con a`:`b, obtenemos un vector fila cuyos elementos son a, $a+1$, etc. hasta el último valor que no supera a b. De esta forma, $2:4$ es el vector `[2, 3, 4]` y $1.7:4$ es `[2.7, 3.7]`. Si necesitamos un incremento diferente de la unidad, basta indicarlo entre los dos puntos. Así podemos generar cualquier vector cuyos elementos estén en progresión aritmética. Por ejemplo:

```
[2:3:15]      →   [2, 5, 8, 11, 14]
```

donde los corchetes son superfluos.

Necesitaremos, a veces, definir estos vectores con elementos en progresión aritmética especificando el origen, el final y el número de elementos. Por supuesto lo podemos conseguir con los dos puntos haciendo un pequeño cálculo aparte, pero es más directo emplear el comando `linspace(`a`,`b`,`n`)` donde n es el número de elementos. Por ejemplo, `linspace(2,5,8)` resultaría

```
2.0000 2.4286 2.8571 3.2857 3.7143 4.1429 4.5714 5.0000
```

(así, sin corchetes ni comas es como aparece en la salida). El incremento es de $3/7$ y por tanto equivale a `2:3/7:5`.

Por cierto, la longitud de un vector se obtiene con el comando `length` y las dimensiones de una matriz con `size`.

La notación de los dos puntos sirve también para establecer rangos de índices de vectores o matrices. Veámoslo con un ejemplo.

```
1  % Una matriz y un vector fila
2  A = [1, 0, 1; 2, -1, 1];
3  v = [2, 3, 5, 7];
4  % La matriz formada por las dos últimas columnas
5  A(1:2,2:3)
6  % Una alternativa
7  A(:,2:3)
8  % La sección central de v
9  v(2:3)
10 % Muestra las dimensiones de A y v
```

```
size(A)
size(v)
length(v)
% Anulamos las dos últimas columnas de A
A(:,2:3) = zeros(2)
```

De hecho la última instrucción se podría escribir como `A(:,2:3) = 0` porque, cuando no hay ambigüedad, los números se identifican con matrices de las dimensiones adecuadas. Por ejemplo, `12+A` es válido y `12` se identifica con la matriz 2×3 que tiene todas sus elementos iguales a 12. Por si sirve para que los matemáticos más estrictos no despotriquen solo de los informáticos, un convenio parecido es común en algunas partes de la física.

Veamos otro ejemplo para practicar. Se sabe que

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \qquad \text{para} \quad n \geq 1,$$

donde F_n son los números de Fibonacci. Utilizando esta fórmula, vamos a hacer un programilla, poco eficiente, que dado n nos diga cuánto vale F_n. Utilizaremos los comandos de entrada/salida `input`, que pide un dato al usuario y `disp` que muestra el valor de una variable sin que aparezca `ans =`. En una sección posterior volveremos sobre los comandos de entrada/salida.

```
% Pedimos la n
n = input('Introduce n');
% Elevamos a n
F = [1,1;1,0]^n;
% El resultado es el elemento 1,2 (o el 2,1)
disp(F(1,2))
```

Como cabe esperar, elevar a un n natural una matriz en `matlab` es la abreviatura de multiplicarla por sí misma n veces. Esto funciona incluso con números enteros si la matriz es invertible.

Las funciones trascendentes se aplican elemento a elemento a una matriz. De esta forma `exp(A)` no tiene nada que ver con el e^A que te explica(rá)n en los cursos de ecuaciones diferenciales sino que es algo tan prosaico como calcular e elevado a cada elemento. De esta forma

```
exp([0,1]) → 1.0000 2.7183,     sin(ones(2)) → 0.84147  0.84147
                                               0.84147  0.84147
```

donde, recuérdese, que `ones(2)` abrevia `ones(2,2)`.

A veces nos gustaría tener un producto de matrices, o potencias naturales, término a término. En principio parece un antojo raro, pero en breve veremos que es fundamental por ejemplo a la hora de representar gráficas. La manera de indicar que `*` o `^` se deben hacer término a término es precediéndolos de un punto. Por ejemplo, `[1:10].^2` es un vector fila con los 10 primeros cuadrados y `[1,2].*[6,3]` resulta `[6,6]`. Con la multiplicación usual `*` no tiene sentido multiplicar dos matrices 2×3, pero sí lo tiene

empleando `.*` para obtener otra matriz de las mismas dimensiones. Análogamente, `./` define una división término a término sin problemas siempre que los denominadores no se anulen.

1.5. Gráficas en dos dimensiones

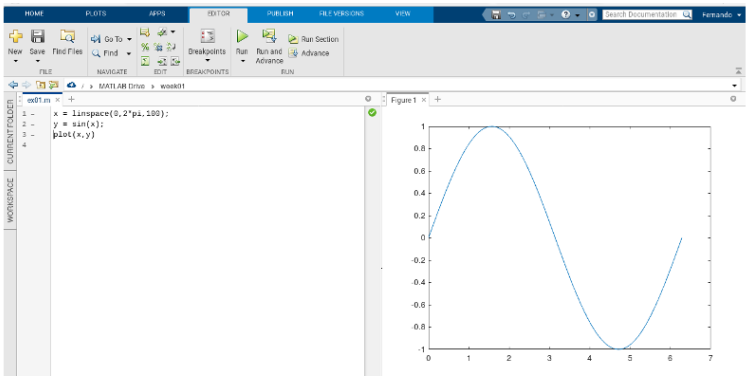
Posiblemente parte del éxito inicial de `matlab` en el mundo académico se debía a que, en comparación con el poco *software* disponible entonces, hacía gráficos bonitos con poco esfuerzo.

El comando básico para hacer gráficas bidimensionales en `plot`. Su estructura más simple es `plot(`a`,`b`)` donde a y b son dos vectores de la misma longitud. Lo que hace es emparejar los elementos respectivos de a y b para formar puntos y después los une.

Si queremos dibujar la gráfica de $y = \sin(x)$ en el intervalo $x \in [0, 2\pi]$ debemos crear un vector que tenga muchos valores de x en este intervalo y construir consecuentemente y. Lo conseguimos con el *script*

```
1  x = linspace(0,2*pi,100);
2  y = sin(x);
3  plot(x,y)
```

Al ejecutarlo en `matlab` *online* veremos que aparece una figura a la derecha, con la etiqueta `Figure 1`, En `octave` y versiones instaladas antiguas de `matlab` la figura sale fuera de la interfaz (como una ventana *pop-up*).



El script y la figura

Si cambiamos 100 en la primera línea por un valor mucho más pequeño, apreciaremos la discretización.

Imaginemos que ahora queremos dibujar $y = e^{-x}\sin(x)$ en el mismo intervalo. Un error de principiante sería escribir en la segunda línea la expresión `exp(-x)*sin(x)`. Esto es incorrecto porque `exp(-x)` y `sin(x)` son vectores fila de 100 elementos, esto es, matrices 1×100 y es imposible multiplicarlas con `*`, lo correcto es `exp(-x).*sin(x)`.

Si ahora pensamos en cómo representar $y = (1+x)^{-1}\sin(x)$, veremos la razón de ser del convenio de interpretar los números como matrices de las

dimensiones adecuadas. Gracias a él, no hay que convertir `1` en un vector de unos para sumárselo a `x`. Dos maneras naturales válidas de introducir esta función en la segunda línea son:

`(1+x).^-1.*sin(x)` y `sin(x)./(1+x)`

A continuación de los vectores que sirven de argumento a `plot`, es posible incluir una cadena de caracteres, limitada por apóstrofos o comillas y separada mediante una coma, con información sobre el color, el estilo de la línea y el símbolo usado para los puntos, según las tablas:

Color →

cyan	magenta	yellow	red	green	blue	white	black
`c`	`m`	`y`	`r`	`g`	`b`	`w`	`k`

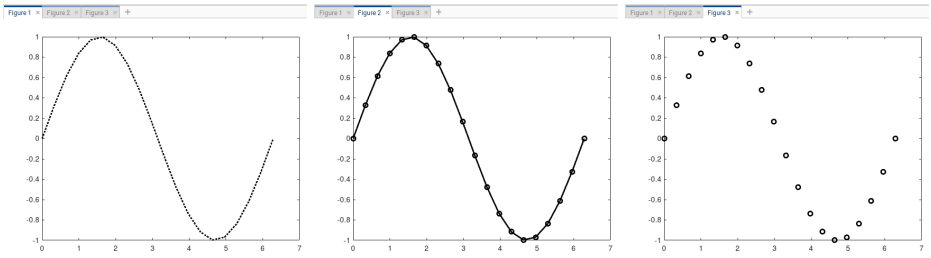
Estilo de línea →

solid	dashed	dotted	dash-dot
`-`	`--`	`:`	`-.`

Punto →

plus	circle	asterisk	x	square	diamond
`+`	`o`	`*`	`x`	`s`	`d`

Por ejemplo, con `plot(x,y,'k:')` obtendremos la gráfica con línea de puntos en negro, mientras que con `'k-o'` la línea será sólida conectando puntos marcados con un círculo y con `'ko'` la línea desaparecería. Si comprobáramos estos ejemplos con nuestro código inicial, los puntos estarían demasiado juntos y se superpondrían. Veamos, en su lugar, los resultados cambiando el vector en la primera línea por `linspace(0,2*pi,20)`.



`plot(x,y,'k:')` `plot(x,y,'k-o')` `plot(x,y,'ko')`

Las pestañas que se han dejado encima de las figuras dan que pensar que estas figuras no fueron creadas modificando cada vez el `plot`. En realidad se utilizó solo el código común:

```
1  x = linspace(0,2*pi, 20);
2  y = sin(x);
3  figure(1)
4  plot(x,y, 'k:', 'linewidth',2)
5  figure(2)
6  plot(x,y, 'k-o', 'linewidth',2)
7  figure(3)
8  plot(x,y, 'ko', 'linewidth',2)
```

Si precedemos una gráfica con `figure(`n`)` los situará en la pestaña n-ésima (o en la venta n-ésima en `octave` o versiones instaladas antiguas de

matlab). Por otro lado, 'linewidth',2 (o si se prefiere, 'LineWidth',2) indica que el grosor de la línea pasa a ser 2. El grosor por defecto es 0.5 y la razón de aumentarlo es para favorecer aquí la visibilidad de las figuras.

Las imágenes anteriores son simples capturas de pantalla, pero podemos guardar los ficheros directamente. Pinchando sobre cada figura pasaremos al menú FIGURE y uno de sus botones es Save (también hay un menú que aparece sobre las figuras al mover el ratón). Si usamos matlab *online*, con él podemos guardarlo en la nube y después descargarlo de allí o bien entrando en nuestra cuenta y eligiendo Matlab Drive, en vez de Matlab Online, o bien sin salirnos de la interfaz buscando el fichero en Current Folder, seleccionándolo y usando en el menú Home el botón Download (la flecha hacia abajo). Más adelante veremos cómo hacer esto también por *software*.

Las figuras admiten rótulos con xlabel, ylabel y title cuyos nombres se explican a sí mismos y cuyo uso queda claro con la gráfica inicial:

```
x = linspace(0,2*pi, 100);
y = sin(x);
plot(x,y)
xlabel('Eje X')
ylabel('Eje Y')
title('Mi primera gráfica')
```

Hay algunas consideraciones que destacar relativas a la superposición de gráficas. La primera es que usar un plot y después otro no produce superposición, sino que cada uno borra el resultado anterior. Para superponer gráficas una posibilidad es repetir los argumentos de plot. Por ejemplo,

```
x = linspace(0,2*pi, 100);
y = sin(x);
y2 = cos(x);
plot(x,y,'--',x,y2,':')
```

muestra en la misma figura la gráfica del seno en línea discontinua y la del coseno en línea de puntos. Nótese que automáticamente se asignan colores diferentes. Podemos cambiar este comportamiento especificando un color. Añadiendo legend(*texto*1,*texto*2) se mostrará la leyenda: un cuadro que explica cada gráfica. Cuando uno se canse de ella, basta incluir legend('off') en el *script* o teclearlo en la ventana de comandos.

Otra manera de superponer gráficas es con hold on. El programa anterior equivale a

```
x = linspace(0,2*pi, 100);
y = sin(x);
y2 = cos(x);
plot(x,y,'k--')
hold on
plot(x,y2,'k:')
hold off
```

El hold off no es, en rigor, necesario, pero sí muy aconsejable porque lo necesitaremos si queremos utilizar la ventana para nuevas gráficas.

Un ejemplo iluminador (que agradezco a Francisco Mengual). Es dibujar una circunferencia usando `plot`. Lo natural parece superponer dos gráficas de semicircunferencias. Por ejemplo:

```
1 x = linspace(-1,1, 100);
2 y = sqrt(1-x.^2);
3 plot(x,y,'k')
4 hold on
5 y = -sqrt(1-x.^2);
6 plot(x,y,'k')
7 axis('square')
8 hold off
```

La línea 7 es para que la ventana de la figura sea cuadrada y así no deforme la escala. Otra posibilidad, conservando la ventana rectangular habitual, sería `axis('equal')`. Sin embargo hay una solución más elegante. Conocemos una parametrización de la circunferencia con un solo trozo, $x = \cos t$, $y = \operatorname{sen} t$ y nadie ha dicho que los valores que aparecen como `x` tengan que ser crecientes. Con esta idea, utilizamos

```
1 t = linspace(0,2*pi, 100);
2 plot (cos(t),sin(t))
3 axis('square')
```

Hay muchas finuras para adornar nuestras gráficas, como cambiar el tamaño de las fuentes o incluir texto. Todas esas cosas exceden lo que nosotros vamos a usar en el curso. Si quieres "lucirte", lo mejor es que consultes la documentación.

1.6. Ejemplos de gráficas en tres dimensiones

En los cursos de análisis de primero es muy posible que sacaras la idea de que la representación de gráficas de funciones $y = f(x)$ siguen una mecánica (crecimiento, decrecimiento, asíntotas, puntos de inflexión ...), pero la de superficies $z = f(x, y)$ es una especie de arte. Un problema que se refleja en el *software* es que cuando representamos un objeto tridimensional en una pantalla plana, el punto de vista es muy importante para darnos una idea certera. En `matlab/octave` hay muchos comandos para la representación de superficies, pero aquí solo veremos los dos más simples y algunos ejemplos sin muchas explicaciones mostrando resultados más atrayentes.

Los comandos más básicos para representar superficies son `mesh` y `surf`. Ambos tienen tres argumentos, la x, la y y la z, y dibujan una malla correspondiente a la discretización de los datos. Difieren en que el primero muestra la malla como si las "baldosas" que limitan fueran blancas mientras que el segundo las colorea.

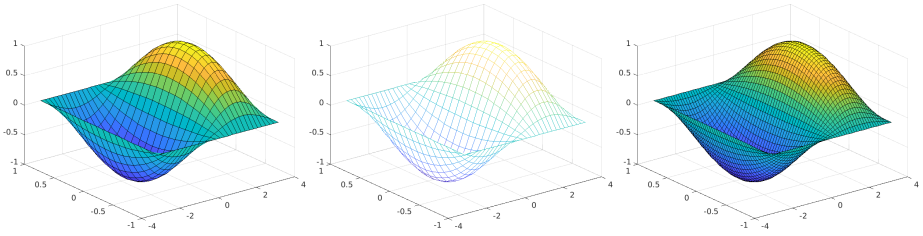
Supongamos que queremos dibujar la superficie $z = (1 - y^2) \operatorname{sen} x$ con $(x, y) \in [-\pi, \pi] \times [-1, 1]$. Esta fórmula es `sin(x).*(1-y.^2)` en formato `matlab/octave`. Un error de principiante sería definir `x` e `y` como vecto-

res, al igual que en la sección anterior. Si lo pensamos un instante, esto no puede funcionar porque con la expresión anterior estamos pidiendo a `matlab/octave` que haga productos elemento a elemento, por ejemplo $x = -\pi$ irá con $y = -1$. Lo que en realidad deseamos es que $x = -\pi$ se combine con *todos* los valores de y. La manera de conseguir este efecto es utilizar el comando `meshgrid` que repite el vector `x` tantas veces como valores de `y` haya y hace lo mismo con `y`, convirtiendo a `x` e `y` en matrices (bidimensionales). Si no quieres dedicar tiempo a entender este argumento, todo lo que tienes que saber es usar el siguiente ejemplo como plantilla:

```
1  [x,y] = meshgrid(linspace(-pi,pi,30), linspace(-1,1,20));
2  z = sin(x).*(1-y.^2);
3  surf(x,y,z)
```

Por supuesto, los nombres `x`, `y`, `z` son arbitrarios. Si todavía no has perdido la esperanza de entender lo que hace realmente `meshgrid` mira la salida que ofrece `[a,b] = meshgrid([1,2,3],[7,8])`.

Las siguientes figuras muestran el resultado de `surf`, `mesh` y el efecto de discretizar con `surf` de manera más fina:



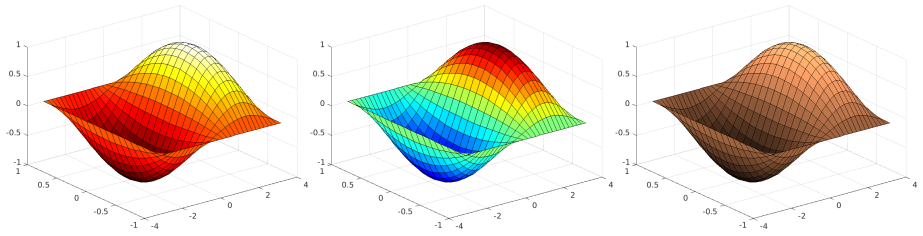
`surf` `mesh` `surf 30, 20 → 50`

Los comandos `xlabel`, `ylabel` y `title` funcionan como antes y además se añade un `zlabel`.

Los colores asignados dependen del valor de la altura siguiendo lo que se llama un *mapa de color* que se especifica mediante `colormap(`*nombre*`)` o `colormap` *nombre*. Algunos de los mapas de color predefinidos son:

`hsv, hot, cool, pink, gray, bone, jet, copper`

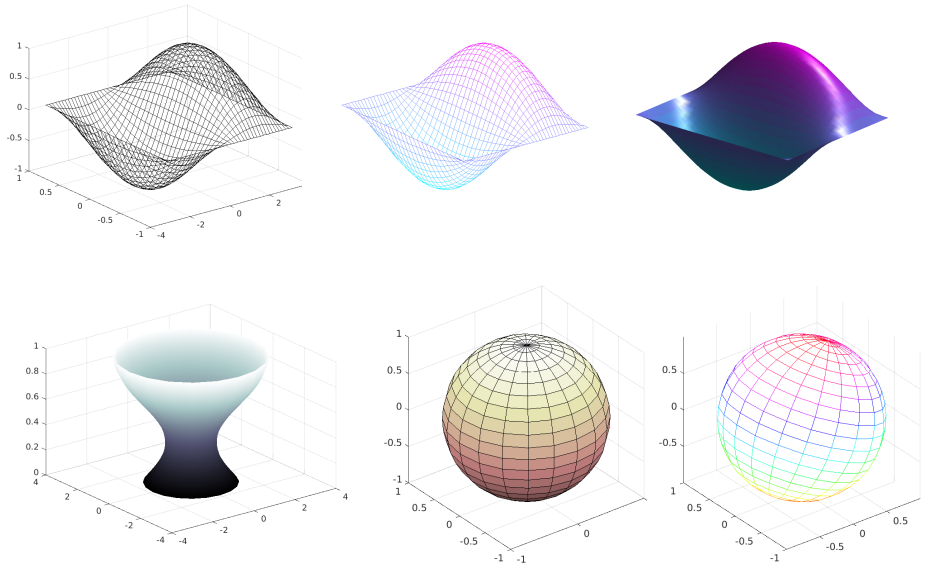
Aquí van tres ejemplos:



`colormap hot` `colormap jet` `colormap copper`

Mirando la documentación aprenderás a crear tus propios mapas de colores, aunque pocos usuarios lo encuentran necesario.

Terminamos con seis ejemplos que dan lugar a las siguientes figuras con el código de más abajo.



En la última figura se ha intentado simular la inclinación de unos 23° del eje de la Tierra (girar 180° alrededor de un eje inclinado 11,5°).

```
[x,y] = meshgrid(linspace(-pi,pi,40), linspace(-1,1,30));
z = sin(x).*(1-y.^2);

figure(1)
surf(x,y,z,'FaceColor','none')

figure(2)
mesh(x,y,z)
colormap cool
axis off

figure(3)
surf(x,y,z,'EdgeColor','none')
colormap cool
camlight right
lighting phong
view([-50,30])
axis off

figure(4)
th = linspace(-3,pi/2,100);
r = 2 + sin(th);
[x,y,z] = cylinder(r);
surf(x,y,z)
shading interp
colormap bone

figure(5)
sphere(20)
colormap pink
axis square

figure(6)
[x,y,z] = sphere(20);
```

```
35  eje_rot = [sin(11.5*pi/180) 0 cos(11.5*pi/180)];
36  rotate( mesh(x,y,z), eje_rot , 180)
37  colormap hsv
38  axis equal
```

En `octave` la línea 16, que especifica cierto algoritmo para la iluminación, no funciona. Sin ella se ven los parches que forman la superficie.

Quien tenga interés en los comandos y parámetros que aparecen en el código anterior, debería consultar la documentación.

1.7. Control de flujo

Al igual que en cualquier lenguaje de programación típico, es posible alterar el orden natural en la ejecución de las líneas de un programa con `for` y con `if`. La sintaxis de `matlab/octave` fuerza a que ambos deban acabar con `end`. La sangría (empezar los renglones más a la derecha) no es obligatoria, pero favorece la legibilidad del código.

Lo único que puede resultar un poco singular de los bucles `for` es que la variable recorre típicamente un vector fila. Por ejemplo, si quisiéramos que se mostrasen los cuadrados de los enteros del 1 al 20, usaríamos:

```
1  for k = 1:20
2      disp(k^2)
3  end
```

Aunque no es tan habitual, también la variable puede recorrer matrices (bidimensionales) y como estas pueden considerarse un vector fila de columnas, tomará como valores cada una de las columnas.

Para utilizar `if` hay que decir algo de los operadores lógicos. Igual que en C/C++, las condiciones AND y OR se indican mediante `&&` y `||`. Por otro lado, $\sim$ es la negación. Por ejemplo, si en nuestra lista de cuadrados queremos saltarnos el 13 porque pensamos que da mala suerte, una manera de proceder es:

```
1  for k = 1:20
2      if k ~= 13
3          disp(k^2)
4      end
5  end
```

También existen `break` y `continue` con el significado habitual (salir totalmente de un bucle y pasar a la siguiente iteración). Así una alternativa para el código anterior es

```
1  for k = 1:20
2      if k == 13
3          continue
4      end
5      disp(k^2)
6  end
```

Cualquiera con experiencia programando sabe por qué aparece `k == 13` en la línea 2 en vez de `k = 13`. Un igual duplicado indica comparación y uno simple, asignación.

Se puede completar la estructura de `if` con `elseif`, para añadir nuevas condiciones, o con `else` para indicar la alternativa a todas ellas. En este ejemplo se pide un número del 1 al 10 y muestra diferentes mensajes dependiendo de su valor:

```
1  n = input('Introduce un número del 1 al 10');
2  if (n<1) || (n>10)
3      disp('No me has hecho caso: no está en el rango')
4  elseif mod(n,2)==0
5      disp('Es un número par')
6  elseif (n-5)^2==16
7      disp('Es un cuadrado impar')
8  elseif (n==3) || (n==5) || (n==7)
9      disp('Es primo')
10 else
11     disp('No es entero, ¿eres matemático?')
12 end
```

Seguro que conoces de antemano el comando `while`. Sirve para ejecutar un bloque mientras que la condición especificada se cumpla.

```
1  % Halla el primer k tal que sqrt(1)+...+sqrt(k) > n
2  n = 10;
3  k = 1;
4  s = 1;
5  while n > s
6      k = k + 1;
7      s = s + sqrt(k);
8  end
9  disp(k)
```

También existe, como en otros lenguajes de programación, un comando `switch`, que no veremos aquí.

1.8. Funciones

Vamos a crear una función (una subrutina) llamada `fibo` de forma que `fibo(n)` dé el `n`-ésimo número de Fibonacci aprovechando el *script* que habíamos creado antes.

En `matlab`, la pestaña de *Current Folder* con el botón derecho seleccionamos `function` y creamos el fichero `fibo.m`. También lo podemos hacer por medio del menú principal eligiendo `New>function`. En `octave`, seguiremos el menú `File > New > New function`.

Tanto en `matlab` como en `octave`, nos aparecerá un plantilla general para funciones. La de `matlab` es:

```
1  function [outputArg1,outputArg2] = fibo(inputArg1,inputArg2)
2  %FIBO Summary of this function goes here
3  %   Detailed explanation goes here
4  outputArg1 = inputArg1;
5  outputArg2 = inputArg2;
6  end
```

En realidad, podríamos no emplearla y crear el fichero por nuestra cuenta como cualquier otro *script*. Lo que es realmente importante, y distinto a los lenguajes de programación típicos, es que el nombre de la función debe coincidir con el nombre del fichero.

Nosotros solo queremos un argumento de salida, que llamaremos `Fn` y otro de entrada que llamaremos `n`. Por lo demás, copiamos el programa para calcular números de Fibonacci y asignamos el resultado final a la variable `Fn`, cuyo valor será el que devuelve la función.

```
1  function Fn = fibo(n)
2    % fibo(n) calcula el número de Fibonacci Fn
3    % Elevamos a n
4    F = [1,1;1,0]^n;
5    % El resultado es el elemento 1,2 (o el 2,1)
6    Fn = F(1,2);
7  end
```

La sangría, que no aparece en la plantilla, de nuevo solo es para favorecer la legibilidad del código. Es natural que en las funciones casi todas las líneas acaben con punto y coma porque no queremos que nos muestren resultados intermedios, sino que actúen como cajas negras que respondan a cada entrada devolviendo un resultado.

Ahora necesitamos un *script* que use la función. Creemos `lista_fibo.m` que dé los primeros 10 elementos de la sucesión de Fibonacci llamando a `fibo`. Para ello debe estar en el mismo directorio que el fichero `fibo.m` de la función. Un posible contenido de `lista_fibo.m` es, por hacer lo más simple,

```
1  for n = 1:10
2      disp(fibo(n))
3  end
```

Sí, seguro que a la mayoría nos parece que el formato del resultado es feo, en la siguiente sección veremos cómo mejorarlo.

Las funciones no están limitadas a argumentos y valores de salida escalares reales, pueden ser también matriciales, en particular vectoriales.

Con `return` se fuerza el retorno desde una función. Esto es útil cuando se utilizan condiciones.

1.9. Ficheros y entrada y salida

En primer lugar vamos a mejorar un poco la presentación de nuestra lista de números de Fibonacci con

```
1  for n = 1:10
2      disp(['El número de Fibonacci ' num2str(n) ' es ' num2str(fibo(n))])
3  end
```

que produce salidas del tipo `El número de Fibonacci ... es ...` con los puntos suspensivos sustituyendo el índice y el valor. La lógica para esto es que `num2str` transforma el número en cadena de caracteres (*number to*

string) y las cadenas de caracteres se pueden yuxtaponer en `matlab`. Los corchetes son necesarios para no hacer un lío a `matlab/octave` con el fin de la cadena de caracteres.

Una manera alternativa con idéntico resultado que nos recordará más al `printf` de C, es:

```
1 for n = 1:10
2     fprintf('El número de Fibonacci %d es %d\n',n,fibo(n))
3 end
```

Como en C, el uso natural de `fprintf` no es imprimir en pantalla sino en ficheros. Para ello hay que que abrir un fichero, obteniendo su identificador y al final cerrarlo. Es prácticamente idéntico a C. Con el siguiente código:

```
1 fid = fopen('mifichero.dat','w');
2 fprintf(fid, 'Minitabla de números de Fibonacci\n');
3
4 for n = 1:5
5     fprintf(fid,'%d --> %d\n',n,fibo(n));
6 end
7 fclose(fid);
```

se genera un fichero conteniendo

```
Minitabla de números de Fibonacci
1 --> 1
2 --> 1
3 --> 2
4 --> 3
5 --> 5
```

Para los que tengan el C más oxidado, o lo desconozcan, aquí van algunas explicaciones. En la línea 1 se abre el fichero `mifichero.dat` en modo de escritura (la `w` es de *write*). Tras ello, `fid` identificará las posiciones de memoria correspondientes al fichero. Con `fprintf(fid,...)` se va escribiendo en ellas. Cuando terminemos de escribir, en nuestro caso en la línea 7, hay que cerrar el fichero con `fclose`.

Para leer un fichero hay que usar `r` (la inicial de *read*) en vez de `w`. Esencialmente `fscanf` es el comando contrario que `fprintf`. En nuestro caso, queremos omitir la lectura del título, por lo que usaremos `fgetl`, que lee una línea. Una posibilidad para recuperar los datos de la lista a partir del fichero es:

```
1 fid = fopen('mifichero.dat','r');
2 titulo = fgetl(fid);
3 datos = fscanf(fid,'%d --> %d\n');
4 fclose(fid);
5 reshape(datos,2,5)'
```

A diferencia de lo que ocurriría en C, no es necesario un bucle, `fscanf` lee todos los datos como un vector. Tras ello, el contenido de `datos` es 1, 1, 2, 1, 3, 2, 4, 3, 5, 5, los diez datos en el orden leído. La instrucción `reshape` de la última línea es para darle la forma de matriz 5×2 que corresponde a la tabla. La trasposición que produce el apóstrofo es solo una cuestión estética,

para que la tabla tenga la misma orientación que la del fichero de datos.

En realidad, `fscanf` admite un último argumento indicando las dimensiones de la salida y entonces el anterior código tiene la versión reducida equivalente:

```
fid = fopen('mifichero.dat','r');
titulo = fgetl(fid);
fscanf(fid,'%d --> %d\n',[2,5])'
fclose(fid);
```

Sabemos guardar una figura usando la interfaz. Con el comando `saveas` lo conseguiremos automáticamente por *software*. Por ejemplo, para guardar la superficie que habíamos usado en un fichero en formato `png` llamado `superficie.png`

```
[x,y] = meshgrid(linspace(-pi,pi,30), linspace(-1,1,20));
z = sin(x).*(1-y.^2);
surf(x,y,z)
saveas(gcf,'superficie.png')
```

Aquí `gcf` es la manera en la que `matlab/octave` indica la figura en curso. Supongo que son las iniciales de *get current figure*, pero no estoy totalmente seguro.

Un ejemplo más avanzado es el siguiente, que crea tres imágenes consistentes en traslaciones de la función seno y asigna automáticamente los nombres `seno1.png`, `seno2.png` y `seno3.png`.

```
x = linspace(0,2*pi,100);
for k = 1:3
    plot(x, sin(x+k))
    nombre = sprintf('seno%d.png',k);
    saveas(gcf,nombre)
end
```

De nuevo, `sprintf` viene heredado de C.

Es raro que utilicemos en el curso programas como el anterior, e incluso la lectura y escritura en ficheros. No obstante, todo ello es muy útil si tenemos que tratar con cantidades relativamente grandes de figuras o datos. Cuando generamos muchas imágenes, sería pesadísimo guardarlas una a una con el interfaz y bucles como el anterior se vuelven necesarios.

Si tienes un número reducido de datos, pero te ha costado mucho generarlos, te resultará de interés guardar todas las variables activas en un momento dado (puedes curiosear cuáles son en la ventana del `workspace`). Por ejemplo, supongamos que `f` almacena el número `65537`, el mayor primo de Fermat conocido hasta la fecha, y quieres recordarlo en una futura sesión. Puedes guardar `f` en el fichero `mis_variables.mat`, junto con el resto de las variables mediante `save('mis_variables.mat')`. Si ahora te sales de la sesión y vuelves a entrar o escribes `clear`, que sirve para que se borren todas las variables, la variable `f` estará indefinida. Para volverla a la vida, basta con que ejecutes `load('mis_variables.mat')`.

1.10. Para saber más

Naturalmente, te surgirá más de una duda con los comandos que menos emplees o querrás aprender otros nuevos. Para lo primero, la ayuda de `matlab` es una buena fuente de información. Si escribes en la ventana de comandos (*command window*) `help` *comando*, te mostrará una ayuda en modo texto que terminará con un enlace a la página correspondiente de *Matlab Online Help*. Por ejemplo, con `help plot` obtenemos una larga parrafada conteniendo una descripción general, los usos más comunes y un ejemplo. Al final se incluyen dos enlaces: *Documentation for plot* y *Other functions named plot*. El primero es el más interesante y nos da la posibilidad de abrir los ejemplos como *live scripts*, una forma decorada de *scripts*, que podemos ejecutar y modificar.

La documentación está en el sitio web

`https://www.mathworks.com/help/matlab/`

A veces he encontrado alguna deficiencia en la traducción al español, por tanto suelo seleccionar el sitio web de Estados Unidos. Allí puedes explorar los temas que prefieras. Si te fijas, hay un enlace a la derecha en el que anuncia la documentación en PDF. Si lo sigues, necesitarás entrar en tu cuenta. A este nivel, el fichero en el que estarás interesado es *MATLAB Primer*. Nota que es bastante largo, 164 páginas, por tanto no lo imprimas si no estás seguro de ello.

Si empleas `octave`, de nuevo `help` *comando* da acceso a la documentación en modo texto. Recuerda usar `more off` si prefieres que no te salga línea a línea cuando la salida es larga. La documentación está en el sitio web

`https://octave.org/doc/v6.1.0/`

Mi experiencia es que está menos cuidada que la de `matlab`, tanto es así que muchas veces prefiero consultar la ayuda de `matlab` para algunos comandos a pesar de estar empleando `octave`. Hay una versión PDF en `https://www.gnu.org/software/octave/octave.pdf` que seguro que no querrás imprimir porque consta de 1121 páginas.

Si quieres más información, algunas referencias (no muy seleccionadas, a decir verdad) son [8], [17], [18], [7], [14]. La primera es un manual en PDF que, por la cantidad de años que lleva empleándose, podría tildarse de clásico. De todas formas, es muy fácil encontrar buen material de aprendizaje con búsquedas más o menos aleatorias en la red.

Capítulo 2

Consideraciones básicas

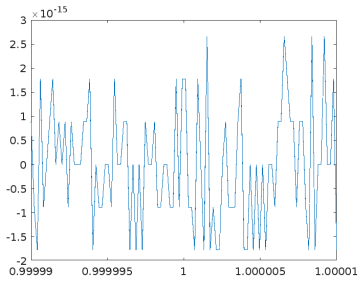
2.1. Un par de sorpresas con el épsilon máquina

La precisión de un ordenador con los cálculos aritméticos básicos es tan grande en comparación con la nuestra que estamos acostumbrados a que no tenga ningún efecto significativo sobre el resultado de nuestros programas. Sin embargo, hay que tomar esta idea con algo de precaución porque algunos cálculos pueden agrandar esa pequeña imprecisión intrínseca dada por el épsilon máquina hasta hacerla bien notoria. Vamos a verlo a través de dos ejemplos, el primero está adaptado de [17] y el segundo es clásico.

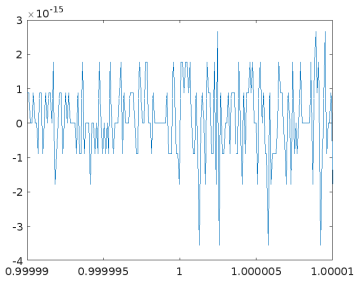
Consideremos la gráfica de un polinomio en apariencia inocente con un resultado rarísimo:

```
1  x = linspace(0.99999,1.00001,100);
2  y = x.^5 - 5*x.^4 + 10*x.^3 - 10*x.^2 + 5*x - 1;
3  plot(x,y)
```

No es problema de la discretización. De hecho, al incrementar `100`, la situación empeora. Además, cambiando cualquiera de los signos de la línea 2, el dibujo sí responde a la realidad del nuevo polinomio.



`linspace(0.99999,1.00001,100)`



`linspace(0.99999,1.00001,200)`

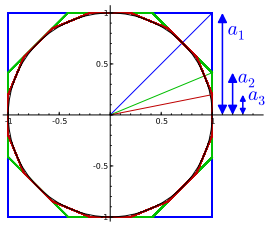
La explicación es que, simbólicamente (con precisión infinita), se tiene $y = (x-1)^5$, pero en nuestro intervalo la x se mueve en 1 ± 10^{-5} y como

$\left(10^{-5}\right)^5$ supera el épsilon máquina, `matlab/octave` no es capaz de detectar la cancelación.

ACTIVIDAD 2.1.1. *Trata de dar más detalles en la explicación anterior. Para ver que lo has entendido, prevé cuál debe ser el orden de la anchura del intervalo alrededor de 1 para empezar a ver cosas raras y comprueba tu predicción.*

ACTIVIDAD 2.1.2. *Experimenta con lo indicado acerca de la discretización y los cambios de signo, explicando cualitativamente los resultados.*

El segundo ejemplo, tiene que ver con un algoritmo para aproximar π. Si llamamos a_n a la longitud de la mitad del lado del polígono regular de 2^{n+1} lados circunscrito en la circunferencia unidad entonces su semiperímetro es $2^{n+1}a_n$, por tanto $2^{n+1}a_n \to \pi$ y la geometría de la situación sugiere que la convergencia es muy rápida. Para que este método sea útil, deberíamos buscar algún algoritmo para calcular a_n.



$a_n =$ mitad del lado del polígono de 2^{n+1} lados

$2^{n+1}a_n \to \pi$

Claramente $a_n = \tan\left(\pi/2^{n+1}\right)$ y por las fórmulas del ángulo doble con $\alpha = \pi/2^n$

$$a_n = \frac{1-\cos\alpha}{\operatorname{sen}\alpha} = \frac{\frac{1}{\cos\alpha}-1}{\frac{\operatorname{sen}\alpha}{\cos\alpha}} = \frac{\sqrt{1+\tan^2\alpha}-1}{\tan\alpha} = \frac{\sqrt{1+a_{n-1}^2}-1}{a_{n-1}}.$$

Esta fórmula permite calcular a_n de manera recurrente a partir de $a_1 = 1$.

Escribamos dos variantes del algoritmo, siempre partiendo de $a_1 = 1$:

$$a_n = \frac{\sqrt{1+a_{n-1}^2}-1}{a_{n-1}} \qquad \text{y} \qquad a_n = \frac{a_{n-1}}{\sqrt{1+a_{n-1}^2}+1}.$$

Ambas fórmulas son idénticas racionalizando. Sin embargo nos empeñamos en distinguirlas en el siguiente programa llamando `an` a los resultados de la primera fórmula y `anp` a los de la segunda. Cuando lo ejecutamos, vemos que en el quinto término aparece una pequeña divergencia prácticamente inapreciable. El problema es cómo evoluciona.

```
1 an = 1;
2 anp = 1;
3 for k = 2:28
4     an = (sqrt(1+an^2)-1)/an;
5     anp = anp/(sqrt(1+anp^2)+1);
6     res = 2^(k+1)*[an,anp];
7     disp(['Iteración ' num2str(k) ' -> ' num2str(res, ' %.10f')])
8 end
```

El primer algoritmo a partir de un momento dado empieza a hacer cosas raras mientras que el segundo funciona perfectamente, a pesar de que las fórmulas son formalmente idénticas. Concretamente la salida de las cinco últimas iteraciones es:

```
Iteración 24 -> 3.1110567880 3.1415926536
Iteración 25 -> 3.0536247479 3.1415926536
Iteración 26 -> 2.6198372952 3.1415926536
Iteración 27 -> 3.0536247479 3.1415926536
Iteración 28 -> 0.0000000000 3.1415926536
```

ACTIVIDAD 2.1.3. *Da una explicación para esta diferencia tan drástica entre los algoritmos.*

ACTIVIDAD 2.1.4. *Practica con las funciones en* `matlab/octave` *sustituyendo las líneas 4 y 5 por* `an = alg1(an)` *y* `anp = alg2(anp)` *donde* `alg1` *y* `alg2` *están en ficheros separados.*

ACTIVIDAD 2.1.5. *Estudia experimentalmente el error en el algoritmo "bueno" y elabora una teoría que explique el resultado.*

2.2. Resolución de sistemas con `matlab/octave`

Si lo único que queremos es resolver con `matlab/octave` un sistema de ecuaciones lineal compatible determinado $A\vec{x} = \vec{b}$ con A una matriz (real o compleja) no singular, todo lo que tenemos que hacer es utilizar `A\b` donde `A` y `b` corresponden a A y $\vec{b}$ de la fórmula anterior. Por ejemplo,

```
1 A = [2,1; 1,-2];
2 b = [4;-3];
3 A\b
```

muestra como salida el vector columna `(1,2)`. Si te estás preguntando de dónde viene esa notación tan rara de la barra de división invertida (llamada *backslash*), te tranquilizará saber que la manera natural de definir la división de matrices B/A es BA^{-1} y para la división "al otro lado" $A^{-1}B$ no suena tan extraño cambiar en la notación la dirección de la barra.

El comando funciona incluso si `b` tiene varias columnas, lo que permite resolver varios sistemas que comparten matriz, como se requiere en algunas aplicaciones. Incluso ofrece un resultado cuando la matriz `A` is rectangular, si el número de filas coincide con el de `b`. En esa situación el sistema puede

ser incompatible. ¿Qué significa la solución de un sistema que no tiene solución? Lo que muestra es un vector que se desvía lo menos posible, en cierto sentido, de verificar las ecuaciones. Consideremos un sistema compatible indeterminado y otro incompatible:

```
A = [1,2,3;1,2,4];
b = [1;2];
A\b

A = [1,2; 1,1; 3,1];
b = [1;1;1];
A\b
```

En el primer caso hay infinitas soluciones dadas por $(-2-2t, t, 1)$ y la que muestra `matlab/octave` es la de de norma mínima `(-0.4, -0.8, 1)`. En el segundo caso, la salida es una aproximación de `(1/5, 7/15)`, que es el vector que al ser multiplicado (en columna) por `A` se queda más cerca de `b`.

Una pregunta muy natural es qué sentido tiene que exista la notación rara `A\b` cuando podríamos escribir simplemente `inv(A)*b` o `A^-1*b` donde `inv(A)` y `A^-1` son dos formas de escribir la inversa. La respuesta, es que en general es más costoso hallar una inversa que resolver un sistema y con `A\b` le avisamos a `matlab/octave` que nos da igual no conocer la inversa y se centra en los algoritmos, más rápidos, para resolver sistemas sin inversas.

De entre esos algoritmos, el que aprendimos todos en primero es la reducción de Gauss. En `matlab/octave` el comando `rref` (abreviatura de *reduced row echelon form*) produce la forma escalonada reducida de una matriz. Es decir, lo que se llama reducción de Gauss-Jordan [9] (consistente en continuar el método de Gauss hasta que la matriz formada por las columnas pivote sea la identidad. Si A es una matriz invertible, la forma escalonada reducida de $(A|I)$ es $(I|A^{-1})$, con I la matriz identidad, por tanto es posible hallar inversas indirectamente con el comando `rref`.

ACTIVIDAD 2.2.1. *Calcula la inversa de* `[1,2,-1; 2, 5, 1; -1,1,11]` *usando Gauss-Jordan con* `rref` *y con el comando directo* `inv`.

2.3. La descomposición LU con `matlab/octave`

Una forma de presentar la reducción de Gauss es la factorización o descomposición LU. En `matlab/octave` se obtiene con el comando `lu`, pero hay que tener en cuenta que el resultado no corresponde a la factorización que habrás visto en primer lugar en la clase de teoría, sino que es la descomposición LU con pivotaje. Por ejemplo:

```
A = [1,2,1;3,4,5;5,8,1];
[L,U] = lu(A);
disp('L =')
disp(L)
disp('U =')
disp(U)
```

```
7  disp(L*U)
```

Es verdad que `L*U` recupera `A`, es una verdadera factorización, pero `L` no es triangular inferior. La razón es que lleva incorporada la matriz de permutación que corresponde a los pivotes empleados.

Si queremos que `L` sea realmente triangular inferior, debemos separar la permutación permitiendo un tercer argumento de salida de `lu`:

```
1  A = [1,2,1;3,4,5;5,8,1];
2  [L,U,P] = lu(A);
3  disp('L =')
4  disp(L)
5  disp('U =')
6  disp(U)
7  disp(L*U)
8
9  disp(P*A)
```

En breve, el resultado de `lu` con tres argumentos de salida, es la factorización LU de `A` con las filas reordenadas. Tal reordenación es interesante desde el punto de vista de la eficiencia numérica cuando se trabaja con matrices grandes.

ACTIVIDAD 2.3.1. *Busca dos ejemplos de matrices* 2×2 *de forma que* `[L,U] = lu(A)` *produzca una matriz triangular superior* `L` *para la primera y no para la segunda. Si no tienes suerte probando al azar, te ayudará saber, o recordar, que lo de "pivotaje" se refiere al tamaño relativo de los pivotes al aplicar reducción de Gauss.*

2.4. Programando la descomposición LU básica

Nuestro propósito ahora es hacer nuestro propio código que para efectuar la descomposición LU "pura", esto es, sin pivotes. El plan ahora es simplemente traducir el pseudocódigo visto en la clase de teoría. Más adelante, lo refinaremos un poco. La traducción es totalmente inmediata salvo aprender que `size` con un argumento extra indica solo la dimensión seleccionada. Es decir, si `A` es una matriz 20×21 entonces `size(A)` es `[20,21]`, `size(A,1)` es 20 y `size(A,2)` es 21.

```
1  A = [1,2,1;3,4,5;5,8,1];
2  n = size(A,1);
3  U = A;
4  L = eye(n);
5
6  for k = 1:n-1
7      for ii = k+1:n
8          L(ii,k) = U(ii,k)/U(k,k);
9          for j = k:n
10             U(ii,j) = U(ii,j) - L(ii,k)*U(k,j);
11         end
12     end
13 end
14
15 disp('L =')
16 disp(L)
```

```
17  disp('U =')
18  disp(U)
19  disp(L*U)
```

Utilizar `ii` en vez de `i` es solo una manía mía para no sobreescribir el valor `i`$=\sqrt{-1}$ de `matlab/octave`. Con la matriz de prueba de la primera línea, el resultado debe ser

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 5 & 1 & 1 \end{pmatrix} \qquad \text{y} \qquad U = \begin{pmatrix} 1 & 2 & 1 \\ 0 & -2 & 2 \\ 0 & 0 & -6 \end{pmatrix}.$$

Multiplicando estas dos matrices obtenemos `A` sin que medie ninguna matriz de permutación. Esta comprobación es la razón de ser de la última línea del código.

Capítulo 3

La descomposición LU

3.1. Vectorización de la descomposición LU

Aprovechándonos de que las matrices y vectores son estructuras de datos básicas en `matlab/octave`, en vez de tres bucles podemos utilizar solo dos en el código antes indicado para la descomposición LU. Cambiamos el más interno, el que involucra `j` por una sola línea que considera una porción de la `ii`-ésima fila de `U` como un todo en vez de recorrerla elemento a elemento con el bucle en `j`:

```
U(ii,k:n) = U(ii,k:n) - L(ii,k)*U(k,k:n);
```

Es decir, esto reemplaza a las líneas 9, 10 y 11.

La filosofía es que en `matlab/octave` las matrices o vectores sirven a veces como sustituto para bucles. ¿Hay alguna ventaja en operar un trozo de fila de `U` como un vector en vez de operar sus elementos? A fin de cuentas, sumar dos vectores de m coordenadas o hacer un bucle que recorra m sumas es lo mismo. La ventaja es que el trabajo se deja a la parte que no vemos, la parte no interpretada, y `matlab/octave` es muy bueno operando internamente con matrices y vectores. A diferencia de lo que nos ocurriría a nosotros trabajando a mano, le va a costar más acceder por separado a cada coordenada que tratar el vector como un todo, o al menos es lo que dice la documentación (véanse los apuntes sobre el rendimiento más adelante).

Con mentalidad informática, uno puede pensar que el `k:n` que aparece en la nueva línea, exigirá reservar memoria tres veces para vectores que son iguales, entonces es de sospechar que reemplazando el interior del bucle `for` en `k` por

```
v = k:n;
for ii = k+1:n
    L(ii,k) = U(ii,k)/U(k,k);
    U(ii,v) = U(ii,v) - L(ii,k)*U(k,v);
end
```

obtendremos alguna ventaja en cuanto al rendimiento. Es decir, el código completo sería

```
n = size(A,1);
U = A;
L = eye(n);

for k = 1:n-1
    v = k:n;
    for ii = k+1:n
        L(ii,k) = U(ii,k)/U(k,k);
        U(ii,v) = U(ii,v) - L(ii,k)*U(k,v);
    end
end
```

ACTIVIDAD 3.1.1. *Hay un teorema [17, §5.1] que dice que una matriz A tiene descomposición LU "pura" (sin pivotes) si y solo si los menores angulares, exceptuando A, son no nulos. Busca una matriz entera 3×3 con menores angulares $\Delta_1, \Delta_3 \neq 0$ y $\Delta_2 = 0$ y comprueba qué tipo de error o resultado produce el código anterior.*

Para entender más acerca de cómo funciona el código anterior, apliquémoslo a la matriz

$$A = \begin{pmatrix} 2 & 1 & -1 & 1 \\ 4 & 3 & 0 & 3 \\ -2 & 1 & 4 & -2 \\ 2 & 3 & 0 & -4 \end{pmatrix}$$

e introduzcamos en el código unas línea que muestren las matrices `L` y `U` después del bucle interior. Lo que obtenemos para `L` en cada uno de los tres pasos de `k` es:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & 2 & 1 & 0 \\ 1 & 2 & 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & 2 & 1 & 0 \\ 1 & 2 & 3 & 1 \end{pmatrix}.$$

y para `U`:

$$\begin{pmatrix} 2 & 1 & -1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 2 & 3 & -1 \\ 0 & 2 & 1 & -5 \end{pmatrix}, \quad \begin{pmatrix} 2 & 1 & -1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & -1 & -3 \\ 0 & 0 & -3 & -7 \end{pmatrix}, \quad \begin{pmatrix} 2 & 1 & -1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & -1 & -3 \\ 0 & 0 & 0 & 2 \end{pmatrix}.$$

Vemos que estas tres matrices corresponden a los diferentes pasos de la eliminación de Gauss tal como la llevaríamos a cabo nosotros "a mano". Por otra parte, las columnas de L se van rellenando con los coeficientes que usamos para crear ceros bajo los pivotes.

3.2. Ahorrando memoria

Para almacenar los elementos de una matriz $n \times n$ necesitamos n^2 posiciones de memoria (o algo proporcional a ello). Al efectuar la descomposición LU, necesitamos también guardar los elementos de L y de U. Ya que L y U están medio vacías, economizando tendríamos ocupadas del orden de $n^2+n^2/2+n^2/2 = 2n^2$ posiciones de memoria. Con los programas anteriores no estamos aplicando dicha economía porque guardamos todos los elementos de L y U, aunque sean nulos.

En realidad, en el caso de la descomposición LU sin pivotes hay una manera sencilla de proceder que permite reducir las posiciones de memoria necesarias a la mitad. Es el *algoritmo de Crout* [15, §2.3]. La idea es sencilla: sobreescribir la matriz A con los elementos de las matrices L y U. Aunque una matriz triangular tiene genéricamente $n(n+1)/2$ elementos no nulos, sabemos que la diagonal de L está formada por n unos y no hace falta almacenarlos, así que la aritmética cuadra perfectamente:

$$\frac{n(n+1)}{2} + \left(\frac{n(n+1)}{2} - n\right) = n^2.$$

Con una pequeña salvedad (véase la siguiente actividad), el código es como antes, identificando con `A` las matrices `L` y `U`, que ahora desaparecen:

```
1  A = [1,2,1;3,4,5;5,8,1];
2  n = size(A,1);
3  for k = 1:n-1
4      v = k+1:n;
5      for ii = k+1:n
6          A(ii,k) = A(ii,k)/A(k,k);
7          A(ii,v) = A(ii,v) - A(ii,k)*A(k,v);
8      end
9  end
10 disp('A')
11 disp(A)
```

La matriz `A` contiene toda la información: la mitad sobre la diagonal, incluida, produce U y el resto completando la diagonal con unos, da L. Para aprender nuevos comandos, completemos el código anterior para que muestre L y U. En `matlab/octave` existen `tril` y `triu` que dan la parte triangular inferior y superior de una matriz, por tanto U es simplemente `triu(A)`. Estos dos comandos admiten un segundo argumento, un entero positivo o negativo, que indica cuántas posiciones queremos separarnos de la diagonal. Con ello, `tril(A,-1)` da L salvo la diagonal. En definitiva, L y U aparecerán en el formato habitual añadiendo las líneas:

```
disp('L =')
disp(tril(A,-1) + eye(n))
disp('U =')
disp(triu(A))
```

ACTIVIDAD 3.2.1. *Comprueba si el programa funciona con el* `v = k:n` *que habíamos usado antes. ¿Por qué quitamos la primera coordenada a* `v`*?*

3.3. Sistemas lineales con LU

Recordemos que la descomposición LU no era un capricho teórico sino que tenía como propósito resolver rápidamente sistemas de ecuaciones lineales. Es una realización de la reducción de Gauss. El hecho clave es que si $A = LU$, resolver $A\vec{x} = \vec{b}$ se reduce a resolver sucesivamente los sistemas $L\vec{y} = \vec{b}$ y $U\vec{x} = \vec{y}$. Ahora bien, es nuy fácil hallar la solución de un sistema lineal con una matriz triangular. Simplemente basta despejar y sustituir sucesivamente de abajo a arriba, si la matriz es triangular superior, o de arriba a abajo, si la matriz es triangular inferior. Concretamente:

El sistema $L\vec{x} = \vec{b}$ con L triangular inferior, tiene como solución

$$x_i = \frac{1}{l_{ii}}\Big(b_i - \sum_{j=1}^{i-1} l_{ij}x_j\Big), \qquad \text{con} \quad x_1 = \frac{b_1}{l_{11}}.$$

Y el sistema $U\vec{x} = \vec{b}$ con U triangular superior, tiene como solución

$$x_i = \frac{1}{u_{ii}}\Big(b_i - \sum_{j=i+1}^{n} u_{ij}x_j\Big), \qquad \text{con} \quad x_n = \frac{b_n}{u_{nn}}.$$

Actividad 3.3.1. *Crea funciones* `ltrs` *y* `utrs` *(de lower y upper triangular system) que resuelvan sistemas triangulares usando estas fórmulas.*

Una vez definidas las funciones de la actividad anterior, es inmediato construir nuestro algoritmo para resolver sistemas lineales. Con `mlu` se indica en el siguiente código la función correspondiente a nuestra implementación de la descomposición LU.

```
1 [L,U] = mlu(A);
2 y = ltrs(L,b);
3 x = utrs(U,y);
4 disp(x)
```

Seguro que con esto no ganaremos en velocidad al `A\b` de `matlab/octave`, pero sí será más eficiente y rápido para sistemas grandes que calcular la inversa `inv(A)` y multiplicarla por `b`. Recuerda, por otra parte, que nuestro algoritmo no es infalible porque no todas las matrices tienen una descomposición LU, aunque, en cierto sentido, hay probabilidad nula de que nos topemos con una excepción eligiendo matrices al azar.

Actividad 3.3.2. *Define la función* `mlu` *para que el código anterior sea operativo.*

3.4. La descomposición LU con pivotes

La técnica de los pivotes requiere calcular el elemento máximo de una columna, el pivote, y pasarlo a la diagonal intercambiando filas. El algoritmo

será igual que antes salvo intercalar este procedimiento. Para apreciar la ventaja de esta complicación sobre un ejemplo, véase [1, §8.2].

El máximo de un vector `v` se calcula en `matlab/octave` con `max(v)`. Si sustituimos `v` por una matriz lo que obtendremos es el vector fila dado por los máximos de las columnas. Nosotros más que el máximo, lo que queremos saber es el índice del máximo, la posición en al que se alcanza, para así trasladarlo a su nueva posición. Es lo que en el contexto del análisis a veces se escribe arg máx. Por ejemplo, el máximo de $f(x) = 1 - x^2$ es 1 y $\arg\max f = 0$ porque el máximo se alcanza en $x = 0$. Para que `matlab/octave` nos dé la posición del máximo basta recuperar la salida de `max` con dos variables, la primera será el máximo y la segunda su posición. El siguiente código es un ejemplo de su funcionamiento.

```
1  v = [3,5,-7,1];
2  m = max(v);
3  disp(['Valor máximo de las coordenadas ' num2str(m)])
4  [m, j] = max(v);
5  disp(['Posición ' num2str(j)])
```

En rigor el pivote no se calcula con el máximo de los elementos de una columna sino con el máximo de sus valores absolutos. Eso tiene tan fácil solución como introducir la función `abs` que toma valores absolutos (o normas en el caso de números complejos) de los elementos de una matriz. De esta forma, cambiando `max(v)` por `max(abs(v))` la salida pasa a ser 7 alcanzado en la posición 3.

Actividad 3.4.1. *Escribe un programa que utilice una discretización de la función $f(x) = x^2 e^{-x}$ y el comando* `max` *para hallar una aproximación del máximo de f en $[0, 4]$ y el valor en el que se alcanza. Haz también los cálculos analíticamente y compara los resultados.*

Lo segundo que tenemos que pensar es cómo intercambiar dos filas en `matlab/octave`. Una solución fea y poco eficiente, que no escribo para que no dé malas ideas a nadie, es intercambiar uno a uno los elementos de las filas. Más elegante es tratar las filas como vectores e intercambiarlas como se haría en C/C++, u otros lenguajes, por medio de una variable temporal intermedia. Por ejemplo, intercambiaríamos las filas 2 y 3 de `A` mediante

```
1  A = [1,2,1;3,4,5;5,8,1];
2  temp = A(2,:);
3  A(2,:) = A(3,:);
4  A(3,:) = temp;
```

Sin embargo hay algo todavía más breve que recuerda mucho a la solución para el intercambio de variables en python. Podemos "vectorizar" también los índices de las filas a intercambiar y conseguir nuestro propósito cambiando su orden con una sola instrucción:

```
1  A = [1,2,1;3,4,5;5,8,1];
2  A([2 3],:) = A([3 2],:);
```

Tras estas consideraciones, el pseucódigo visto en la clase de teoría, con la matriz ejemplo `A` considerada allí, se traduce en:

```
1  A = [3,2,-1;6,6,2;-1,1,3];
2
3  n = size(A,1);
4  U = A;
5  L = eye(n);
6  P = eye(n);
7
8  for k = 1:n-1
9          [m,j] = max( abs(U(k:n,k)) );
10         if j~=1
11                 r = j+k-1;
12                 U([k,r], k:n) = U([r,k], k:n);
13                 L([k,r], 1:k-1) = L([r,k], 1:k-1);
14                 P([k,r], :) = P([r,k], :);
15         end
16     v = k:n;
17     for ii = k+1:n
18         L(ii,k) = U(ii,k)/U(k,k);
19         U(ii,v) = U(ii,v) - L(ii,k)*U(k,v);
20     end
21 end
```

La gestión de los pivotes se consigue con las líneas 9–15, el resto es el mismo código empleado para la descomposición LU "pura".

Veamos su efecto paso a paso sobre

```
A = [1,2,1,-1; 2,2,-4,0; 1,4,1,3; 1,2,0,5]
```

Para ello, modificamos el código anterior de forma que nos muestre `U` después de la línea 15 y después de la línea 20. Es decir, cuando ha permutado las filas y cuando aplica reducción de Gauss. En la primera pasada los resultados son:

$$U_1 = \begin{pmatrix} 2 & 2 & -4 & 0 \\ 1 & 2 & 1 & -1 \\ 1 & 4 & 1 & 3 \\ 1 & 2 & 0 & 5 \end{pmatrix} \quad \rightarrow \quad U_2 = \begin{pmatrix} 2 & 2 & -4 & 0 \\ 0 & 1 & 3 & -1 \\ 0 & 3 & 3 & 3 \\ 0 & 1 & 2 & 5 \end{pmatrix}$$

Comprobamos que U_1 coincide con la matriz original salvo permutar las dos primeras filas, de este modo el 2, que es el mayor elemento de la primera columna, está en el lugar del primer pivote. La matriz U_2 muestra el resultado de la eliminación de Gauss con este pivote. Observamos que el elemento u_{32} es mayor que el u_{22}, que naturalmente actuaría como pivote en la descomposición LU "pura" , por tanto debemos intercambiar las filas segunda y tercera. Lo que resulta es:

$$U_1 = \begin{pmatrix} 2 & 2 & -4 & 0 \\ 0 & 3 & 3 & 3 \\ 0 & 1 & 3 & -1 \\ 0 & 1 & 2 & 5 \end{pmatrix} \quad \rightarrow \quad U_2 = \begin{pmatrix} 2 & 2 & -4 & 0 \\ 0 & 3 & 3 & 3 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 1 & 4 \end{pmatrix}.$$

En la tercera y última pasada no es necesaria ninguna permutación de filas porque $u_{33} > u_{43}$. En este caso el condicional `if` evita que se haga un

intercambio trivial.

ACTIVIDAD 3.4.2. *Define una función* `plu` *tal que* `[L,U,P]=plu(A)` *funcione como el comando correspondiente* `lu` *en* `matlab/octave`.

Se puede crear una descomposición con *pivotes totales*, lo que significa que se permite intercambiar filas y también intercambiar columnas con el fin de elegir como pivote un elemento aún mayor en valor absoluto aunque no esté en la columna seleccionada [19, §4.1]. Si nos atenemos a [15, p.37], es casi tan bueno como lo que hemos hecho y por tanto no merecen la pena las complicaciones que entraña. Apenas se usa en la práctica.

3.5. Apuntes sobre el rendimiento

Siguiendo este enlace de la ayuda de `matlab`:

`https://www.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html`

verás algunos consejos para mejorar el rendimiento. Dentro de "*Programming Practices for Performance*" lo que hemos hecho para perfeccionar el código inicial de la factorización *LU* está dentro del segundo consejo: "*Vectorize –Instead of writing loop-based code, consider using MATLAB matrix and vector operations*" y en parte del tercero: "*Place independent operations outside loops*". Ahora vamos a comprobar experimentalmente si nuestro código ha mejorado.

Para estudiar el rendimiento de un bloque de código, se suele encerrar entre los comandos `tic` y `toc` lo que muestra en la ventana de comandos un mensaje con el tiempo transcurrido entre ellos. Para comparar los algoritmos queremos una matriz grande de prueba. Con `rand(`n`)` se obtiene una aleatoria, en cierto sentido, de tamaño $n \times n$. Por ejemplo, si quisiéramos medir cuánta tarda nuestro ordenador en crear 100 matrices cuadradas de dimensión 2022, escribiríamos:

```
1 tic
2 for k = 1:100
3     A = rand(2022);
4 end
5 toc
```

Con ello obtengo como salida en mi ordenador:

```
Elapsed time is 3.29011 seconds.
```

Este tiempo presenta variaciones de acuerdo con la velocidad del ordenador e incluso con el momento en que se ejecute, por la carga no uniforme de la CPU.

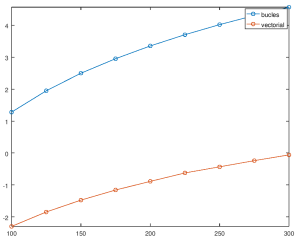
Si en vez de ver el resultado en la ventana de comandos, queremos guardar el tiempo transcurrido en una variable, es conveniente usar `t0 = tic` que

almacena en `t0` el tiempo en el que se llega a `tic` y `toc(t0)` que da el tiempo en el que se llega a `toc` menos `t0`, es decir, el tiempo transcurrido.

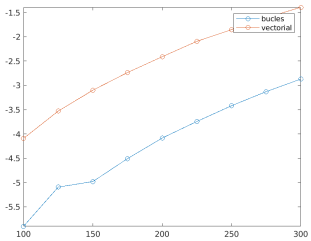
El siguiente código, guarda en los vectores `times1` y `times2` los valores del tiempo usando nuestro algoritmo inicial para la descomposición LU con bucles y el algoritmo optimizado quitando un bucle y definiendo `v`. Esos vectores se dibujan en escala semilogarítmica en una misma gráfica para compararlos. La gráfica se almacena en el fichero `figura.png`.

```
% Rango de dimensiones
D = 100:25:300;

times1 = [];
times2 = [];
for n = D
        A = rand(n);

        % Triple bucle
        t0 = tic;
        U = A;
        L = eye(n);
        for k = 1:n-1
                for ii = k+1:n
                        L(ii,k) = U(ii,k)/U(k,k);
                        for j = k:n
                                U(ii,j) = U(ii,j)
                                        ↪ -L(ii,k)*U(k,j);
                        end
                end
        end
        times1 = [times1, toc(t0)];

        % Vectorial
        t0 = tic;
        U = A;
        L = eye(n);
        for k = 1:n-1
                v = k:n;
                for ii = k+1:n
                        L(ii,k) = U(ii,k)/U(k,k);
                        U(ii,v) = U(ii,v) -L(ii,k)*U(k,v);
                end
        end
        times2 = [times2, toc(t0)];
end

plot(D, log(times1),'-o', D, log(times2),'-o')
legend('bucles', 'vectorial')
axis tight
saveas(gcf,'figura.png')
```

La única relativa novedad del código anterior es que los vectores `times1` y `times2` están al principio vacíos y se van añadiendo coordenadas dinámicamente. Aquí están dos de las figuras obtenidas:



`Octave`



`Matlab` *online*

Las cosas funcionan como un esperaría con `octave`, pero, sorprendentemente, con `matlab` *online* o `matlab` instalado, el algoritmo vectorial es más lento para dimensiones medianamente grandes (al menos con las últimas versiones).

No tengo una explicación clara para ello. Si se me permite un brindis al sol en la línea R. Stallman, un problema general con el *software* que no es libre, es que todavía es más difícil, si no imposible, saber qué está ocurriendo por debajo, limitando las posibilidades para mejorar nuestros programas. Sospecho que el intérprete de `matlab` tiene unos trucos para cambiar internamente de alguna forma los bucles (¿es posible que sea "inteligente" hasta el punto de que paralelice?). Aquí hay algunas explicaciones sobre algo relacionado:

`www.mathworks.com/matlabcentral/answers/54522-why-is-indexing-vectors-matrices-in-matlab-very-inefficient`

Hay una asimetría en `matlab/octave` entre filas y columnas respecto al rendimiento, debida a que la memoria es algo secuencial, las matrices bidimensionales a la postre se guardarán como tiras de números, y hay una diferencia apreciable entre modificar un montón de posiciones de memoria consecutivas y modificar otras que no lo son. El siguiente código, ilustra este punto:

```
N = 10000;
A = rand(N);
tic
for k = 1:N
    A(:,k) = 1;
end
toc

A = rand(N);
tic
for k = 1:N
    A(k,:) = 1;
end
toc
```

ACTIVIDAD 3.5.1. *Ejecuta el código anterior quizá cambiando el valor de* `N` *para que a tu ordenador le cueste al menos unas décimas de segundo. ¿Qué tiempo es mayor, el primero o el segundo? Intenta conjeturar a partir de ello si* `matlab/octave` *guarda internamente las matrices por filas o por columnas.*

Por si te interesa la historia de la informática, los dos grandes lenguajes que monopolizaron por mucho tiempo la computación científica, FORTRAN y C, seguían políticas diferentes en el orden de almacenamiento de los elementos de matrices.

ACTIVIDAD 3.5.2. *Modifica el algoritmo para factorización LU sin pivotes intercambiando los dos índices en L y en U, de forma que ofrezca las traspuestas de L y U. ¿Es el algoritmo en esta forma más rápido o más lento que el original para matrices grandes?*

Capítulo 4

Aproximaciones sucesivas y sistemas lineales

4.1. Normas con `matlab/octave`

El comando básico para hallar normas en `matlab/octave` es `norm`. Aplicado a un vector (fila o columna), da la norma usual euclídea. Funciona tanto para vectores reales como complejos. Así,

```
1 v = [1, -2, 10, 4];
2 disp( norm(v) )
3 v = [1+3*i; 5+i];
4 disp( norm(v) )
```

produce 11 y 6 como salida.

La norma usual, y de hecho cualquier norma en $\mathbb{R}^n$ o $\mathbb{C}^n$, se extiende a una norma de matrices por medio de

$$\|A\| = \max_{\|\vec{x}\|=1} \|A\vec{x}\|.$$

No hace falta que la matriz A sea cuadrada, es decir, las dos normas del segundo miembro pueden ser en espacios de distintas dimensiones. El comando `norm` aplicado a matrices da este resultado. No es difícil demostrar [19, §4.4] que $\|A\|$ admite una fórmula "explícita": coincide con la raíz cuadrada del mayor autovalor de $A^\dagger A$ (que es real y no negativo) donde $A^\dagger$ es la traspuesta conjugada.

ACTIVIDAD 4.1.1. *Sabiendo que* `eig(A)` *da la lista de autovalores de* `A`*, escribe un pequeño código que genere una matriz compleja al azar y comprueba la afirmación anterior.*

Respecto a otras normas que aparecen en análisis, añadiendo un argumento más a `norm`, con `norm(v,p)`, hallaremos la norma `p` de un vector `v`. Se permite `p=Inf` para indicar la norma infinito. Por ejemplo,

```
v = [1,2,-3,0];
disp( norm(v,1) )
disp( norm(v,4) )
disp( norm(v,Inf) )
```

ofrece como salida 6, $\sqrt[4]{98}$ (que está cerca de π) y 3. Por supuesto, `norm(v,2)` es lo mismo que `norm(v)`.

Por medio de la fórmula anterior, reemplazando $\|\cdot\|$ por $\|\cdot\|_p$, estas normas inducen unas correspondientes sobre las matrices, que se escriben en `matlab/octave` con el mismo comando, `norm(A,p)`. Es fácil ver [1, §7.3] que las normas de matrices correspondientes a las normas $\|\cdot\|_1$ y $\|\cdot\|_\infty$ en $\mathbb{R}^n$ o $\mathbb{C}^n$ admiten las fórmulas

$$\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{n} |a_{ij}| \qquad \text{y} \qquad \|A\|_\infty = \max_{1\le i\le n} \sum_{i=1}^{n} |a_{ij}|.$$

No es necesario que A sea cuadrada, es decir, la n puede ser distinta en ambas expresiones. Las siguientes líneas hacen una comprobación de estas fórmulas sobre una matriz de ejemplo:

```
A = [1,2;-3,-4;5,6*i];

no = norm(A,1)
am = max( sum(abs(A)) )
disp(['norm(A,1) = ' num2str(no) '. A mano = ' num2str(am)])

no = norm(A,Inf)
am = max( sum(abs(A')) )
disp(['norm(A,Inf) = ' num2str(no) '. A mano = '
      ↪ num2str(am)])
```

La clave para entender este código tan breve es recordar que `sum` aplicado a una matriz da la suma de las filas, es decir, el resultado es un vector fila cuya coordenada j coincide con la suma de la fila j.

Hay también un comando especial `norm(A,'fro')` que da la *norma de Frobenius*. Esta es la norma usual cuando se recolocan los elementos de la matriz para formar un vector. Es decir, es la raíz cuadrada de la suma de los valores absolutos al cuadrado de los elementos.

Una cosa a tener en cuenta es que `octave` permite calcular $\|A\|_p$ con `norm(A,p)` pero `matlab` solo permite $p = 1$ y $p = \infty$, aparte del $p = 2$ de la norma usual.

ACTIVIDAD 4.1.2. *De la definición se deduce que para matrices cuadradas A y B se cumple $\|AB\|_p \le \|A\|_p\|B\|_p$. Genera 1000 pares (A, B) de matrices aleatorias 3×3 con* `rand` *y calcula el máximo de $\|AB\|\|A\|^{-1}\|B\|^{-1}$. Si usas* `octave`*, halla este máximo para $\|AB\|_5\|A\|_5^{-1}\|B\|_5^{-1}$.*

4.2. Ejemplos de aproximaciones sucesivas

La aproximación numérica de las soluciones de una ecuación se lleva a cabo habitualmente con métodos iterativos. Más adelante en el curso,

trataremos estos métodos con algún detalle. Aquí solo veremos un par de ejemplos en una dimensión que nos den una idea como preparación al caso de sistemas de ecuaciones lineales.

Uno de los algoritmos más antiguos, quizá el que más, es un método iterativo para aproximar raíces cuadradas que responde a la fórmula:

$$x_{n+1} = \frac{1}{2}\Big(x_n + \frac{N}{x_n}\Big).$$

Aquí $N > 0$ es el número del que queremos calcular la raíz cuadrada. A pesar de la notación, no es necesario que sea un entero, aunque en ese caso las aproximaciones son números racionales, lo que tiene algún interés. Partiendo de cualquier $x_0 > 0$ se tiene que $x_n \to \sqrt{N}$. Además la convergencia es bastante rápida. Por ejemplo, para $N = 2$, partiendo de $x_0 = 1$ que es la aproximación más obvia de $\sqrt{2}$, el siguiente código

```
format long
x = 1;
for k = 1:6
        x = (x+2/x)/2
end
```

produce

```
x =   1.50000000000000
x =   1.41666666666667
x =   1.41421568627451
x =   1.41421356237469
x =   1.41421356237309
x =   1.41421356237309
```

Lo cual es bastante espectacular, en seis iteraciones ya se ha estabilizado a un valor indistinguible de $\sqrt{2}$ habida cuenta del épsilon máquina.

Actividad 4.2.1. *Busca una fórmula de recurrencia para el numerador y el denominador de las aproximaciones anteriores de $\sqrt{2}$ y escribe código que los calcule. Teniendo en cuenta que los enteros se almacenan en 64 bits, ¿hasta que iteración te deberías fiar de los resultados?*

Si damos por hecho que x_n, con $x_0 > 0$ converge (lo cual es cierto), entonces no es difícil probar que $x_n \to \sqrt{N}$ porque tomando límites en la fórmula de recurrencia $x_n \to \ell$ implica $2\ell = \ell + N/\ell$ y la única solución positiva de esta ecuación es $\ell = \sqrt{N}$.

La idea es entonces que siempre que la convergencia no nos fastidie, con $x_{n+1} = f(x_n)$ tendremos una sucesión que tiende a una solución de $x = f(x)$. Sin embargo, la convergencia sí se estropea a menudo aunque a veces reescribir la ecuación en una forma equivalente, soluciona el problema. Veámoslo en un ejemplo muy tonto. Supongamos que queremos resolver iterando la ecuación lineal $x = 3x - 8$, que tiene obviamente como solución $x = 4$. Despejando e intercambiando ambos miembros, podríamos reescribir la ecuación como $x = (x+8)/3$. Resulta que en la primera forma, no da lugar a un método convergente para todo valor de partida, pero en la segunda sí

(esto está relacionado con que el coeficiente de x en el segundo miembro sea mayor o menor que 1). El siguiente código ilustra este punto calculando unas pocas iteraciones de $x_{n+1} = 3x_n - 8$ y de $x_{n+1} = (x_n + 8)/3$ con $x_0 = 1$.

```
1 x = 1;
2 y = 1;
3 for k = 1:5
4           x = 3*x-8;
5           y = (y+8)/3;
6          disp(['primero -> ' num2str(x) ', segundo ->'
                 ↪ num2str(y)])
7 end
```

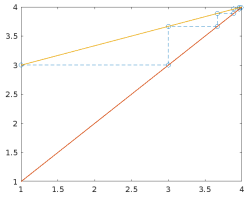
La salida es:

```
primero -> -5, segundo ->3
primero -> -23, segundo ->3.6667
primero -> -77, segundo ->3.8889
primero -> -239, segundo ->3.963
primero -> -725, segundo ->3.9877
```

La situación para sistemas de ecuaciones lineales es similar y hay algunos métodos que reescriben el sistema de forma que sea más fácil que satisfaga la condición de convergencia. Los dos más famosos son el de Jacobi y el de Gauss-Seidel, que veremos a continuación. Es importante señalar que no siempre funcionan, hay infinidad de sistemas para los que no convergen y su interés se debe a que tienen éxito en otra infinidad con relevancia práctica [16, §4.2].

ACTIVIDAD 4.2.2. *Para un ejemplo convergente $x_{n+1} = ax_n + b$, escribe un programa que dibuje las rectas $y = x$, $y = ax + b$ y la sucesión de puntos (x_0, x_1), (x_1, x_1), (x_1, x_2), (x_2, x_2), etc. unida con línea discontinua.*

Para el ejemplo lineal que hemos manejado antes, al resolver la actividad anterior deberíamos obtener algo del tipo:



4.3. El método de Jacobi

Ahora queremos resolver un sistema de ecuaciones lineales compatible determinado $A\vec{x} = \vec{b}$, con matriz cuadrada, y deseamos definir un esquema iterativo $\vec{x}^{(k+1)} = A'\vec{x}^{(k)} + \vec{b}'$ que converja a la solución. Como hablaremos a veces de coordenadas y utilizaremos n para el tamaño de la matriz, escribimos $\vec{x}^{(k)}$ en vez de $\vec{x}_n$.

En el método de Jacobi las iteraciones vienen dadas por:

$$\vec{x}^{(k+1)} = D^{-1}\big(\vec{b} - (A - D)\vec{x}^{(k)}\big)$$

donde $D = \mathrm{diag}(a_{11}, \ldots, a_{nn})$, es decir, es la matriz resultante al dejar solo la diagonal de A. Es un simple ejercicio comprobar que el método es consistente, es decir que si sustituimos $\vec{x}^{(k+1)}$ y $\vec{x}^{(k)}$ por la solución de $A\vec{x} = \vec{b}$, la ecuación es una identidad.

En sintonía con lo hecho antes con una ecuación lineal de una incógnita, se puede probar que el método converge si $D^{-1}(A - D)$ es "pequeño", en cierto sentido, y lo hará más rápido cuanto más pequeño sea. Por tanto, el método de Jacobi es muy eficiente aplicado a sistemas con matrices que sean "aproximadamente" diagonales.

La expresión en coordenadas del método es:

$$x_i^{(k+1)} = a_{ii}^{-1}\Big(b_i - \sum_{j=1,\, j\neq i}^{n} a_{ij} x_j^{(k)}\Big), \qquad i = 1, 2, \ldots, n.$$

A pesar de que veremos unos atajos con comandos de `matlab/octave` mediante los cuales no tendremos que usarla, es interesante tenerla en mente para entender otros métodos iterativos de resolución de sistemas lineales.

Tomemos como ejemplo el sistema $A\vec{x} = \vec{b}$ con

$$A = \begin{pmatrix} 5 & 2 & 1 \\ -3 & 6 & 1 \\ 2 & 3 & 5 \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 10 \\ 6 \\ 20 \end{pmatrix}, \quad \text{que tiene solución} \quad \vec{x} = \begin{pmatrix} 1 \\ 1 \\ 3 \end{pmatrix}.$$

En `matlab/octave` el comando `diag` aplicado a un vector, genera la matriz que tiene en la diagonal las coordenadas del vector y en el resto ceros. Cuando se aplica a una matriz, hace prácticamente lo contrario, devuelve el vector formado por la diagonal de la matriz. Por ejemplo

```
1 A = [5,2,1; -3,6,1; 2,3,5];
2 diag(A)
3 diag([1;1])
```

da lugar al vector columna $(5, 6, 5)$ y a la matriz identidad 2×2.

De esta forma, el $A - D$ anterior será `A - diag(diag(A))`. Valiéndonos de esto, en vez de programar el método con coordenadas, que sería lo más habitual, vamos a proceder con matrices. No le dejaremos calcular a `matlab/octave` la inversa D^{-1} para que no aplique algoritmos generales de la inversa que son lentos (en realidad "sabe" que es diagonal) y esa es la razón para definir `Dinv` en el siguiente código que aplica 5 iteraciones del método de Jacobi partiendo de $\vec{x} = \vec{0}$, en nuestro ejemplo:

```
1 N = 5;
2 A = [5,2,1; -3,6,1; 2,3,5];
3 b = [10;6;20];
4
5 n = size(A,1);
6 Ap = A - diag(diag(A));
7 Dinv = diag(A).^-1;
8
```

```
x = zeros(n,1);
for k = 1:N
        x = Dinv.*(b-Ap*x);
end
disp(x)
```

El resultado es $(0.99440, 1.01178, 2.98907)$, lo cual no es una mala aproximación de la solución.

ACTIVIDAD 4.3.1. *En el código anterior busca una matriz* `A` *para la que el método no converja y haz que muestre las normas de las sucesivas iteraciones.*

En la práctica sería muy incómodo ir probando con diferentes números de iteraciones hasta conseguir la precisión requerida. Es más interesante establecer un número máximo de iteraciones y un criterio de parada del algoritmo. Hay varias políticas posibles. Aquí utilizaremos (siguiendo [12]) que el algoritmo pare cuando $\|\vec{x}^{(k+1)} - \vec{x}^{(k)}\|/\|\vec{x}^{(k)}\|$sea menor que cierta tolerancia establecida. Esto quiere decir que el error relativo al aproximar $\vec{x}^{(k)}$ por $\vec{x}^{(k+1)}$ es menor que dicha tolerancia, lo que sugiere que se están acercando a un límite.

```
Nmax = 100;
tol = 0.1;
A = [5,2,1; -3,6,1; 2,3,5];
b = [10;6;20];

n = size(A,1);
Ap = A - diag(diag(A));
Dinv = diag(A).^-1;

x = zeros(n,1);
for k = 1:Nmax
        x_old = x;
        x = Dinv.*(b-Ap*x);
        err = norm(x-x_old)/norm(abs(x_old)+eps);
        if err < tol
                disp(['Se han usado ',num2str(k),'
                    ↪ iteraciones'])
                disp('La aproximación a la solución es:')
                disp(x)
                break
        end
end
```

Tal como está, con `tol = 0.1`, este código nos informa que solo se han usado 4 iteraciones (de las `Nmax` de que disponemos) y la aproximación será $(1.03733, 0.99333, 3.04133)$. Reduciendo la tolerancia a `tol = 1e-4`, se necesitan 10 iteraciones y la aproximación es $(1.00005, 0.99998, 3.00006)$.

Si creamos una función `mjac` respondiendo al prototipo:

```
function [x,err] = mjac (A, b, Nmax, tol)
        ...
end
```

que devuelva la aproximación de la solución con el criterio de parada anterior, así como el último error relativo estimado, lo podemos combinar con el siguiente código para decidir automáticamente si estamos satisfechos con el resultado o no.

```
1  Nmax = 20;
2  tol = 1e-4;
3  A = [5,2,1; -3,6,1; 2,3,5];
4  %A = [5,2,1; -3,1,1; 2,3,1];
5  b = [10;6;20];
6
7  [x,err] = mjac(A, b, Nmax, tol);
8
9  if err<tol
10         disp('La aproximación a la solución es:')
11         disp(x)
12 else
13         disp('No se consigue una aproximación')
14         disp('con los parámetros de partida')
15         disp(err)
16         if err> 0.1
17                 disp('El error es muy grande,')
18                 disp('es posible que el método no converja')
19         end
20 end
```

Ejecutándolo tal como está, nos mostrará $(1.00005, 0.99998, 3.00006)$, la aproximación antes mencionada. Si bajamos el número máximo de iteraciones con `Nmax = 5` nos avisará con el mensaje de las líneas 13–14. Si quitamos el `%` que comenta la línea 4 para activar la matriz con la que el método no converge, saltará el mensaje de las líneas 17–18.

ACTIVIDAD 4.3.2. *Escribe el código de la función* `mjac`.

4.4. El método de Gauss-Seidel

En términos de coordenadas, el método de Gauss-Seidel es:

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

donde i varía de 1 a n. En este caso, la representación matricial es un poco liosa y de interés principalmente teórico [19, §8.2].

Traduciendo la fórmula símbolo a símbolo a `matlab/octave` tenemos para el ejemplo de prueba anterior con 5 iteraciones el siguiente código. Se usa de nuevo `ii` en lugar de `i` por mi manía de no modificar la definición de la unidad imaginaria.

```
1  N = 5;
2  A = [5,2,1; -3,6,1; 2,3,5];
3  b = [10;6;20];
4
5  n = size(A,1);
6  x = zeros(n,1);
7
8  for k = 1:N
9          for ii = 1:n
10                 x(ii) = (b(ii) - A(ii,1:ii-1)*x(1:ii-1) -
                       ↪ A(ii,ii+1:n)*x(ii+1:n)
                       ↪ )/A(ii,ii);
11         end
12 end
13 disp(x)
```

Nótese que, en la fórmula que define el algoritmo, para $i = 1$ el primer sumatorio es vacío y para $i = n$, lo es el segundo. Deberíamos asegurarnos de que `matlab/octave` trata correctamente las sumas vacías como cero sin dar errores. El siguiente código lo ilustra:

```
A = [5,2,1; -3,6,1; 2,3,5];
x = [1;1;3];

rango = 1:3;
disp( A(1,rango)*x(rango) )

rango = 2:1;
disp( A(1,rango)*x(rango) )
```

Los resultados son 10 y 0. El segundo rango es vacío, si pedimos que nos los muestre, resultará `[](1x0)`.

Otra observación, es que se ha aprovechado la precedencia de las operaciones para escribir cosas como `x(1:ii-1)` en vez de la expresión más abigarrada equivalente `x(1:(ii-1))`. Los paréntesis son innecesarios porque primero se hace la resta `ii-1` y después se construye el vector.

ACTIVIDAD 4.4.1. *Escribe el código de una función* `mgase` *con una estructura similar a* `mjac` *pero con el algoritmo de Gauss-Seidel.*

4.5. Comparación numérica

Intuitivamente, el algoritmo de Gauss-Seidel debe ser mejor que Jacobi porque estamos actualizando parte del vector con coordenadas de la siguiente iteración que esperamos que sean más precisas. Bajo ciertas condiciones sobre la matriz (que en particular aseguran la convergencia) se puede demostrar que es así [19, §8.2], aunque no hay un teorema que lo asegure incondicionalmente. De hecho, como se ha visto en la teoría y repasaremos más adelante, es posible crear ejemplos especiales en los que el método de Gauss-Seidel no converja para todas las condiciones iniciales mientras que Jacobi sí lo haga.

Vamos a comprobar sobre nuestro ejemplo la precisión de ambos métodos. Para aprovechar las funciones que `mjac` y `mgase` que hemos construido, utilizaremos el siguiente código. Es importante notar que esto sería absurdo (muy ineficiente) para matrices grandes. Si no ves por qué, se explica a continuación. Todavía tienes una líneas de margen para pensarlo por ti mismo.

```
A = [5,2,1; -3,6,1; 2,3,5];
b = [10;6;20];
x_exact = [1;1;3];

for k = 1:10
        [x,err] = mjac(A, b, k, 0);
        e1 = norm(x-x_exact);
        [x,err] = mgase(A, b, k, 0);
        e2 = norm(x-x_exact);
```

```
10          disp( ['k = ' num2str(k) ' -> J: '
                   ↪ num2str(e1,'%.5e') ' -> G-S: '
                   ↪ num2str(e2,'%.5e')]  )
11  end
```

La ineficiencia antes mencionada radica en que al llamar a las funciones con los argumentos `(A, b, k, 0)` estamos forzando a que haga `k` operaciones (porque la tolerancia se ha puesto a cero y nunca se alcanza) pero no estamos aprovechando el resultado para obtener la función evaluada en `k+1`, que solo requeriría un paso más.

Por si no lo recuerdas o no lo sabes, el argumento `'%.5e'` de `num2str` indica que se muestren 5 cifras decimales en formato científico exponencial.

El resultado es:

```
k = 1 -> J: 1.41421e+00 -> G-S: 1.73205e+00
k = 2 -> J: 5.57773e-01 -> G-S: 2.14580e-01
k = 3 -> J: 1.35483e-01 -> G-S: 5.07620e-02
k = 4 -> J: 5.60952e-02 -> G-S: 4.80372e-03
k = 5 -> J: 1.70180e-02 -> G-S: 1.42026e-03
k = 6 -> J: 5.53404e-03 -> G-S: 1.17154e-04
k = 7 -> J: 2.14433e-03 -> G-S: 3.79722e-05
k = 8 -> J: 5.10085e-04 -> G-S: 3.35890e-06
k = 9 -> J: 2.23854e-04 -> G-S: 9.70198e-07
k = 10 -> J: 7.92543e-05 -> G-S: 1.06250e-07
```

Una condición suficiente típica que asegura la convergencia de los métodos de Jacobi y Gauss-Seidel es la de ser *diagonalmente dominante por filas*, lo que significa que cada $|a_{ii}|$ sea mayor o igual que la suma de los valores absolutos del resto de los elementos de la fila i-ésima. Sabiendo que `rand` genera matrices aleatorias con elementos entre 0 y 1, la siguiente línea genera matrices aleatorias diagonalmente dominantes por filas de tamaño `N`

```
1  A = rand(n) + (n-1)*eye(n)
```

ACTIVIDAD 4.5.1. *Subsana la ineficiencia del programa anterior para comparar los métodos en el ejemplo de prueba y estudia su efecto sobre* $A\vec{x} = \vec{b}$ *con* A *una matriz aleatorias diagonalmente dominantes por filas grande generada con las líneas anteriores y* $\vec{b}$ *generado con* `rand(n,1)`.

Por ejemplo, al resolver la actividad anterior, en una prueba para `n = 100` se obtuvo

```
k = 1 -> J: 1.69636e-02 -> G-S: 9.10007e-03
k = 2 -> J: 8.42881e-03 -> G-S: 1.11833e-03
k = 3 -> J: 4.19373e-03 -> G-S: 6.91222e-05
k = 4 -> J: 2.08657e-03 -> G-S: 1.09428e-05
k = 5 -> J: 1.03817e-03 -> G-S: 1.14720e-06
k = 6 -> J: 5.16536e-04 -> G-S: 8.70231e-08
k = 7 -> J: 2.57000e-04 -> G-S: 1.47590e-08
k = 8 -> J: 1.27870e-04 -> G-S: 1.10525e-09
k = 9 -> J: 6.36210e-05 -> G-S: 1.35638e-10
k = 10 -> J: 3.16544e-05 -> G-S: 1.72836e-11
```

Por supuesto, como los números son aleatorios, el resultado no se puede replicar exactamente pero da una idea de la situación típica.

La conclusión que debemos sacar, es que en las condiciones habituales de aplicación de los métodos, el de Gauss-Seidel es bastante mejor que el

de Jacobi. La rapidez de convergencia crece cuanto más dominante sea la diagonal y, en general, decrece con la dimensión.

Ahora vamos con una situación atípica. Un estudio del radio espectral (posiblemente llevado a cabo en la teoría) lleva a que la matriz

$$A = \begin{pmatrix} 1 & 1 & 2 \\ \beta & 1 & 1 \\ -5/2 & \beta^{-1} & 1 \end{pmatrix} \qquad \text{con} \quad \beta = \frac{5+\sqrt{21}}{2},$$

genéricamente da lugar a sistemas de ecuaciones lineales $A\vec{x} = \vec{b}$ para los que el método de Gauss-Seidel no converge mientras que el método de Jacobi lo hace para cualesquiera condiciones iniciales. Tomemos $\vec{b} = A(1,1,1)^t$ para que la solución sea $x = y = z = 1$ y veamos lo que ocurre con ambos métodos partiendo de $\vec{x} = \vec{0}$. El siguiente código, salvo cambios triviales, viene de concatenar códigos anteriores.

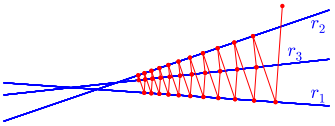
```
N = 40;

bet = (5+sqrt(21))/2;
A = [1,1,2; bet,1,1; -5/2,1/bet,1];
b = A*ones(3,1);

n = size(A,1);

% Jacobi
Ap = A - diag(diag(A));
Dinv = diag(A).^-1;
x = zeros(n,1);
for k = 1:N
        x = Dinv.*(b-Ap*x);
end
disp([num2str(N) ' iteraciones de Jacobi:'])
disp(x)

% Gauss -Seidel
n = size(A,1);
x = zeros(n,1);
for k = 1:N
        for ii = 1:n
                x(ii) = (b(ii) - A(ii,1:ii-1)*x(1:ii-1) -
                        ↪ A(ii,ii+1:n)*x(ii+1:n)
                        ↪ )/A(ii,ii);
        end
end
disp([num2str(N) ' iteraciones de Gauss-Seidel:'])
disp(x)
```

Cuando lo ejecutamos tal como está, con `N = 40`, se obtiene con el método de Jacobi `1.00037`, `1.00071`, `0.99972`, que está muy cerca de la solución. Sin embargo, el método de Jacobi produce las coordenadas astronómicas `-1.9115e+18`, `6.9862e+18`, `-6.2370e+18`.

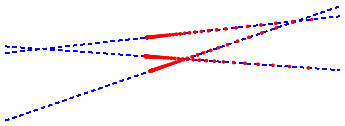
4.6. El algoritmo de Kaczmarz

Un algoritmo iterativo conceptualmente muy sencillo para resolver sistemas de ecuaciones lineales consiste en considerar cada una de las ecuaciones que lo componen como un hiperplano e ir haciendo proyecciones sucesivas en

cada uno de ellos. Si no has entendido esta frase, seguro que la primera de las siguientes figuras, correspondiente a dos incógnitas y por tanto hiperplanos que son rectas, te dará alguna luz.



Tres ecuaciones, dos incógnitas



Iteraciones en la zona central

La segunda figura muestra que es para sistemas que sean "ligeramente incompatibles" se obtiene algo que va oscilando entre soluciones aproximadas.

En un sistema real $m \times n$ la m-ésima ecuación es el hiperplano $\vec{a}_i \cdot \vec{x} = b_i$ donde $\vec{a}_i$ es la i-ésima fila considerada como vector columna y este vector es normal al hiperplano. Proyectar sobre el hiperplano es lo mismo que quitar la componente que está sobre $\vec{a}_i$ y si recordamos que la proyección de un vector sobre otro unitario venía dada por el producto escalar, no debería extrañarnos la fórmula del *algoritmo de Kaczmarz*.

Para sistemas con matrices reales $m \times n$, tal fórmula es:

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + \left(b_i - \vec{a}_i \cdot \vec{x}^{(k)}\right) \frac{\vec{a}_i}{\|\vec{a}_i\|^2}$$

donde i es el resto al dividir k por m más uno. Esta elección de i es solo una forma de decir que se van recorriendo cíclicamente todas las filas.

Para sistemas complejos hay que hacer algún pequeño cambio introduciendo conjugados porque resulta que el vector normal al hiperplano $a_1x_1 + \cdots + a_nx_n = b$ en $\mathbb{C}^n$ no es exactamente el dado por sus coeficientes sino por sus coeficientes conjugados.

Copiando el algoritmo tal como está, tenemos el siguiente código que lo aplica a un sistema de prueba 3×3 con solución $(1, 1, 2)$.

```
% Número iteraciones
N = 100
% Sistema
A = [2,2,1; 8,4,1; 4,7,-4];
b = [6; 14; 3];

% número de filas
m = size(A,1);
% número de comunas
n = size(A,2);

% Vector inicial
x = zeros(n,1);

for k = 1:N
        ii = mod(k,m) +1;
        fila = A( ii , : );
        x = x + (b(ii) - fila*x)*fila'/norm(fila)^2;
end

disp(x)
```

Con las 100 iteraciones indicadas se obtiene (`1.00222`, `0.99659`, `1.99589`), mientras que con 25 iteraciones dan (`1.12047`, `0.81596`, `1.77239`), lo que difícilmente nos haría sospechar la solución real. Cabe entonces preguntarse qué interés tiene un algoritmo que ya es lento en un sistema 3×3. Su gran virtud es su sencillez y su aplicabilidad a sistemas de dimensiones increíblemente grandes, intratables por otros métodos, que sean *dispersos*, esto es, con una matriz con muchos ceros, porque en ese caso $\vec{a}_i \cdot \vec{x}^{(k)}$ y $\|\vec{a}_i\|^2$ típicamente conllevarán muy poco esfuerzo computacional.

Un ejemplo práctico en el que se ha aplicado, aunque no es el método más común, es en tomografía axial computarizada (TAC). Allí cada uno de los rayos X atraviesa solo una proporción muy pequeña de los píxeles tridimensionales (*voxels*) de la muestra, lo que se relaciona con que cada ecuación contenga muy pocas incógnitas [3]. Por otro lado, la información es muy redundante y por tanto nos enfrentamos a un sistema teóricamente incompatible.

Es fácil percatarse de que el código anterior no está muy optimizado: cada m veces estamos pidiendo que calcule la misma norma. Una solución para no hacerlo sería normalizar las ecuaciones de manera que cada fila de la matriz tuviera norma 1. Una primera solución es hacer un bucle previo con este fin. En las versiones modernas de `matlab/octave` hay comandos como `vecnorm` y `normr` que nos pueden ayudar a simplificar y acelerar el proceso. Por otro lado, si en `v` almacenamos los inversos de las normas, en las versiones modernas `v.*A` no da error para multiplicar cada fila por el factor normalizante. En versiones antiguas la sintaxis es más rígida y requiere cosas como `bsxfun(@times, v.', A)`.

ACTIVIDAD 4.6.1. *Aplica alguna de las soluciones sugeridas u otra que se te ocurra para que* `norm(fila)` *no aparezca en el código anterior.*

Capítulo 5

Ortogonalización y descomposición QR

5.1. Ortogonalización

Algunos problemas de álgebra lineal se vuelven más sencillos si disponemos de bases ortogonales u ortonormales de un subespacio vectorial. Por ello, dado un conjunto de vectores linealmente independientes $\{\vec{v}_1, \ldots, \vec{v}_n\}$ tiene interés obtener $\{\vec{u}_1, \ldots, \vec{u}_n\}$ ortogonales o $\{\vec{q}_1, \ldots, \vec{q}_n\}$ ortonormales que generen el mismo subespacio. A esto es a lo que se refieren los palabros *ortogonalizar* y *ortonormalizar*. Evidentemente, el paso de los $\vec{u}_j$ a los $\vec{q}_j$ es tan fácil como *normalizar* mediante $\vec{q}_j = \vec{u}_j / \|\vec{u}_j\|$.

Casi todas las veces la simplificación proviene de que hallar coordenadas en una base ortonormal $\{\vec{q}_1, \ldots, \vec{q}_n\}$ no requiere resolver un sistema lineal, sino que basta con utilizar productos escalares. Concretamente, la solución de $x_1\vec{q}_1 + \cdots + x_n\vec{q}_n = \vec{v}$ es $x_i = \vec{v} \cdot \vec{q}_i$. El siguiente programa comprueba sobre un ejemplo, en el que la ortonormalización se reduce a normalización, que el resultado de proceder con productos escalares y resolviendo el sistema es el mismo.

```
1  v = [1,2,2,5]';
2
3  % vectores ortogonales (de hecho con la misma norma)
4  v1 = [4,5,6,7]';
5  v2 = [-7,-2,-3,8]';
6  v3 = [-5,-4,9,-2]';
7  v4 = [-6,9,0,-3]';
8
9  % (orto)Normalización
10 q1 = v1/norm(v1); q2 = v2/norm(v2); q3 = v3/norm(v3); q4 =
       ↪ v4/norm(v4);
11
12 % Con productos escalares
13 disp([v'*q1, v'*q2, v'*q3, v'*q4 ]')
14
15 % Resolviendo el sistema
16 A = [q1,q2,q3,q4];
17 disp(' ')
18 disp([ A\v ])
```

A todos nos han enseñado a ortogonalizar con un algoritmo llamado *proceso de Gram-Schmidt*, que recordaremos más adelante, pero no está implementado en `matlab/octave`, seguramente porque no es lo más eficiente desde el punto de vista numérico (en [15, §2.6] se llega a decir que es "terrible", aunque esto suena un poco exagerado). El comando directo de `matlab/octave` que ortonormaliza, con cierto algoritmo, las columnas de una matriz `A` es `orth(A)` y funciona incluso si tales columnas no son linealmente independientes, es decir, elimina de algún modo las innecesarias. Por ejemplo:

```
% Vectores de partida
v1 = [-2;1;0];
v2 = [5;0;1];
v3 = v1 + v2;
v4 = 2*v1 - 7*v2;

% Las columnas de A generan un subespacio W
A = [v1,v2,v3,v4];

% Las columnas del resultado son
% base ortonormal de W
orth(A)
```

da lugar a:

```
 0.982980   -0.020408
-0.048520    0.929684
 0.177188    0.367792
```

La tercera y la cuarta columnas de `A` son superfluas en cuanto al subespacio generado porque `v1` y `v2` ya lo generan. Sin embargo, `orth(A(:,1:2))` no da la misma base ortonormal.

El proceso de Gram-Schmidt antes mencionado, responde a las fórmulas:

$$\vec{u}_k = \vec{v}_k - \sum_{j=1}^{k-1} \langle \vec{v}_k, \vec{q}_j \rangle \vec{q}_j, \qquad \vec{q}_k = \frac{\vec{u}_k}{\|\vec{u}_k\|}$$

donde se sobreentiende que para $k = 1$ estamos tomando $\vec{u}_1 = \vec{v}_1$. Aquí hay una aclaración que hacer cuando se trabaja con números complejos y es que en álgebra lineal hay una tradición de definir el producto escalar usual entre vectores columna de $\mathbb{C}^n$ como $\langle \vec{a}, \vec{c} \rangle = \vec{a}^t \overline{\vec{c}}$ (lineal en el primer argumento) mientras que en casi el resto de las áreas se toma $\langle \vec{a}, \vec{c} \rangle = \overline{\vec{a}^t} \vec{c}$ (lineal en el segundo argumento). Esta segunda definición es más simple de implementar en `matlab/octave`, pues se escribiría `a'*c`, y, si la usamos, debemos cambiar $\langle \vec{v}_k, \vec{q}_j \rangle$ por $\langle \vec{q}_j, \vec{v}_k \rangle$ en la fórmula del proceso de Gram-Schmidt. Con estas precauciones, una posible implementación sobre un ejemplo es la siguiente, donde las columnas de `Q` son los vectores ortonormalizados, los $\vec{q}_j$.

```
A = [3,9; 6,4; 2,-1];

m = size(A,2);
Q = A;
Q(:,1) = A(:,1)/norm(A(:,1));

```

```
for k = 2:m
        u_k = A(:,k);
        for j = 1:k-1
                u_k = u_k - Q(:,j)' * A(:,k) * Q(:,j);
        end
        Q(:,k) = u_k/norm( u_k );
end
```

El resultado exacto para este ejemplo es $\vec{q}_1 = \frac{1}{7}(3,6,2)^t$, $\vec{q}_2 = \frac{1}{7}(6,-2,-3)^t$.

Para cualquier entrada, si escribimos `norm(Q'*Q-eye(m))` debe obtenerse un valor próximo a cero porque el elemento i,j de `Q'*Q` es el producto escalar (con la segunda definición) de la columna i por la columna j de `Q` y entonces simbólicamente resulta la matriz identidad. Por supuesto, el épsilon máquina evitará, en casos genéricos, que obtengamos un cero exacto.

ACTIVIDAD 5.1.1. *Compara con ejemplos aleatorios de cientos de coordenadas la rapidez de* `orth` *con respecto a la implementación anterior del proceso de Gram-Schmidt.*

ACTIVIDAD 5.1.2. *Escribe un programa que, dada una matriz $m \times n$, indique qué pares de sus columnas son ortogonales. Para evitar problemas con el épsilon máquina, introduce un parámetro de tolerancia para identificar productos escalares pequeños con cero.*

5.2. Las matrices de Householder

Aunque es muy posible que te lo ocultaran en la asignatura correspondiente de álgebra lineal, hay una fórmula sencilla para hallar la matriz de la simetría por un plano que contiene al origen con vector normal unitario $\vec{v}$. Esta fórmula es la *matriz de Householder* $I - 2\vec{v}\vec{v}^t$ donde, como es habitual, el vector $\vec{v}$ se considera en columna. Esto se extiende de $\mathbb{R}^3$ a $\mathbb{R}^n$ sustituyendo planos por hiperplanos. En particular, tenemos una forma sencilla de construir *matrices ortogonales*. Recuerda, que reciben este nombre las matrices reales no singulares que cumplen $A^{-1} = A^t$ o, equivalentemente, cuyas columnas forman una base ortonormal de $\mathbb{R}^n$. Además las matrices de Householder son simétricas. El siguiente código comprueba que la matriz es ortogonal con un ejemplo aleatorio en dimensión `N`.

```
N = 10

v = rand(N,1);
v = v/norm(v);
Q = eye(N)-2*v*v';
norm(Q'*Q-eye(N))
```

El resultado no es cero, por culpa del épsilon máquina. El truco también funciona con números complejos, siendo la matriz de Householder $I - 2\vec{v}\vec{v}^\dagger$ donde $\dagger$ indica la traspuesta conjugada, aunque tiene menos relevancia en cálculo numérico. La terminología al uso es llamar *matrices unitarias* a las

matrices no singulares complejas que cumplen $A^{-1} = A^\dagger$. De nuevo, esto equivale a que las columnas de A formen una base ortonormal de $\mathbb{C}^n$. Las matrices de Householder complejas son unitarias y además *hermíticas* (iguales a su traspuesta conjugada). El análogo complejo del código anterior solo requiere modificar la tercera línea:

```
1  N = 10
2
3  v = rand(N,1)+i*rand(N,1);
4  v = v/norm(v);
5  Q = eye(N)-2*v*v';
6  norm(Q'*Q-eye(N))
```

Si alguna vez impartes clases de álgebra lineal, la siguiente actividad te puede resultar útil. Si quieres romper la simetría de las matrices ortogonales obtenidas, puedes multiplicar dos resultados porque el producto de matrices simétricas no es, en general, una matriz simétrica.

ACTIVIDAD 5.2.1. *Escribe un programa que genere un vector aleatorio no nulo en dimensión N con coordenadas aleatorias en $[-k, k] \cap \mathbb{Z}$ y utiliza las matrices de Householder para obtener una matriz ortogonal de la forma $d^{-1}A$ con $d \in \mathbb{Z}^+$ y $A \in \mathcal{M}_{N\times N}(\mathbb{Z})$.*

Una observación muy útil para lo que viene después, es que dado un vector $\vec{x}$ es fácil construir una matriz de Householder que al ser aplicada a este vector anula todas sus coordenadas excepto la primera. Basta tomar

$$\vec{v} = \frac{\vec{w}}{\|\vec{w}\|} \qquad \text{con} \quad \vec{w} = \vec{x} \pm \|\vec{x}\|\vec{e}_1$$

donde $\vec{e}_1$ es el vector $(1, 0, 0, \ldots, 0)^t$ y el signo $\pm$ lo podemos elegir a nuestra voluntad. El siguiente código muestra que esto funciona con el signo positivo:

```
1  N = 4;
2  x = 2*rand(N,1)-1;
3  v = x;
4  v(1) = v(1)+norm(x);
5  v = v/norm(v);
6  Q = eye(N)-2*v*v';
7  Q*x
```

A estas alturas, seguro que sabes que hay cierto peligro en cálculo numérico al restar dos números próximos, por tanto es buena política elegir el signo positivo si $x_1 \geq 0$ y el signo negativo si $x_1 < 0$.

ACTIVIDAD 5.2.2. *Escribe una función llamada* `vh` *tal que* `vh(x)` *dé el vector unitario* `v` *correspondiente a* `x` *siguiendo la política de signos anterior.*

5.3. La descomposición QR

A pesar de que la terminología no sea muy esclarecedora, la descomposición QR de una matriz es casi equivalente a la ortonormalización de sus

columnas. La Q se refiere a una matriz cuyas columnas son ortonormales (y, por tanto, si es cuadrada es ortogonal o unitaria) y R es una matriz triangular superior, quizá completada con ceros.

Si recordamos las fórmulas del proceso de Gram-Schmidt, obtendremos la Q y la R con unos ligeros añadidos en el código que almacenen los coeficientes en la R. En una función:

```
% Descomposición QR (reducida) con Gram-Schmidt
function [Q,R] = mQR(A)
        m = size(A,2);
        Q = A;
        R = zeros(m);
        R(1,1) = norm(A(:,1));  % nueva
        Q(:,1) = A(:,1)/R(1,1);

        for k = 2:m
                u_k = A(:,k);
                for j = 1:k-1
                        R(j,k) = Q(:,j)' * A(:,k);  % nueva
                        u_k = u_k - R(j,k) * Q(:,j);
                end
                R(k,k) = norm( u_k );  % nueva
                Q(:,k) = u_k/R(k,k);
        end
end
```

Las líneas marcadas como nuevas son solo definiciones para que se almacene el valor de los elementos de R.

El comando nativo de `matlab/octave` para la descomposición QR es `qr`. Si ejecutamos

```
A = [-1,-5,-10; -2,2,-8; -2,-4,-5];

[Q,R] = qr(A)
[Q,R] = mQR(A)
```

veremos que, sobre este ejemplo, nuestra función y la de `matlab/octave` dan lo mismo. La unicidad para matrices cuadradas no singulares está asegurada imponiendo que los elementos de la diagonal de R sean positivos.

Sin embargo, cuando usamos el ejemplo:

```
A = [1,1,2; 1,5,1; 1,1,1; 1,5,0];

[Q,R] = mQR(A)
[Q,R] = qr(A)
```

vemos una gran diferencia. Nuestra función da

$$Q = \frac{1}{2}\begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & -1 & -1 \\ 1 & 1 & 1 \end{pmatrix} \in \mathcal{M}_{4\times 3} \qquad \text{y} \qquad R = \begin{pmatrix} 2 & 6 & 2 \\ 0 & 4 & -1 \\ 0 & 0 & 1 \end{pmatrix} \in \mathcal{M}_{3\times 3}.$$

Es lo que en la teoría se ha llamado la descomposición QR *reducida* en la que R es cuadrada, pero Q no necesariamente.

La de `matlab/octave` produce siempre una Q cuadrada (ortogonal o unitaria), lo cual es más útil en las aplicaciones más importantes. Además,

no impone $r_{ii} \geq 0$, lo cual afecta a la unicidad. Su resultado en el ejemplo es:

$$Q = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ -1 & 1 & -1 & -1 \\ -1 & -1 & -1 & 1 \end{pmatrix} \in \mathcal{M}_{4\times 4} \quad \text{y} \quad R = \begin{pmatrix} -2 & -6 & -2 \\ 0 & -4 & -1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \in \mathcal{M}_{4\times 3}.$$

La eficiencia de nuestra función para matrices cuadradas no es muy buena en comparación con `qr` de `matlab/octave`. Consideremos

```
N = 100;
A = rand(N);

disp('Nuestra función')
tic
[Q,R] = mQR(A);
norm(Q*R-A)
toc

disp('')
disp('La de matlab/octave')
tic
[Q,R] = qr(A);
norm(Q*R-A)
toc
```

Tal como está, una ejecución en mi ordenador con `octave` dio `0.0798371` segundos con nuestra función basada en Gram-Schmidt y `0.00107002` con `qr`. Es decir, `qr` fue 70 veces más rápido. Al cambiar la primera línea a `N = 500` los tiempos pasaron a ser `1.98888` y `0.0308831` que todavía es 60 veces más rápido. Los errores cometidos al aproximar `Q*R` por `A`, que debieran ser iguales, son comparables, aunque resultaron menores con Gram-Schmidt.

ACTIVIDAD 5.3.1. *Haz un programa que muestre la gráfica del tiempo requerido por nuestra función dividido por el requerido por* `qr` *para una matriz aleatoria* $N \times N$ *cuando* N *varía entre* 100 *y* 500 *de* 10 *en* 10. *Por alguna razón los resultados son más exagerados en* `octave` *que en* `matlab`.

ACTIVIDAD 5.3.2. *Estudia qué ocurre con los autovalores de* `Q'*Q` *y de* `Q*Q'` *en nuestra descomposición* QR *cuando la matriz de partida es* $n \times m$ *con* $m < n$. *Como se ha mencionado en la teoría, el comando* `eig(A)` *devuelve los autovalores de la matriz* `A`.

Ahora vamos a implementar la descomposición QR con matrices de Householder. Para ello utilizamos la función `vh(x)` que daba un vector tal que la correspondiente matriz de Householder aplica `x` en un múltiplo de $\vec{e}_1$. El algoritmo consiste en ir considerando sucesivamente como vector `x` los elementos de la matriz por debajo o en la diagonal para cada una de las columnas, con ello la matriz de partida, supuesta cuadrada, se irá transformando en una triangular superior. En una función, esto es:

```
function [Q,R] = mQR2(A)
```

```
2           n = size(A,1);
3           Q = eye(n);
4           R = A;
5           for k = 1:n-1
6                   x = R(k:n,k);
7                   v = zeros(n,1);
8                   v(k:n) = vh(x);
9                   % dim Qt = n. Es la matriz de Householder
                        ↪ completada
10                  % con k-1 unos porque v(1:k-1) es nulo
11                  Qt = eye(n)-2*v*v';
12                  Q = Qt*Q;
13                  R = Qt*R;
14                  % Para que sea exactamente triangular
15                  R(k+1:n,k) =zeros(n-k,1);
16          end
17          % Esto es lo mismo que la inversa
18          % porque Q es ortogonal o unitaria
19          Q = Q';
20  end
```

Las líneas 12 y 13 son en realidad derrochadoras, porque solo necesitamos hacer esta operación para un bloque de la matriz y no necesitamos construir toda la `Qt`. Se deja como ejercicio, perfeccionar el código.

Si lo aplicamos a una matriz no cuadrada dará la descomposición QR *completa*, es decir, con Q cuadrada (ortogonal o unitaria), como el comando nativo `qr` de `matlab/octave`. De hecho, el siguiente código muestra que sobre la matriz no cuadrada considerada antes, se obtiene lo mismo:

```
1  A = [1,1,2; 1,5,1; 1,1,1; 1,5,0];
2  [Q,R] = qr(A)
3  [Q,R] = mQR2(A)
```

Actividad 5.3.3. *Repite la comparativa anterior para matrices aleatorias $N \times N$ cuando N varía entre 100 y 500 de 10 en 10, pero ahora con la nueva función* `mQR2`.

5.4. Aplicación a la resolución de sistemas

Si $A = QR$ es la descomposición QR (completa) de una matriz no singular A, como Q es ortogonal o unitaria, se tiene $Q^{-1} = Q^\dagger$ y por tanto el sistema $A\vec{x} = \vec{b}$ equivale a $R\vec{x} = Q^\dagger\vec{b}$, el cual es muy sencillo porque R es triangular superior al que se puede aplicar sustitución regresiva, que es muy rápida. Por otro lado, calcular la descomposición QR no es gratis y la teoría dice que LU conllevará asintóticamente $2n^3/3$ para resolver un sistema genérico $n \times n$ y QR requerirá el doble. En realidad, tanto con LU como con QR la parte de sustitución regresiva o progresiva es menor en comparación con lo que tardan las descomposiciones. Por tanto, está más cerca de una confirmación experimental, un código del tipo:

```
1  N = 3000
2  A = rand(N);
3  b = rand(N,1);
4
5  tic
6  [L,U,P] = lu(A);
```

```
7  toc
8
9  tic
10 [Q,R] = qr(A);
11 toc
```

Realmente se obtienen de esta forma factores más grandes que dos, en `octave` mucho mayores que en `matlab`. No tengo explicación para ello.

ACTIVIDAD 5.4.1. *Modifica ligeramente el código de la actividad anterior para haga la comparativa de tiempos entre* `lu` *y* `qr` *cuando* `N` *está en el rango* `1000:100:3000`.

Recuerda que, si has sido obediente u ordenado, tienes implementadas unas funciones de días anteriores `ltrs` y `utrs` para resolver, con sustitución progresiva y regresiva, un sistema lineal con matriz triangular inferior y superior, respectivamente. Según lo dicho, la manera de resolver un sistema con QR sería del tipo:

```
1 N = 3
2 A = rand(N);
3 b = A*ones(N,1);
4
5 [Q,R] = qr(A);
6 x = utrs(R,Q'*b);
7
8 norm(x-ones(N,1))
```

El algoritmo en sí son las líneas 5 y 6. Las primeras fuerzan a que la solución tenga todas sus coordenadas uno y la última calcula el error, que para `N` pequeño debe ser comparable al épsilon máquina.

ACTIVIDAD 5.4.2. *Aunque sea una tarea retrospectiva, escribe un código que resuelva un sistema con la descomposición LU usando* `lu`. *Solo lo habíamos hecho para la función que habíamos construido nosotros. La única dificultad está en el uso de la matriz de permutación con los pivotes. Comprueba que funciona viendo que* `norm(x-A\b)` *es pequeño.*

Capítulo 6

Valores singulares y optimización

6.1. Soluciones de mínimos cuadrados

Una de las aplicaciones más interesantes de la descomposición QR es que permite resolver aproximadamente sistemas incompatibles, es decir, que no tienen solución. Ya vimos que la barra invertida de `matlab/octave` lo hacía. Cambiando un poco el ejemplo visto entonces,

```
1 A = [1,2; 1,1; 3,1];
2 b = [2;8;6];
3
4 disp('solución matlab/octave')
5 A\b
```

da como salida el vector $(2,1)^t$ a pesar de que $A\vec{x} = \vec{b}$ no tiene solución.

La clave teórica es que, usando propiedades de la proyección ortogonal, si $A \in \mathcal{M}_{n\times m}(\mathbb{C})$ con $m < n$ tiene rango máximo (igual a m), entonces

$$\|\vec{b} - A\vec{x}\| \quad \text{se minimiza para} \quad \vec{x} = (A^\dagger A)^{-1} A^\dagger \vec{b}.$$

Con la norma usual, esta es la *solución de mínimos cuadrados*. La matriz $(A^\dagger A)^{-1}A^\dagger$ funciona como una especie de "inversa aproximada" de la matriz no cuadrada A. Se la conoce como la *inversa de Moore-Penrose* o también, a veces, *pseudoinversa*.

El punto a destacar es que la descomposición QR simplifica estos cálculos. Para A como antes, la descomposición QR completa será de la forma

$$A = QR \quad \text{con} \quad Q \in \mathcal{M}_{n\times n}(\mathbb{C}) \quad \text{y} \quad R = \begin{pmatrix} T \\ O \end{pmatrix} \in \mathcal{M}_{n\times m}(\mathbb{C}),$$

donde $T \in \mathcal{M}_{m\times m}(\mathbb{C})$ es triangular superior. Entonces, usando $Q^\dagger Q = I$,

$$(A^\dagger A)^{-1} A^\dagger \vec{b} = (R^\dagger R)^{-1} R^\dagger Q^\dagger \vec{b} = (T^\dagger T)^{-1} T^\dagger Q_m^\dagger \vec{b} = T^{-1} Q_m^\dagger \vec{b}$$

donde Q_m son las m primeras columnas de Q. Invertir una matriz triangular superior es muy sencillo. Incluso pasando este importante punto por alto, las tres expresiones anteriores para $(A^\dagger A)^{-1}A^\dagger\vec{b}$ constituyen simplificaciones del cálculo directo. El siguiente código recoge estas posibilidades:

```
A = [1,2; 1,1; 3,1];
b = [2;8;6];

disp('solución matlab/octave')
A\b

[Q,R] = qr(A);
m = size(A,2);
T = R(1:m,:);
Qm = Q(:,1:m);
% Solución mín. cuad -> inv(A'*A)*A'*b

disp('solución QR simplificada 1')
inv(R'*R)*R'*Q'*b

disp('solución QR simplificada 2')
inv(T'*T)*T'*Qm'*b

disp('solución QR simplificada 3')
inv(T)*Qm'*b
```

Por supuesto, lo más eficiente es calcular $T^{-1}Q_m^\dagger\vec{b}$ mediante sustitución regresiva como solución del sistema $T\vec{x} = Q_m^\dagger\vec{b}$. Recuerda que la resolución de sistemas con matrices triangulares superiores por medio de sustitución regresiva fue parte de una actividad anterior. Por si no tienes el código a mano, una posibilidad es:

```
function x = utrs(A, b)
    % Resuelve sistema con matriz triangular superior
    n = size(A,1);
    x = zeros(n, 1);
    x(n) = b(n)/A(n,n);
    for ii = n-1:-1:1
        x(ii) = (b(ii) - A(ii,(ii+1):n)*x((ii+1):n)
                ↪ )/A(ii,ii);
    end
end
```

El uso de la descomposición QR para hallar soluciones aproximadas de sistemas incompatibles no es un artificio académico. Si miras la documentación o [17, §5.6], verás que la barra inversa de `matlab` opera de esta forma en los casos incompatibles.

ACTIVIDAD 6.1.1. *Escribe un código eficiente que halle la solución de mínimos cuadrados llamando a* `utrs`.

Si m es muy pequeño, puede no merecer la pena meterse en la descomposición QR. Un ejemplo destacado con $m = 2$ es la recta de regresión [5]. La situación es que tenemos una lista de datos formando un vector $\vec{y} \in \mathbb{R}^n$ que queremos ajustar linealmente con unos datos de partida $\vec{x}$. Es decir, buscamos un a y un b tales que y_i sea aproximadamente $a + bx_i$, la recta $y = a+bx$ es la *recta de regresión*. Introduciendo la matriz X de dimensiones $n \times 2$ que tiene como primera columna unos y como segunda columna $\vec{x}$,

lo que buscamos es un $\vec{\beta} \in \mathbb{R}^2$ tal que $\vec{y} \approx X\vec{\beta}$ con $\vec{\beta} = (a, b)^t$. Según lo anterior, la solución de mínimos cuadrados es $\vec{\beta} = (X^tX)^{-1}X^t\vec{y}$.

El siguiente código hace el cálculo y muestra gráficamente el resultado sobre un ejemplo en el que $\vec{y}$ es una perturbación del doble de $\vec{x}$.

```
1  N = 6
2  x = (1:N)';
3  y = 2*x + 4*rand(N,1);
4
5  n = size(x,1);
6  X = [ones(n,1), x];
7
8  c = inv(X'*X)*X'*y;
9
10 plot(x,y,x,c(1)+c(2)*x)
```

ACTIVIDAD 6.1.2. *Escribe una función* `rrerr` *que dados dos vectores de datos* `x` *e* `y` *devuelva el error absoluto máximo, sin signo, cuando se aproxima* `y` *por la recta de regresión. ¿Se cumple* `rrerr(x,y)=rrerr(y,x)` *en general?*

6.2. SVD y optimización

Recuerda de la teoría que la descomposición en valores singulares SVD (por *Singular Value Decomposition*) de una matriz A es la factorización $A = UDV^\dagger$ donde U y V son matrices unitarias (su inversa coincide con su traspuesta conjugada, indicada mediante $\dagger$) y D es una matriz cuyos únicos elementos no nulos son $d_{ii} \geq 0$, estos son los llamados *valores singulares* que dan nombre a la descomposición. En `matlab/octave`, el comando `svd`, utilizado como en el siguiente ejemplo, permite obtener la SVD.

```
1  A = rand(10,7);
2  [U,D,V] = svd(A);
3  norm(A-U*D*V','fro')
```

Como es evidente por la primera línea, la descomposición no requiere que la matriz sea cuadrada. La última línea es la comprobación de que el error es pequeño. Se ha escogido la norma de Frobenius solo para recordar el comando.

Si se llama a `svd` sin recuperar la salida con tres matrices, se obtiene solo el vector de valores singulares. Así las líneas 4 y 7 del siguiente código muestran el mismo resultado.

```
1  A = rand(3,5);
2  [U,D,V] = svd(A);
3  disp('Completa')
4  disp( diag(D) )
5  disp('Solo valores singulares')
6  v = svd(A);
7  disp(v)
```

Parte del interés de la SVD es que permite resolver algunos problemas de optimización. En particular, en la teoría has visto que la solución de

mínimos cuadrados de un sistema incompatible $A\vec{x} = \vec{b}$ viene dada por

$$\vec{x} = \sum_{k=1}^{r} \sigma_k^{-1} \langle \vec{b}, U^{(k)} \rangle V^{(k)}$$

donde r es el rango, el superíndice (k) indica la columna k-ésima y σ_k son los valores principales. El producto escalar es el usual, definido en el caso complejo con el convenio del álgebra lineal. Si $A \in \mathcal{M}_{n\times m}(\mathbb{C})$ con $r = m < n$, la situación típica en la que nos habíamos fijado, esto es lo mismo que $VE(U_m)^\dagger \vec{b}$ donde E es la matriz diagonal $m \times m$ que tiene $e_{ii} = \sigma_i^{-1}$ y U_m indica que nos quedamos solo con las primeras m columnas. Veamos en un ejemplo que se obtiene lo mismo que ya obtuvimos con QR o con la barra de `matlab/octave`.

```
A = [1,2; 1,1; 3,1];
b = [2;8;6];
[n,m] = size(A);
[U,D,V] = svd(A);
E = diag( diag(D) );
V*inv(E)*U(:,1:m)'*b
A\b
```

Las dos últimas líneas producen una salida idéntica, $(2, 1)^t$. Invertir `E` no le cuesta esfuerzo a `matlab/octave` porque sabe que es una matriz diagonal. Si trabajáramos con un lenguaje de programación sin esta estructura de datos, lo eficiente será multiplicar las columnas de V con los σ_k^{-1}.

ACTIVIDAD 6.2.1. *Comprueba si lo anterior da el mismo resultado que* `A\b` *también cuando trabajamos con números complejos. Para ello, parte de una matriz* `A` *aleatoria compleja* `2N` $\times$ `N` *y de un vector* `b` *de dimensión* `2N` *y estudia si ambos resultados tienen una diferencia despreciable.*

Otra aplicación de la SVD relacionada con la optimización es el *teorema de Eckart-Young* (a veces se añade el nombre de Mirsky) que afirma que si tenemos una matriz A, la mejor aproximación A_r de rango $r \leq \mathrm{rg}(A)$, en el sentido de que la norma de Frobenius de $A - A_r$ es mínima, se obtiene mediante $A_r = \sum_{k=1}^{r} \sigma_k U^{(k)} (V^{(k)})^\dagger$. Esto es lo mismo que anular en la SVD de A todos los valores singulares de índice mayor que r. Normalmente el resultado se enuncia para matrices reales pero el caso complejo es similar y está esencialmente hecho en [13].

El siguiente código muestra un ejemplo:

```
A = [1,2,3; 1,1,2; 3,1,4; 3,1,4];

r = 2;
[U,D,V] = svd(A);
for k=r+1:min(size(A))
        D(k,k)=0;
end

Ar = U*D*V';

disp(Ar)
disp(norm(A-Ar,'fro'))
```

Como en este ejemplo la matriz de partida tiene rango 2, se obtiene `Ar` igual a `A`. Si cambiamos la tercera línea por `r = 1`, obtendremos una matriz con todas sus filas proporcionales.

ACTIVIDAD 6.2.2. *Para una matriz real aleatoria de tamaño* 100×200 *fijada, dibuja una gráfica que muestre cómo cambia la norma de Frobenius de* $A - A_r$ *cuando* $1 \leq r < 100$.

Un tema relacionado es que, en este mismo sentido, si $A = UDV^\dagger$ es la SVD de una matriz cuadrada A, entonces $UV^\dagger$ es la matriz unitaria que mejor aproxima a A, de nuevo en el sentido de la norma de Frobenius. El siguiente código ejemplifica esto gráficamente. Se dibuja la circunferencia unidad centrada en $(2, 0)$, se le aplica una transformación lineal que la convierte en una elipse y $UV^\dagger$ es un giro que superpone la circunferencia y la elipse.

```
A = [0.546, -0.722; 0.783, 0.427];
t = linspace(0,2*pi,300);
x = [2+cos(t); sin(t)];
y = A*x;

[U,D,V] = svd(A);
z = U*V'*x;

figure(1)
plot(x(1,:), x(2,:))
hold on
plot(y(1,:), y(2,:))
hold on
plot(z(1,:), z(2,:))
hold off
axis('equal')
```

ACTIVIDAD 6.2.3. *Modifica el código anterior para que la figura de partida sea el cuadrado definido por la frontera de* $[2, 3] \times [0, 1]$.

Las matrices U y V de la SVD tienen también una interpretación natural dentro de la teoría básica del álgebra lineal. Limitándonos al caso $A \in \mathcal{M}_{m\times n}(\mathbb{C})$ con $m < n$ y rango igual a m, las m columnas de U forman una base ortonormal de la imagen y las $n - m$ últimas columnas de V forman una base ortonormal del núcleo. El siguiente código comprueba estas afirmaciones sobre una matriz compleja aleatoria.

```
M = 4;
N = 7;
% Matriz aleatoria compleja MxN
A = rand(M,N) + i*rand(M,N);
[U,D,V] = svd(A);
% Últimas N-M columnas de V
disp( 'Base ortonormal del núcleo' )
disp( V(:,M+1:N) )
% Comprobación de que están en el núcleo
% y de que son ortonormales
disp(' Estas cantidades deben ser pequeñas' )
disp( norm( A*V(:,M+1:N) ) )
disp( norm(V(:,M+1:N)'*V(:,M+1:N)-eye(N-M)) )
```

En la teoría no has visto ningún algoritmo eficiente para calcular la SVD

que podamos implementar (si tienes curiosidad, mira [19, §6.7]). Sin embargo, sí te han contado un método para hallarla "a mano" diagonalizando $A^\dagger A$, aunque no es muy eficiente. En resumen, la diagonalización da V y U es una matriz unitaria que cumple $AV = UD$.

ACTIVIDAD 6.2.4. *Implementa en* `matlab/octave` *el algoritmo anterior usando que* `[V,D] = eig(S)` *da la diagonalización* $S = VDV^\dagger$ *con* V *unitaria para una matriz hermítica* S*. Halla* U *y* D *con* `qr` *y trata de justificar por qué sabemos que la* D *hallada de esta forma debe tener ceros fuera* d_{ii}*, lo cual va más allá de ser triangular (en rigor, trapezoidal) superior.*

Capítulo 7

Valores y vectores propios

7.1. El comando `eig`

Antes de nada, recordemos que el comando nativo en `matlab/octave` para hallar valores y vectores propios es `eig`. Si ejecutamos directamente `eig(A)` o recuperamos su salida con una variable, se obtiene un vector columna cuyas coordenadas son los autovalores (repetidos con sus multiplicidades), mientras que si especificamos dos variables de salida con `[C,D]=eig(A)`, obtendremos las matrices que corresponden a la diagonalización $A = CDC^{-1}$. Por la teoría de álgebra lineal, la diagonal de D estará compuesta por los autovalores y las columnas respectivas de C por los autovectores.

En lo sucesivo usaremos como ejemplo las matrices A_1, A_2 y A_3 definidas con `matlab/octave` como:

```
% Autovalores: 4, 1\pm i, 1
A1 = [-7,6,-8,5; -16,10,-13,10; -5,2,-2,3; -8,6,-8,6];

% Autovalores: 4, 3, 2, 1
A2 = [-1,3,-5,2; -7,7,-7,4; -1,1,1,0; -2,3,-5,3];

% Autovalores: 8, 6, 4, 2
A3 = [5,0,-2,1; 0,5,-1,2; -2,-1,5,0; 1,2,0,5];
```

La ordenación de los autovalores puede no ser la indicada. Así en una máquina con `octave` el comando `eig(A2)` produjo $1, 2, 4, 3$. El comando `sort` sirve para ordenar y por tanto `sort(eig(A2))` los muestra en orden creciente mientras que con `sort(eig(A2),'descend')` el orden es decreciente.

Para A_1 y A_2 es posible elegir C con elementos enteros, pero como `eig` normaliza, con este comando no nos percataremos de ello. La matriz A_3 es simétrica y por tanto diagonaliza en una base ortonormal.

Incidiendo en lo dicho antes, una forma de presentar los autovalores y autovectores de A_3 es:

```
A3 = [5,0,-2,1; 0,5,-1,2; -2,-1,5,0; 1,2,0,5];
[C,D]= eig(A3);
for k = 1:4
        disp('')
```

```
5           disp(['autovalor ' num2str(k) ': ' num2str(D(k,k))])
6           disp('autovector respectivo (normalizado):')
7           disp(C(:,k))
8   end
```

Si dentro del bucle calculamos `norm(A3*C(:,k)-D(k,k)*C(:,k))`, veremos que es depreciable, lo que prueba que los vectores propios verifican la ecuación $A\vec{v} = \lambda\vec{v}$ que los definen.

Una cuestión a tener en cuenta es que, desde el punto de vista numérico, todas las matrices parecen diagonalizables sobre $\mathbb{C}$ porque cada matriz no diagonalizable está infinitamente cerca en norma de una infinidad de ellas que sí lo son. Recordando del álgebra lineal que si una matriz no es diagonalizable sobre $\mathbb{C}$ necesariamente tiene autovalores múltiples, no es de extrañar que un enemigo fundamental del cálculo numérico de autovalores y autovectores, ya sea usando `eig` o cualquier otro algoritmo, es la existencia de multiplicidades.

Por ejemplo, consideremos el sencillo código:

```
1  A = [10,1;0,10];
2  [C,D]= eig(A);
3  disp('autovalores')
4  disp(diag(D))
5  disp('autovectores')
6  disp(C)
```

La matriz no es diagonalizable y los valores propios se calculan correctamente (10 con multiplicidad dos). Se muestran dos vectores propios proporcionales y esto suena satisfactorio porque la matriz no es diagonalizable y, consecuentemente, no hay una base de autovectores. Sin embargo, al cambiar a `format long` (esto puede variar con versiones de `matlab/octave`) veremos que en realidad las columnas de C no son proporcionales. Podemos exagerar el efecto cambiando la matriz a `[10,1e-13;0,10]`. Sigue sin ser diagonalizable, pero el código produce dos autovectores claramente distintos. Para el segundo de ellos $\|A\vec{v} - 10\vec{v}\|$ se acerca al épsilon máquina y es imposible detectar que es un falso autovector.

ACTIVIDAD 7.1.1. *Considera la matriz $N \times N$ que tiene $a_{ii} = 2$, $a_{ij} = 1$ si $|i - j| = 1$ y el resto de sus elementos nulos. Escribe un programa que dibuje sus autovalores en orden creciente.*

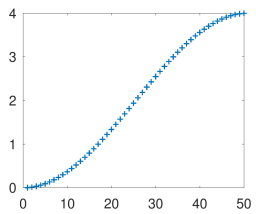
Si te sorprende la regularidad del resultado en la actividad anterior cuando N crece, la explicación más directa se basa en una fórmula exacta [11] para los valores propios que corresponde a una discretización de $4\,\mathrm{sen}^2\, x$ en cierto intervalo.

Mucho más sorprendente es que hay aparezca un patrón en la distribución de los autovalores de matrices aleatorias. Sin entrar en detalles sobre los comandos (consúltese la ayuda de `matlab/octave` si es necesario), el siguiente código comprueba que para las matrices aleatorias simétricas el

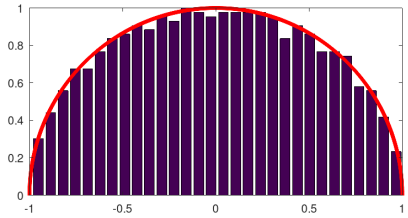
histograma de los autovalores convenientemente normalizado, se aproxima a un semicírculo.

```
% Matriz simétrica aleatoria
A = 2*rand(1000)-ones(1000);
A = A'+A;

% Normalización: máx |\lambda|=1
A = A/max(abs(eig(A)));

% Calcula el histograma de los autovalores
[nn,xx] = hist(eig(A),30);

figure(1)
% Normaliza en el dibujo la altura a 1
h = bar(xx,nn/max(nn));
hold on
% Semicircunferencia
t = linspace(0,pi,150);
plot( cos(t),sin(t),'r', 'LineWidth',6 )
axis('equal')
hold off
```

La *ley del semicírculo de Wigner* afirma que esto es un hecho general siempre que los elementos de la matriz simétrica respondan a variables aleatorias independientes, de media cero y varianza acotada.



Actividad para $N = 60$



Ley del semicírculo de Wigner

Sustituyendo la segunda línea del último programa por `A = randn(1000)` obtenemos resultados similares, ya que `randn` genera matrices cuyos elementos siguen una distribución normal estándar, la cual tiene media cero y varianza 1.

7.2. El método de la potencia

Aunque disponemos del comando `eig`, este es un curso para matemáticos y nuestro objetivo será entender algunos algoritmos para el cálculo, aunque sea parcial, de autovectores y autovalores de una matriz cuadrada.

Uno de los métodos más sencillos para calcular un autovalor y un autovector de una matriz cuadrada A es la iteración $\vec{x}_{n+1} = A\vec{x}_n$, lo que se llama *método de la potencia* porque $\vec{x}_n = A^n\vec{x}_0$. Habitualmente se normalizan los $\vec{x}_n$ para evitar divergencias. Si A tiene un solo *autovalor dominante* λ_1, es decir, es simple y $|\lambda_1| > |\lambda_j|$ para $j \neq 1$, para casi cualquier elección de $\vec{x}_0$, $\vec{x}_n$ se acerca paulatinamente a un autovector correspondiente al autovalor λ_1 y, consecuentemente, λ_1 se aproxima por el llamado *cociente de*

Rayleigh $\langle A\vec{x}_n, \vec{x}_n\rangle / \|\vec{x}_n\|^2$, donde el denominador es 1 si hemos normalizado. Por ejemplo, $-A_1$ tiene autovalor dominante -4 que podemos aproximar, junto con su autovector con el código:

```
A1 = [-7,6,-8,5; -16,10,-13,10; -5,2,-2,3; -8,6,-8,6];
A = -A1;
N = 8;
v = rand(size(A,1),1);
v = v/norm(v);

for k = 1:N
        vn = A*v;
        v = vn/norm(vn);
end

disp('Aproximación del autovector:')
disp(v)

disp('Aproximación del autovalor:')
disp(v'*A*v)
```

Normalmente queremos establecer un criterio de parada para saber si se ha conseguido cierta tolerancia en el error relativo estimando autovalores o autovectores, antes de haber gastado todas las iteraciones máximas disponibles. Para los primeros, una posibilidad es cambiar el bucle por:

```
tol = 0.01
fl = false;
for k = 1:N
        vn = A*v;
        vn = vn/norm(vn);
        la = v'*A*v;
        lan = vn'*A*vn;
        v = vn;
        if abs(la/lan-1) < tol
                fl = true;
                break
        end
end


if fl
        disp(['Se ha alcanzado la tolerancia exigida con N
                ↪ = ' num2str(k)])
else
        disp('No se ha alcanzado la tolerancia exigida')
end
```

Si queremos usar un criterio similar con los vectores propios hay que tener cuidado porque aunque informalmente decimos que $\vec{x}_n$ "converge" al autovector, los autovectores solo están definidos salvo multiplicar por constantes. Si por ejemplo λ_1 es negativo, $\vec{x}_n$ cambiará alternativamente de sentido y algo más complicado ocurre para autovalores complejos. Una solución es normalizar poniendo una de las coordenadas iguales a 1, típicamente la mayor en valor absoluto.

ACTIVIDAD 7.2.1. *Siguiendo la idea anterior, trata de hacer un criterio de parada que tome en cuenta el error relativo de los autovectores. Seguramente necesites usar el comando* `max` *en la forma* `[m,j] = max(abs(...))`.

La rapidez de convergencia depende de lo grande que sea el cociente $|\lambda_1|/|\lambda_2|$ donde λ_2 es el *autovalor subdominante*, el siguiente en tamaño del

módulo, que podría ser múltiple. Cuanto más cercano sea a uno este cociente, peor para la convergencia. Esto lleva a una situación curiosa. Matemáticamente tiene la misma complicación hallar los autovalores y autovectores de A que los de $A - \mu I$, pero numéricamente los cocientes $|\lambda_1|/|\lambda_2|$ pueden ser bien distintos. Por ejemplo, para A_2 es $4/3$ y para $A_2 - 2I$ es 2. Por tanto, es más conveniente aplicar el algoritmo a $A_2 - 2I$. Comprobémoslo:

```
A2 = [-1,3,-5,2; -7,7,-7,4; -1,1,1,0; -2,3,-5,3];
A = A2;
N = 10;
n = size(A,1)
v1 = rand(n,1);
v1 = v1/norm(v1);
v2 = v1;

for k = 1:N
        vn1 = A*v1;
        vn2 = (A-2*eye(n))*v2;
        v1 = vn1/norm(vn1);
        v2 = vn2/norm(vn2);
        disp([num2str(k) ': aprox 1 -> '
                ↪ num2str(v1'*A*v1,'%.6f') ' aprox 2 -> '
                ↪ num2str(v2'*A*v2,'%.6f')])
end
```

7.3. Localización de autovalores

Según lo que acabamos de ver, una idea previa de la localización de los autovalores ayuda a decidir acerca de la velocidad de convergencia. Los *círculos de Gershgorin* dan una información burda que se vuelve muy precisa para pequeñas perturbaciones de matrices diagonales.

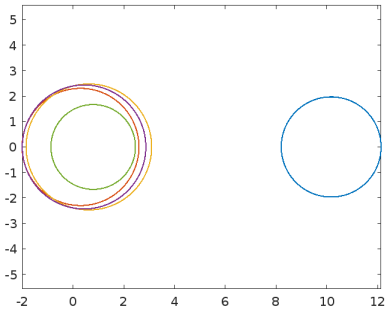
El resultado básico es que los autovalores de una matriz A están, en el plano complejo, dentro de la unión de los círculos (de Gershgorin) definidos por $|z - a_{ii}| \leq \sum_{j \neq i} |a_{ij}|$ y que si la unión de m de estos círculos no se solapa con los otros, entonces dicha unión contiene exactamente m autovalores [17, §6.3].

El siguiente código dibuja los círculos de Gershgorin para una matriz aleatoria con a_{11} mucho mayor que el resto de los elementos, con ello λ_1 estará en el círculo aislado a la derecha y λ_2 en la unión de los otros.

```
N = 5;
A = rand(N,N);
A(1,1) = A(1,1) + 2*N;

n = size(A,1);

t = linspace(0,2*pi,100);
figure(1)
for k = 1:n
        r = sum( abs(A(k,:)) ) - abs(A(k,k));
        plot( real(A(k,k))+ r*cos(t), imag(A(k,k))+
                ↪ r*sin(t) )
        hold on
end
axis('equal')
hold off
```

Usar `real`, `imag` en la línea 11 permite que el programa funcione para matrices complejas.

Una posible salida del código es la figura



ACTIVIDAD 7.3.1. *Añade alguna línea la código para que muestre también las posiciones reales de los autovalores. Intenta extremar la brevedad. Se puede hacer con una sola línea.*

En física se presenta la situación en la que uno conoce un autovalor simple λ de una matriz hermítica A y su autovector normalizado $\vec{v}$ y quiere estimar sus análogos λ' y $\vec{v}'$ para $A' = A+P$ donde P es una matriz hermítica pequeña. La teoría de perturbación dice que las fórmulas relevantes son

$$\lambda' \approx \lambda + \vec{v}^\dagger P \vec{v} \qquad \vec{v}' \approx \vec{v} + \sum_{\mu \neq \lambda} \frac{\vec{v}_\mu^\dagger P \vec{v}}{\lambda - \mu} \vec{v}_\mu$$

donde μ recorre los autovalores de A distintos de λ y $\vec{v}_\mu$ los correspondientes autovectores normalizados. Al igual que en otras ocasiones, la notación $\vec{v}^\dagger$ significa la matriz traspuesta conjugada de la que corresponde al vector columna $\vec{v}_\mu$. Estas fórmulas permiten encontrar un buen punto de partida para el método de la potencia o la extensión que veremos a continuación.

ACTIVIDAD 7.3.2. *Comprueba que la primera de las fórmulas anteriores da buenas aproximaciones para los autovalores de* `A3+ep*(R+R')` *con* `R = rand(4)` *y* `ep` *pequeño, considerada como una perturbación de* `A3`.

7.4. El método de la potencia inversa

Si aplicamos el método de la potencia a A^{-1}, con A no singular, el autovalor de menor valor absoluto pasa a ser el dominante. En general, aplicando el método a $(A - \mu I)^{-1}$ con μ cercano a λ_j (supuesto autovalor simple), conseguiremos que λ_j sea dominante y podremos aplicar el método de la potencia. Esta es la forma más básica del *método de la potencia inversa* y funciona tanto para autovalores reales como complejos, como se puede apreciar en el siguiente ejemplo:

```
A1 = [-7,6,-8,5; -16,10,-13,10; -5,2,-2,3; -8,6,-8,6];
A = A1;
N = 8;
n = size(A,1);
v = rand(n,1);
v = v/norm(v);

la0 = 0.8+0.8*i;
Ai = inv(A-la0*eye(size(A,1)));

for k = 1:N
        vn = Ai*v;
        v = vn/norm(vn);
end

disp('Aproximación del autovector:')
disp(v)

disp('Aproximación del autovalor:')
disp(v'*A*v)
```

Como `Ai` no varía, para un número grande de iteraciones, nos conviene calcular la inversa una sola vez, aunque esto sea costoso. También podríamos ir actualizando el valor de μ con las aproximaciones del autovalor para conseguir una convergencia más rápida. En ese caso, el cálculo de la inversa es menos ventajoso que la resolución del sistema en cada iteración. El siguiente código, que alguno diría que corresponde al algoritmo llamado *iteración del cociente de Rayleigh* (véase también la variante [15, (11.7.10)]), ilustra la situación:

```
A1 = [-7,6,-8,5; -16,10,-13,10; -5,2,-2,3; -8,6,-8,6];
A = A1;
N = 5;
n = size(A,1);
v = rand(n,1);
v = v/norm(v);

la = 0.8+0.8*i;

for k = 1:N
        vn = (A-la*eye(n))\v;
        v = vn/norm(vn);
        la = v'*A*v;
end

disp('Aproximación del autovector:')
disp(v)

disp('Aproximación del autovalor:')
disp(v'*A*v)
```

La convergencia es mucho más rápida que en el caso anterior, pero a cambio hay que pagar con resolver un sistema en cada iteración.

ACTIVIDAD 7.4.1. *Escribe un programa que compare el error en la aproximación del autovalor en las dos formas del algoritmo para este ejemplo.*

El método de la potencia inversa pone de relieve el interés tener estimaciones previas de la localización de los autovalores. Si λ es un autovalor simple de A separado del resto y μ es una buena aproximación suya, entonces $(A-\mu I)^{-1}$ tendrá a $(\lambda-\mu)^{-1}$ como valor propio y su valor absoluto será muy grande, lo que permite que la convergencia sea rápida.

7.5. El algoritmo QR (teórico)

Según la teoría, partiendo de $A_0 = A$ real con autovalores reales simples, la iteración $A_{n+1} = R_n Q_n$ donde Q_n y R_n son los factores en la descomposición QR de A_n, converge a una matriz triangular superior T igual a cambiar A a cierta base ortonormal, cuya diagonal son los autovalores de A. Si además A es simétrica, T será diagonal. Esta es la teoría bajo el algoritmo QR, pero como cada paso requiere del orden de n^3 operaciones, es bastante poco eficiente para matrices medianamente grandes. Por otro lado, resulta ridículamente simple de programar en `matlab/octave` utilizando la función nativa `qr`:

```
1  A2 = [-1,3,-5,2; -7,7,-7,4; -1,1,1,0; -2,3,-5,3];
2  A = A2;
3  N = 15;
4  n = size(A,1);
5  %Q = eye(n);
6
7  for k = 1:N
8          [Qp,R] = qr(A);
9          A =  R*Qp;
10 %        Q = Q*Qp;
11 end
12
13 disp('Aproximación de los autovalores')
14 disp(diag(A))
```

El método se puede adaptar con pocos cambios al caso de autovalores complejos [19, S6.6.4.15], pero no lo veremos aquí. Esencialmente lo que ocurre es que aparece un valor $T_{i\,i+1} \neq 0$ por cada pareja de autovalores conjugados.

Si quitamos el comentario en las las líneas 5 y 10, en Q tendremos una matriz unitaria tal que $Q^\dagger A_2 Q$ es aproximadamente triangular superior T, cuyos autovectores son fáciles de hallar y hallaríamos una aproximación de los autovectores de A_2 premultiplicando los de T por Q.

ACTIVIDAD 7.5.1. *Escribe una función* `eigt` *tal que* `[C,D] = eigt(R)` *funcione como* `eig` *para una matriz triangular superior* `R` *con todos los elementos de su diagonal distintos.*

Sin entrar en detalles, la forma de volver eficiente el algoritmo es reducir primero A a una matriz de Hessenberg superior con un cambio de base unitario. Estas matrices son las que tienen $a_{ij} = 0$ para $i > j + 1$. La particularidad de ellas es que la descomposición QR se puede hacer de forma mucho más económica. Así que el esquema sería:

```
1  A2 = [-1,3,-5,2; -7,7,-7,4; -1,1,1,0; -2,3,-5,3];
2  A = A2;
3  N = 10;
4  n = size(A,1);
5
6  [P,A] = hess(A);
7  %Q = P;
8
9  for k = 1:N
10         [Qp,R] = qr(A);
```

```
11          A =  R*Qp;
12  %       Q = Q*Qp;
13  end
14
15  disp('Aproximación de los autovalores')
16  disp(diag(A))
```

El comando `hess` de la línea 6 halla P unitaria con $P^\dagger AP$ de Hessenberg que se vuelve a almacenar en A.

ACTIVIDAD 7.5.2. *Compara el tiempo de ejecución de los programas anteriores con muchas iteraciones sobre matrices aleatorias simétricas* 100×100 *para estudiar si* `matlab/octave` *está realmente aprovechando la mayor eficiencia de la descomposición* QR *con las matrices de Hessenberg.*

Por supuesto, el procedimiento descrito solo tiene sentido si hay una manera eficiente de hallar una matriz de Hessenberg que sea igual a la de una matriz dada tras un cambio de base, preservando de esta forma los valores propios. Esto se reduce a una variante de la reducción de Gauss en la que se actúa simultáneamente en filas y columnas [15, §11.5].

Capítulo 8

Ecuaciones no lineales

8.1. Iteraciones de punto fijo

Antes de introducir de los métodos de Jacobi y Gauss-Seidel, ya habíamos visto un ejemplo de iteraciones de punto fijo para resolver ecuaciones no lineales. Concretamente, la recurrencia $x_{n+1} = \frac{1}{2}(x_n + 2/x_n)$, uno de los algoritmos más antiguos de la historia, que, como muestra el siguiente programa, da una aproximación muy rápida a $\sqrt{2}$.

```
format long
N = 4;
x = 1;
for k = 1:N
        x = (x+2/x)/2;
end
disp(x)
err = abs(x-sqrt(2));
disp(['Error = ' num2str(err)])
```

Con `N = 5` ya tenemos un error comparable al épsilon máquina.

Según la teoría, la idea es que $x_{n+1} = g(x_n)$ genera una sucesión $\{x_n\}_{n=0}^{\infty}$ que tiende a un cero α de $f(x) = g(x) - x$ siempre que x_0 esté suficientemente cerca de α y g, supuesta regular, cumpla $\left|g'(\alpha)\right| < 1$.

Por ejemplo, la elección más tonta $f(x) = x^2 - 2$ no es adecuada para aproximar $\sqrt{2}$ porque $g(x) = x^2 - 2 + x$ cumple $g(\sqrt{2}) > 1$. El siguiente código ilustra esta situación. A pesar de que el punto de partida está cerca de $\sqrt{2}$, no se observa ninguna convergencia, ni incrementando el valor de `N`.

```
N = 10;
x = 1.4;
for k = 1:N
        x = x^2 - 2 + x
end
```

Imitando, lo hecho con los sistemas lineales, introducimos una constante $c \neq 0$ y consideramos $f(x) = c(x^2 - 2)$. De esta forma, no se alteran los ceros y $g(x)$ pasa a ser $x + c(x^2 - 2)$. Por ejemplo con $c = -1/2$ se tiene $\left|g'(\sqrt{2})\right| = \left|1 - \sqrt{2}\right| < 1$ y la convergencia está asegurada suficientemente

cerca de $\sqrt{2}$.

```
1 N = 10;
2 c = -1/2
3 x = 1;
4 for k = 1:N
5         x = c*(x^2 - 2) + x
6 end
7 disp(['Error absoluto (con signo) = ' num2str(x-sqrt(2))])
```

La elección de la constante es importante. Si cambiamos la línea 2 por `c = -5/14` obtendremos un error más pequeño con solo tres iteraciones.

ACTIVIDAD 8.1.1. *Busca una fracción de denominador menor que 20 que reemplace a* `-5/14` *para obtener menor error todavía y compruébalo experimentalmente.*

Cuanto menor sea $|g'(\alpha)|$, más rápida será la convergencia. El éxito de `c = -5/14` se debe a que $|g'(\alpha)|$ es del orden de 10^{-2} y con `c = -0.3536` pasaría a ser 10^{-4}, resultando un método todavía más rápido. Esta idea nos explica de por qué el método inicial es tan bueno. Allí $g'(x) = \frac{1}{2}(1 - 2/x^2)$ y por tanto $g'(\sqrt{2}) = 0$. Dicho sea de paso, aparentemente es más simple ajustar c para que $g'(\sqrt{2}) = 0$, pero eso es un espejismo en general porque si no conocemos un cero α de $f(x) = g(x) - x$, es casi seguro que no conoceremos el valor de $g'(\alpha)$.

Aunque cero sea el menor valor absoluto posible, hay diferentes categorías de ceros porque los órdenes de anulación aceleran la convergencia. Por ejemplo, $g(x) = x + \operatorname{sen} x$ cumple no solo $g'(\pi) = 0$ sino también $g''(\pi) = 0$. Eso tiene que ver con la fantástica rapidez de convergencia a π del siguiente algoritmo que se puede replicar en una antigua calculadora de bolsillo sin más que partir de 3 y pulsar sucesivamente las teclas $\boxed{+}$, $\boxed{\text{sen}}$ y $\boxed{=}$.

```
1 N = 2;
2 x = 3;
3 for k = 1:N
4         x = x + sin(x);
5 end
6 disp(x)
7 err = abs(x-pi);
8 disp(['Error = ' num2str(err)])
```

Con dos iteraciones tenemos un error al aproximar π menor que `2e-11` y con tres iteraciones se llega a algo comparable al épsilon máquina.

Una situación habitual, que ya habíamos implementado en entregas anteriores, es que nuestro objetivo no es tanto llevar a cabo un número definido de iteraciones como alcanzar un error, o bien absoluto o bien relativo, por debajo de cierta tolerancia. En las iteraciones de punto fijo, el error está aproximado por $x_n - x_{n+1}$, ya que pensamos que x_{n+1} sirve como sustituto del valor buscado porque aproxima mejor que x_n. Con esta idea en mente, el siguiente código establece como criterio de parada que la estimación del error relativo sea menor que `tol`. Si quisiéramos usar el error absoluto como criterio de parada, bastaría eliminar `*abs(xn)` de la línea 6.

```
Nmax = 100;
tol = 1e-4;
x = 1;
for k = 1:Nmax
        xn = x+1/2*(2-x^2);
        if abs(xn-x)<tol*abs(xn)
                x = xn;
                break;
        end
        x = xn;
end
fprintf('%d iteraciones. Error = %.6e.\n',k,x-sqrt(2));
```

El uso de `fprintf` en vez de `disp`, que funciona como el `printf` de C, es simplemente para practicar con diferentes comandos.

8.2. Aceleración de la convergencia

Seguramente te sorprenda que ya L. Euler en el siglo XVIII inventó varios procedimientos para acelerar series de convergencia lenta. En principio eso suena imposible porque una serie es la que es y parece que solo hay un método para hacer una suma.

A modo de introducción, a menudo en cálculo numérico un algoritmo que requiere un esfuerzo computacional de N pasos, consigue una aproximación $A(N)$ de una cantidad objetivo α de la forma $A(N) = \alpha + CN^{-p} + \dots$ con C una constante, p típicamente un entero positivo y los puntos suspensivos representando términos de orden inferior, en el sentido de que al multiplicar por N^p tienden a cero. Cuanto mayor sea p, mejor aproxima a la larga el algoritmo. La *extrapolación de Richardson* [20], permite eliminar el término CN^{-p} por medio de

$$\frac{2^p A(2N) - A(N)}{2^p - 1} = \alpha + \dots$$

por tanto el primer miembro de esta expresión mejora a $A(N)$ a la larga, esto es, acelera el algoritmo.

El esquema anterior no se aplica tal cual a la iteración de punto fijo porque en ellas esperamos desarrollos exponenciales. En el caso de orden uno, sin entrar en la definición, la idea es que en cada iteración el error se reduce multiplicándose aproximadamente por una constante C de valor absoluto menor que 1 y entonces

$$x_n = \alpha + AC^n + \dots$$

donde A es constante y los puntos suspensivos son términos de orden inferior. Definamos

$$x_n^* = x_{n-2} - \frac{(x_{n-1} - x_{n-2})^2}{x_n - 2x_{n-1} + x_{n-2}}.$$

Esto es lo que se llama *fórmula de extrapolación de Aitken* [1, §2.6]. Operando un poco vemos que en $x_n^* - \alpha$ solo quedan términos de orden inferior. Además, el cálculo de x_n^* sigue requiriendo las mismas n iteraciones.

ACTIVIDAD 8.2.1. *Crea una función* `acel` *que, utilizando el comando* `circshift`*, transforme una sucesión* $\{x_n\}_{n=1}^N$ *en una sucesión acelerada* $\{x_n^*\}_{n=1}^N$ *por medio de la fórmula anterior. Define* $x_n^* = x_n$ *para* $n = 1, 2$.

Por ejemplo, la iteración de punto fijo con $g(x) = x - (x^2 - 2)/2$

```
N = 10;
x = 1;
for k = 1:N-1
        x = x-(x^2 - 2)/2;
end
disp(['Error absoluto (con signo) = ' num2str(x-sqrt(2))])
```

muestra un error `7.916e-05`. Sabiendo que el método es de orden 1, en el sentido anterior, vamos a acelerarlo con las mismas 10 iteraciones:

```
N = 10;
x = ones(N,1);
x(1) = 1;
for k = 1:N-1
        x(k+1) = x(k)-(x(k)^2 - 2)/2;
end

for k = 3:N
        disp('----------')
        disp(['Iteración ' num2str(k)])
        xa = x(k-2)-
            ↪ (x(k-1)-x(k-2))^2/(x(k)-2*x(k-1)+x(k-2));
        disp(['Error sin acelerar ' num2str(x(k)-sqrt(2))])
        disp(['Error acelerado ' num2str(xa-sqrt(2))])
end
```

El error pasa a ser ahora `-3.1169e-08`, lo cual es una mejora muy considerable, más de mil veces menor. El programa ofrece la comparativa entre los valores anteriores sin acelerar y los acelerados. La diferencia es bastante espectacular.

La fórmula anterior que transforma una sucesión en otra acelerada tiene la desventaja de que solo sirve para orden uno. Además es un poco absurdo utilizar en cada paso los valores no acelerados x_n en lugar de las mejoras x_n^* que vamos hallando. Por ello, una forma más conveniente de la *extrapolación de Aitken*, y además válida para todos los órdenes, es [17, §2.4]:

$$x_{n+1} = x_n - \frac{\big(g(x_n) - x_n\big)^2}{g(g(x_n)) - 2g(x_n) + x_n}.$$

ACTIVIDAD 8.2.2. *Incorpora esta forma de la extrapolación de Aitken en la comparativa anterior. Seguramente te sea conveniente definir una* función anónima. *Si las desconoces, mira la siguiente sección.*

ACTIVIDAD 8.2.3. *Comprueba que este método también acelera el algoritmo inicial* $x_{n+1} = (x_n + 2/x_n)/2$ *para aproximar* $\sqrt{2}$*, que es de orden 2.*

Observarás que el error es tan pequeño que enseguida se producen problemas de división por cero.

8.3. El método de la bisección

Según el algoritmo visto en la teoría, ligeramente reordenado, el método de la bisección para $f(x) = x^2 - 2$ partiendo del intervalo $[1, 2]$ se implementa como:

```
f = @(x) x^2 -2; % función anónima

Nmax = 10;
a = 1;
b = 2;

for k = 1:Nmax
        c = (a+b)/2;
        if f(a)*f(c)<0
                b = c;
        else
                a = c;
        end
end
disp(['Error: ' num2str((a+b)/2-sqrt(2))])
```

Aquí se ha usado una *función anónima* `f`. En vez de definir `f` en un fichero externo, escribimos `@(x)` seguido de la fórmula en términos de `x`. Como siempre en `matlab/octave`, la variable `x` puede ser matricial, aunque no sea el caso aquí.

La siguiente modificación del código anterior, muestra la elección de los intervalos sucesivos en el método de la bisección. El error en el paso n está acotado por $(b-a)2^{-n}$.

```
f = @(x) x^2 -2; % función anónima

Nmax = 10;
a = 1;
b = 2;

for k = 1:Nmax
        c = (a+b)/2;
        if f(a)*f(c)<0
                b = c;
                disp(['Izquierdo ' num2str(a) ' , '
                        ↪ num2str(b)])
        else
                a = c;
                disp(['Derecho ' num2str(a) ' , '
                        ↪ num2str(b)])
        end
end
```

En `matlab/octave`, con una sintaxis que ya aparece en las funciones anónimas, es posible pasar funciones definidas en ficheros como argumentos de otras funciones. Simplemente se precede el nombre de la función a la que se apela con `@`.

Por ejemplo, supongamos que hemos definido en `pf4.m` una función que aplica cuatro veces la iteración de punto fijo partiendo de $x_0 = 1$:

```
function x = pf4( f )
        x = 1;
        for k =1:4
                x = f(x);
        end
end
```

y, para replicar el ejemplo inicial, queremos que la aplique a la función `f1` definida en un fichero `f1.m`

```
function res = f1( x )
        res = (x+2/x)/2;
end
```

Si intentamos `pf4( f1 )` obtendremos un error. Esto es lógico porque, cuando llame a `f1`, `matlab/octave` no sabrá qué es `x`. Lo correcto es escribir `pf4( @f1 )`. Las funciones anónimas se comportan como si ya incorporasen `@`. De este modo, `pf4( @f1 )` es equivalente a

```
g = @(x) (x+2/x)/2;
pf4( g )
```

Si tienes una versión (¿muy?) antigua de `matlab/octave`, pasar funciones anónimas como argumento hará saltar errores raros y las dos líneas anteriores no funcionarán. Para resolverlo tendrás que sustituir `f(x)` por `feval(f,x)` dentro de la definición de `pf4`.

ACTIVIDAD 8.3.1. *Escribe una función* `bisec(f,a,b,err)` *que aplique el método de la bisección donde los argumentos son la función* `f`*, el intervalo inicial* `[a,b]` *y el* `err` *el tamaño del intervalo a partir del cual se detiene la iteración.*

8.4. El método de Newton

El esquema

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

es el conocido *método de Newton* o *método de Newton-Raphson* para hallar soluciones de $f(x) = 0$. Geométricamente corresponde a aproximar cerca de x_n la función por su tangente y calcular su corte con el eje X para obtener x_{n+1}.

Una implementación `mnewton( f, df, x0, N )` con `f` la función, `df` su derivada, `x0` el valor inicial y `N` el número de iteraciones es:

```
function x = mnewton( f, df, x0, N )
        x = x0;
        for k = 1:N
                x = x-f(x)/df(x);
        end
end
```

En el caso $f(x) = x^2 - 2$, para replicar el ejemplo inicial, lo llamaríamos con `mnewton(@newf, @newdf, 1, 4)` definiendo

```
function res = newf( x )
        res = x^2-2;
end
```

y

```
function res = newdf( x )
        res = 2*x;
end
```

Por la teoría, sabemos que el método de Newton converge para hallar α con $f(\alpha) = 0$ siempre que $f''(\alpha) \neq 0$ y x_0 esté suficientemente cerca de α. Fuera de estas condiciones, podemos entrar en problemas. En [19, §5.4] se da como ejemplo $f(x) = \arctan x$ que da una sucesión divergente para $\arctan|x_0| > 2|x_0|/(1 + x_0^2)$, lo cual se cumple cuando x_0 es grande.

Por otro lado, no hay una respuesta fácil a la pregunta de para qué valores converge el método de Newton ni a qué converge. La complicación de la situación se manifiesta si lo aplicamos en el plano complejo $\mathbb{C}$. Típicamente la frontera de los puntos en los que converge a cierto cero de la función es un fractal (un *conjunto de Julia*). Para explorar esta situaciones, introduzcamos primero el comando `imshow` que, en nuestro uso, transforma los valores de una matriz en tonos de gris. El segundo argumento es un intervalo que indica el valor mínimo que se asocia al negro y el valor máximo que se asocia al blanco. Por ejemplo:

```
un = ones(50);
A = [0*un,1*un;2*un,3*un];
imshow(A, [0,3])
```

muestra un cuadrado dividido en otros cuatro de 50×50 píxeles con tonos de gris que van del negro al blanco.

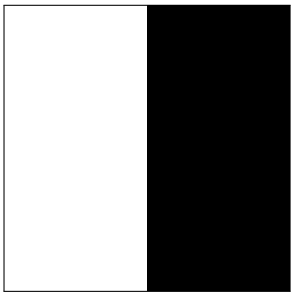
Coloreemos con tres tonos de gris diferentes cada x_0 en el cuadrado $\max\left(|\Re x_0|, |\Im x_0|\right) < 3/2$, dependiendo de si el método de Newton aplicado a $f(x) = x^3 - 2$ converge a cada una de sus tres ceros complejos.

```
N = 400;
Nmax = 30;
z0 = 2^(1/3);
z1 = z0*exp(2*pi*i/3);
z2 = z0*exp(-2*pi*i/3);
tol = 0.01;

A = meshgrid(linspace(-3/2,3/2,N),1:N);
A = A + i*A' + eps;
for l = 1:Nmax
        A = 2*(A.^3+1)./A.^2/3;
end

A = (abs(A-z0)<tol) + 2*(abs(A-z1)<tol) + 3*(abs(A-z2)<tol);

imshow(A, [0,3])
```

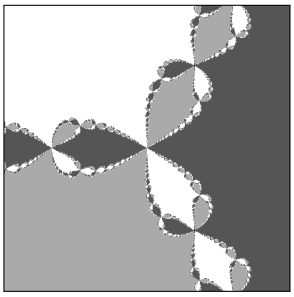
El dibujo evidencia la complicación de la convergencia. Hay ternas de valores

iniciales arbitrariamente próximos que dan sucesiones que convergen a ceros distintos.

En el caso cuadrático x^2-2 sí hay un criterio fácil para la convergencia en el plano complejo: si $\Re x_0 > 0$ el algoritmo converge a $\sqrt{2}$ y si $\Re x_0 < 0$ lo hace a $-\sqrt{2}$ (además no converge para $\Re x_0 = 0$). Es decir, no se manifiesta ningún comportamiento fractal. Las siguientes figuras ilustran la situación:



Newton x^2-2



Newton x^3-2

ACTIVIDAD 8.4.1. *Busca información sobre el comando* `subplot` *y completa el programa anterior para que muestre adosadas las imágenes correspondientes a* $f(x) = x^3-2$, $f(x) = x^4-2$ *y* $f(x) = x^5-2$. *Tendrás que incrementar* `Nmax` *en las dos últimas para no ver puntos negros asociados a falta de convergencia en la tolerancia permitida.*

Hay también un método de Newton válido para el caso de varias variables reemplazando la derivada por la matriz jacobiana [17] [19].

8.5. El método de la secante

No siempre tenemos la suerte de contar con una expresión analítica para la función de la que queremos aproximar las raíces ya que no es inusual que f en realidad represente la salida de un algoritmo más o menos complicado. Incluso si la tenemos, la expresión de la derivada puede ser demasiado compleja, afectando al rendimiento. La solución es aproximar $f'(x_n)$ en el método de Newton por la derivada numérica $\big(f(x_n)-f(x_{n-1})\big)/(x_n-x_{n-1})$, lo que da lugar al método de la secante:

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

Hay también una justificación geométrica, que da nombre al método y consiste reemplazar en el método de Newton tangentes por secantes que unen los puntos $\big(x_n, f(x_n)\big)$ y $\big(x_{n+1}, f(x_{n+1})\big)$.

Una posible implementación para $f(x) = x^2-2$ es:

```
1  Nmax = 4;
2  x1 = 1;
```

```
x2 = 2;
for k = 1:Nmax
        x3 = x2-(x2^2-2)*(x2-x1)/(x2^2-x1^2);
        x1 = x2;
        x2 = x3;
end
disp(['Error secante = ',num2str(x3-sqrt(2))]);
```

El rendimiento es algo peor que el del método de Newton. Este código aplica ambos algoritmos

```
Nmax = 4;
x1 = 1;
x2 = 2;
x = x1;
for k = 1:Nmax
        x3 = x2-(x2^2-2)*(x2-x1)/(x2^2-x1^2);
        x1 = x2;
        x2 = x3;
        x = x/2+1/x;
end
fprintf('Error secante = %.6e.\n',x3-sqrt(2));
fprintf('Error Newton = %.6e.\n',x-sqrt(2));
```

y arroja como salida

```
Error secante = -2.123898e-06.
Error Newton = 1.594724e-12.
```

ACTIVIDAD 8.5.1. *Realiza un programa que aplique el método de la secante para* $f(x) = x^2 - 2$ *con* $x_0 = 1$, $x_1 = 2$ *y dibuje gráfica de* f *en* $[1, 2]$ *y, en diferente color, las secantes que unen* $\big(x_n, f(x_n)\big)$ *y* $\big(x_{n+1}, f(x_{n+1})\big)$ *con* $n = 0, 1, 2$.

Capítulo 9

Interpolación

9.1. Interpolación de Lagrange y baricéntrica

La situación más básica en la teoría de interpolación consiste en que tenemos ciertos *nodos* $x_0 < x_1 < \ldots < x_n$ en el eje X, ciertas imágenes respectivas y_j y queremos conectar los puntos $\{(x_j, y_j)\}_{j=0}^{n}$ por medio de una función. En el caso de la interpolación polinómica, buscamos un *polinomio interpolador* de grado mínimo. En la forma de Lagrange, viene dado por:

$$P(t) = \sum_{j=0}^{n} y_j L_j(t) \qquad \text{donde} \qquad L_j(t) = \prod_{k \neq j} \frac{t - x_k}{x_j - x_k}.$$

Genéricamente el grado de P es n, pero para datos especiales puede tener grado menor. La explicación teórica de que la fórmula para P se ajusta a nuestras necesidades es sencilla. Se basa simplemente en notar que $L_j(t)$ es 1 si $t = x_j$ y se anula en el resto de los nodos.

Dados vectores de datos `x` e `y`, la siguiente función `matlab/octave` evalúa el polinomio de interpolación en un valor `t` aplicando la fórmula anterior, tal cual está.

```
function res = lagr_inef(x,y,t)
        n = length(x)-1;
        res = 0;
        for j =0:n
                % Calcula Lj(t)
                Lj = 1;
                for k=0:j-1
                        % El punto antes de * permite que t sea
                            ↪ vectorial
                        Lj = Lj.*(t-x(k+1))/(x(j+1)-x(k+1));
                end
                for k=j+1:n
                        Lj = Lj.*(t-x(k+1))/(x(j+1)-x(k+1));
                end
                % Suma la contribución al resultado
                res = res + y(j+1)*Lj;
        end
end
```

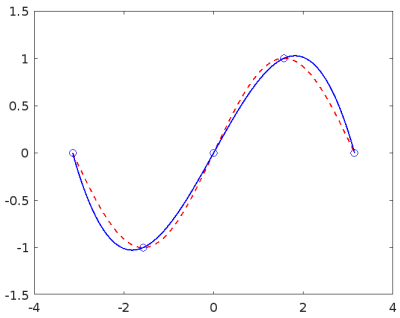
Descomponer el producto que define L_j en dos, es algo más eficiente que comprobar $n+1$ veces la condición $k \neq j$, por eso se ha escrito así. Por otro

lado, conociendo el comando `prod`, cuyo uso se aclarará en una actividad posterior, es posible ganar en rapidez de ejecución. Como sugiere uno de los comentarios, los puntos antes de `*` en las líneas 9 y 12 son superfluos para calcular valores individuales, pero si los dejamos, servirá también para `t` vectorial, evitando llamadas múltiples.

Por ejemplo, el siguiente código dibuja cinco puntos de la gráfica de $y = \operatorname{sen} x$ en $[-\pi, \pi]$, la gráfica en sí (en línea de puntos roja) y el polinomio de interpolación:

```
1  n = 4;
2  x = linspace(-pi,pi, n+1);
3  y = sin(x);
4
5  figure(1)
6  plot(x,y,'bo')
7  hold on
8  t = linspace(-pi,pi,100);
9  plot(t,sin(t), 'r--')
10 hold on
11 plot( t,  lagr_inef(x, y, t) )
12 hold off
```

En la línea 11 se usa de manera fundamental que `lagr_inef` admite valores de `t` vectorial, en otro caso habría que complicar el código con un bucle que llame repetidamente a la función. La salida es la figura



El sufijo `inef` del nombre de la función viene porque esta es una manera ineficiente de proceder si el número de nodos $n + 1$ es grande, ya que la evaluación para cada una de las coordenadas de `t` requiere una cantidad de operaciones comparable a n^2 que no es absolutamente necesario.

Una forma mejor de proceder numéricamente es utilizar la *forma baricéntrica* del polinomio de interpolación. Esta se resume en que, con unas manipulaciones ingeniosas elementales en $\sum y_j L_j(x)$, se obtiene

$$P(t) = \frac{N(t)}{D(t)} \quad \text{con} \quad N(t) = \sum_{j=0}^{n} \frac{w_j y_j}{t - x_j}, \quad D(t) = \sum_{j=0}^{n} \frac{w_j}{t - x_j}$$

y w_j el denominador de L_j, es decir, $w_j = \prod_{k \neq j} (x_j - x_k)^{-1}$. Hay dos ventajas principales en esta formulación. La primera es que al ser los w_j independientes de t, cada nueva evaluación solo requiere una cantidad de operaciones

comparable a n. La segunda ventaja es que añadir nuevos nodos, es decir, variar n, no implica rehacer todos los cálculos desde cero, basta añadir un sumando a N y a D por cada nuevo nodo. Esta propiedad la comparte con el *método de diferencias divididas de Newton*, que también se verá en la teoría. Una pequeña desventaja, meramente técnica, es que hay que tener cuidado con no aplicar la fórmula para t muy cercano o igual a x_j. Además de las ventajas anteriores, se tiene que para ciertas elecciones naturales de los nodos los w_j admiten fórmulas explícitas [2] que permiten una precomputación de sus valores.

Teniendo en mente la primera ventaja, construimos una función que tiene los w_j como uno de sus argumentos:

```
function res = lagr_bar(x,y,w,t)
        n = length(x)-1;
        N = 0;
        D = 0;
        for j=1:n+1
                v = w(j)./(t-x(j));
                N = N + y(j)*v;
                D = D + v;
        end
        res = N./D;
end
```

De nuevo `t` puede ser vectorial.

No es difícil probar que para nodos equiespaciados en el intervalo $[a, b]$ se puede tomar [2, (5.1)]:

$$w_j = (-1)^j \binom{n}{j}.$$

En realidad, usando la definición, w_j es un múltiplo de esta cantidad, pero esto introduce factores comunes en N y D que se cancelan.

Sabiendo que `nchoosek` es la función en `matlab/octave` que da los números combinatorios, para conseguir con `lagr_bar` el mismo dibujo obtenido con `lagr_inef` usamos:

```
a = -pi;
b = pi;
n = 4;
x = linspace(a,b, n+1);
y = sin(x);
w = zeros(n+1,1);
for j = 0:n
        w(j+1)= (-1)^j*nchoosek(n,j);
end

figure(1)
plot(x,y,'bo')
hold on
t = linspace(a,b,100);
plot(t,sin(t), 'r--')
hold on
plot( t,  lagr_bar(x, y, w, t) )
hold off
```

Si usamos el código tal como está, veremos que la línea continua se aleja un poco de los nodos inicial y final. Esto se debe a las singularidades en

x_0 y x_n. Una manera, un poco chapucera, pero efectiva, de evitarlo, es desplazar `t` añadiendo una cantidad algo mayor que el épsilon máquina. Por ejemplo, completando la línea 14 con `+10*eps`. El bucle de las líneas 7–9 se ha introducido porque en `matlab/octave` el comando `nchoosek` no tiene versión vectorial. Por otro lado, el desplazamiento en el índice de w_j introducido con `w(j+1)` en la línea 8, se debe a que `matlab/octave` no admite índices nulos en vectores y matrices.

La actividad siguiente muestra las deficiencias del polinomio interpolador y que palían otros métodos de interpolación como los *splines*.

ACTIVIDAD 9.1.1. *Utilizando la función que has creado, dibuja la gráfica obtenida al interpolar* $\{(x_j, y_j)\}_{j=0}^{2K}$ *donde* $x_j = j - K$, $y_j = x_j$ *excepto por* $y_K = 0{,}01$. *¿qué ocurre cuando* K *crece (por ejemplo para* $K = 10$*)?*

ACTIVIDAD 9.1.2. *La función* `prod` *en* `matlab/octave` *calcula el producto de las coordenadas de un vector fila y* `v(j)=[]` *elimina la coordenada* `j` *de* `v`. *Sabiendo esto, escribe una función* `wj` *con un solo bucle tal que* `wj(x)` *genere el vector* $\{w_j\}_{j=0}^{n}$ *correspondiente a* $\{x_j\}_{j=0}^{n}$.

9.2. Polinomios e interpolación con `matlab/octave`

Si estamos usando el polinomio de interpolación para aproximar de una función fija que vamos a usar muchas veces, seguramente preferiremos el polinomio en sí en vez de un algoritmo para construirlo y evaluarlo cada vez. En `matlab/octave` hay algunos comandos que permiten tratar algunos aspectos muy básicos del álgebra de polinomios. Un polinomio se representa con un vector fila que indica sus coeficientes, siendo el último elemento el que corresponde al término independiente. Por ejemplo, `[1,1,1]` es $x^2 + x + 1$ y `[1,-1,0,1,-1]` es $x^4 - x^3 + x - 1$. La función `conv` multiplica dos polinomios[1]. Por otro lado `polyval(p,x)` evalúa un polinomio `p` en `x`, que puede ser un escalar o un vector. Tras ejecutar el siguiente código:

```
1  % Polinomio x^2+x+1
2  p1 = [1,1,1];
3  % Polinomio x^4-x^3+x-1
4  p2 = [1,-1,0,1,-1];
5  % Producto
6  p = conv(p1,p2)
7  % Evaluación en x =0,1,2
8  x = 0:2;
9  ev = polyval(p,x)
```

obtendremos como salida `p=[1,0,0,0,0,0,-1]`, porque $(x^2+x+1)(x^4-x^3+x-1) = x^6 - 1$ y `ev=[-1,0,63]` porque esos son los resultados al evaluar en $x = 0, 1, 2$.

[1] El nombre `conv` se debe a que `matlab/octave` está más dirigido a ingenieros que a matemáticos o informáticos y una operación importante entre señales, llamada *convolución*, es similar formalmente a la multiplicación de polinomios.

También se tiene el comando `poly(r)` que dado un vector fila `r` construye el polinomio mónico que lo tiene por raíces. Así `poly([2,3])` es `[1,-5,6]`. Como ejemplo el siguiente código calcula en `Lj`, el polinomio L_j de la forma de Lagrange de la interpolación. Tal como está, corresponde a L_1 con cinco nodos ($n = 4$) equiespaciados en $[-1, 1]$.

```
1  j = 1;
2  n = 4;
3  x = linspace(-1,1,n+1);
4  temp = x
5  temp(j+1) = []
6  Lj = poly(temp)/prod(x(j+1)-temp);
```

ACTIVIDAD 9.2.1. *Modifica el código anterior para que muestra las gráficas de todos los L_j para $n = 4$ y los nodos como antes.*

La interpolación es un tema muy común en cálculo numérico por eso no debe extrañar que `matlab/octave` tenga una función nativa para hallar los coeficientes del polinomio de interpolación su estructura más básica es `polyfit(x,y,n)` donde `x` e `y` son los vectores cuyas coordenadas dan los puntos de interpolación (x_j, y_j) y `n` es un parámetro redundante que, siguiendo la notación habitual, indica el número de nodos menos uno. Utilizando esta función, el ejemplo de la interpolación de $y = \operatorname{sen} x$ que venimos manejando, sería:

```
1  n = 4;
2  x = linspace(-pi,pi, n+1);
3  y = sin(x);
4  cpi = polyfit(x,y,n);
5
6  figure(1)
7  plot(x,y,'bo')
8  hold on
9  t = linspace(-pi,pi,100);
10 plot(t,sin(t), 'r--')
11 hold on
12 poin = polyval(cpi,t);
13 plot( t,  poin )
14 hold off
```

Las líneas a las que hay que prestar atención son la 4 que define `cpi` como la lista de los coeficientes del polinomio de interpolación y la 12, que evalúa tal polinomio en los valores dados por las coordenadas del vector `t`.

Seguro que te estás preguntando acerca de qué sentido tiene el argumento redundante `n` en `polyfit(x,y,n)`. La respuesta es que `polyfit` tiene una utilidad más general que la vista aquí en la que `n` indica el grado del polinomio que ajusta los datos `x` e `y`. Si `n` es al menos la longitud de estos vectores menos uno, existe un polinomio que ajusta perfectamente, el de interpolación. Si `n` fuera menor, entonces lo que devuelve `polyfit` son los coeficientes del ajuste por mínimos cuadrados.

ACTIVIDAD 9.2.2. *Considera los nodos $x_j = j - 2$ con $0 \leq j \leq n = 4$ e $y_j = \cos x_j + \operatorname{sen} x_j$ y dibuja las gráficas correspondientes a* `polyfit(x,y,n)` *con* `n` $= 1, 2, 3, 4$ *en una misma figura.*

9.3. Fenómeno de Runge

Tomando muchos nodos que cubran bien un intervalo, uno esperaría una buena aproximación de cualquier función regular por medio del polinomio interpolador. Sin embargo, no es así. El ejemplo clásico es la función $f(x) = (1+x^2)^{-1}$. A pesar de pertenecer a C^∞, su polinomio interpolador P_n correspondiente a $n+1$ nodos equiespaciados en $[-5,5]$ no cumple $\|f-P_n\|_\infty \to 0$, de hecho $\|f-P_n\|_\infty \to \infty$. Esto es lo que se llama *fenómeno de Runge*. Si tienes curiosidad en ver pruebas matemáticas del inesperado comportamiento del polinomio interpolador en este ejemplo, las puedes encontrar en [10, §3.4] y [6].

Con el siguiente código se ilustra la situación. Para simplificar, se ha usado la función nativa `polyfit`.

```
n = 10;
x = linspace(-5,5, n+1);
y = 1./(1+x.^2);
cpi = polyfit(x,y,n);

figure(1)
t = linspace(-5,5,200);
plot(t,1./(1+t.^2), 'r--')
hold on
plot( t,  polyval(cpi,t) )
hold off
```

En [6] se muestra que la convergencia se recuperaría con otra elección de los nodos.

Teniendo en mente la fórmula del error de interpolación, la siguiente actividad da indicios de por qué el fenómeno de Runge se produce en los extremos y sugiere que para evitarlo deberíamos juntar más los nodos cerca de ellos.

ACTIVIDAD 9.3.1. *Haz un programa que dibuje la gráfica de* $g(x) = \prod_{j=0}^{n} |x-x_j|$ *para* $x_0 = -5, x_1, \ldots, x_n = 5$ *equiespaciados.*

En esta línea, en la teoría verás que, en cierto sentido, la elección óptima consiste en tomar los *nodos de Chebyshev* adaptados al intervalo. En $[-1,1]$, estos nodos vienen dados por la fórmula

$$x_j = \cos\left(\frac{(2j+1)\pi}{2n+2}\right) \qquad \text{con} \quad j = 0,1,\ldots,n.$$

Nota que no están en el orden habitual. Para utilizarlos en $[-5,5]$, debemos escalarlos multiplicando por 5. El siguiente código muestra la extraordinaria diferencia entre utilizar en este intervalo los nodos equiespaciados y los de Chebyshev para $f(x) = (1+x^2)^{-1}$ con $n = 10$.

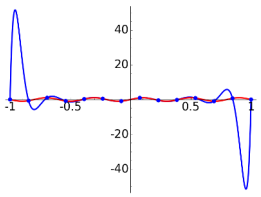
```
n = 10;
x = linspace(-5,5, n+1);
y = 1./(1+x.^2);
cpi = polyfit(x,y,n);

```

```
xc = 5*cos((2*[0:n]+1)*pi/(2*n+2));
yc = 1./(1+xc.^2);
cpic = polyfit(xc,yc,n);

figure(1)
t = linspace(-5,5,200);
plot(t,1./(1+t.^2), 'g--', t, polyval(cpi,t), 'b', t,
        ↪ polyval(cpic,t), 'r' )
```

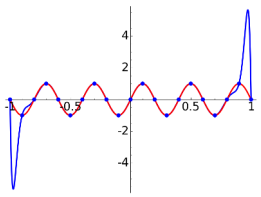
El fenómeno de Runge para nodos equiespaciados no es universal y tiene que ver con cuestiones de variable compleja [6].

ACTIVIDAD 9.3.2. *Comprueba que el fenómeno de Runge no se muestra para* $f(x) = \cos x$ *en* $[-5, 5]$ *cuando* $n \leq 20$.

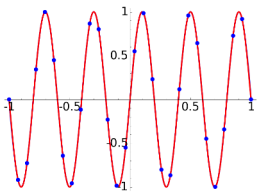
En la actividad anterior se limita n a un valor no muy elevado para no entrar en problemas con el épsilon máquina. Desde el punto de vista teórico, no hay ninguna limitación. La explicación tiene que ver con que las derivadas sucesivas de f están acotadas. Por otro lado, un crecimiento incluso exponencial de las derivadas no asegura que se produzca el fenómeno de Runge. Si experimentamos con $f(x) = \text{sen}(5\pi x)$ en $[-1, 1]$, según vamos incrementando el número de nodos inicialmente se manifiesta el fenómeno, pero a partir de cierto momento parece arreglarse.



f(x) = sen(5πx), n = 13



f(x) = sen(5πx), n = 20



f(x) = sen(5πx), n = 27

9.4. Splines

El fenómeno de Runge muestra que no es buena idea utilizar la interpolación con n grande como instrumento de aproximación genérico, porque la coincidencia en los nodos es compatible con un comportamiento alocado cerca de ellos. Tiene más sentido hacer una división de los nodos en grupos pequeños y en cada uno de ellos llevar a cabo interpolación con un grado bajo. Para evitar picos en los extremos de los dominios de los polinomios, es necesario introducir ciertas condiciones y es ahí donde aparecen los *splines*.

El comando en `matlab/octave` para definir un polinomio a trozos es `mkpp(x,A)` donde `x` son los nodos en los que se cambia de polinomio y las filas de `A` son los coeficientes de los polinomios suponiendo el origen en el nodo, esto significa que en vez de los coeficientes de $p(x)$ debemos usar los de $p(x + x_j)$. Para evaluar un polinomio definido a trozos `pt` en una lista de

valores `t` se usa `ppval(pt,t)`. Por ejemplo, consideremos

$$f(t) = \begin{cases} t^2 & \text{si } -1 \le t < 0, \\ t & \text{si } 0 \le t < 1, \\ 1 & \text{si } 1 \le t < 2, \\ (3-t)^3 & \text{si } 2 \le t \le 3. \end{cases}$$

Para dibujar su gráfica, podríamos usar

```
A = [0,1,-2,1; 0,0,1,0; 0,0,0,1; -1,3,-3,1];
pp = mkpp( -1:3, A);
t = linspace(-1,3,300);
plot(t, ppval(pp,t));
axis('equal')
```

Los elementos de la primera fila de `A` vienen de los coeficientes de $\big(x+(-1)\big)^2$ y los de la última, de los coeficientes de $\big(3-(x+2)\big)^3$.

Dados $\{(x_j, y_j)\}_{j=0}^{n}$, en su versión más simple y empleada, la interpolación con *splines* (*cúbicos naturales*) es la que se consigue mediante una función f tal que $f\big|_{[x_j,x_{j+1}]}$ es un polinomio de grado menor o igual que tres, f, f', f'' son continuas y $f''(x_0) = f''(x_n) = 0$. Se puede probar que solo hay una función que satisface estas propiedades. Además, tal función minimiza la norma 2 de la derivada segunda [4, §2.1.3]. Esto es muy interesante en términos prácticos porque la derivada segunda tienen que ver con la curvatura y entonces los splines tienden a curvarse lo menos posible, en algún sentido, lo cual, como examinaremos a continuación, nos libra del comportamiento salvaje de la interpolación que hemos observado en $f(x) = 1/(1+x^2)$.

En `matlab/octave` la función `spline(x,y)` nos da el spline cúbico como polinomio definido a trozos que podemos evaluar con `ppval`. Veamos su efecto sobre el ejemplo del fenómeno de Runge:

```
n = 10;
x = linspace(-5,5, n+1);
y = 1./(1+x.^2);
cpi = polyfit(x,y,n);

figure(1)
t = linspace(-5,5,200);
sc = spline(x,y);
plot(x,y,'bo',t,polyval(cpi,t), '-', t, ppval(sc,t), '-')
```

Para favorecer la comparación se incluye también lo obtenido con el polinomio de interpolación y los (x_j, y_j). Si añades la gráfica de $f(x) = 1/(1+x^2)$ verás que es indistinguible a simple vista de lo obtenido con la interpolación con splines. En [17, §3.3] puedes encontrar un programa que no usa la función nativa de `spline` y es un poco más flexible.

ACTIVIDAD 9.4.1. *Dibuja la gráfica obtenida al interpolar con splines* $\{(x_j, y_j)\}_{j=0}^{2K}$ *donde* $x_j = j - K$, $y_j = x_j$ *excepto por* $y_K = 0{,}01$. *Compara el resultado con el del polinomio interpolador.*

El cálculo de los coeficientes de los polinomios cúbicos que conforman los splines se reduce a resolver un sistema $n \times n$ [19, §2.4], pero es un sistema muy especial, con matriz tridiagonal, que requiere mucho menos esfuerzo computacional que uno genérico, lo cual añade la rapidez a las ventajas de los splines.

9.5. Curvas de Bézier y B-splines

Dados cuatro puntos P_0, P_1, P_2 y P_3, la curva definida paramétricamente por

$$\sigma(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3 \qquad 0 \leq t \leq 1,$$

conecta P_0 y P_3 y los segmentos P_0P_1 y P_2P_3 son tangentes a la curva. Además el cuadrilátero determinado por los P_j contiene a la curva. Los puntos P_1 y P_2 son los llamados *puntos de control*, no son interpolados por la curva, pero la atraen en cierta forma.

El siguiente código dibuja el caso correspondiente a $P_0 = (0,0)$, $P_1 = (1,1)$, $P_2 = (3,1)$ y $P_4 = (4,1/2)$.

```
P = [0,0; 1,1; 3,1; 4,1/2]

figure(1)
plot( P(:,1), P(:,2), 'o', P(:,1), P(:,2), '--')
hold on
t = linspace(0,1, 300)';
R = [(1-t).^3, 3*t.*(1-t).^2, 3*t.^2.*(1-t), t.^3]*P;
plot( R(:,1), R(:,2) )
hold off
```

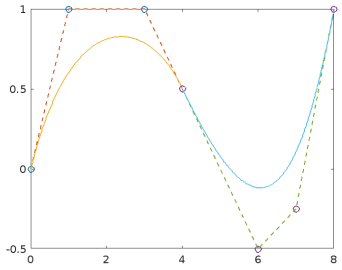
Si ahora concatenamos esta curva con otra correspondiente a los puntos P_3, P_4, P_5 y P_6 de forma que P_4 esté alineado con P_2P_3, el resultado será derivable, no tendrá picos. La generalización a $P_0, \ldots, P_n$ con $n = 3k$ conforma lo que se llama una *curva de Bézier*. Estas se usan a menudo en programas de diseño gráfico interactivos en los que se permite mover los puntos de control conservando la alineación para modificar la forma de la curva. Las propiedades anteriores hacen que el procedimiento sea bastante intuitivo y, con un poco de práctica, se logre aproximar bien cualquier curva con unos cuantos puntos de interpolación y control.

El siguiente código muestra lo que ocurre cuando se completa el anterior con nuevos puntos. Nótese que $P_2 = (3,1)$, $P_3 = (4,1/2)$ y $P_4 = (6,-1/2)$ son colineales.

```
P = [0,0; 1,1; 3,1; 4,1/2; 6,-1/2; 7,-1/4; 8,1]

figure(1)
n = (size(P,1)-4)/3;
for j = 0:n
        r = 3*j+1:3*j+4;
        plot( P(r,1), P(r,2), 'o', P(r,1), P(r,2), '--')
        hold on
        t = linspace(0,1, 300)';
```

```
10          R = [(1-t).^3, 3*t.*(1-t).^2, 3*t.^2.*(1-t),
                ↪ t.^3]*P(r,:);
11          plot( R(:,1), R(:,2) )
12          hold on
13      end
14      hold off
```

El código funciona con cualquier lista de puntos `P`, siempre con un número de filas del tipo $3k+1$. La salida del código, tal como está, es la figura



ACTIVIDAD 9.5.1. *Dibuja la curva de Bézier que interpola* $P_{3j} = (3j, 0)$, $0 \le j \le 2J$ *con puntos de control* $P_{3j+1} = \big(3j+1, (-1)^j\big)$, $P_{3j+2} = \big(3j+2, (-1)^j\big)$. *Con este propósito, quizá te sea útil utilizar* `repmat(v,1,r)` *que aplicado a un vector fila* `v` *lo repite* `r` *veces multiplicando su dimensión por esta cantidad.*

Las curvas de Bézier no interpolan todos los P_j y los llamados *B-splines* exageran esta situación pues, genéricamente no interpolan ninguno de los puntos de partida, solo los aproximan.

Para motivar e introducir los B-splines, fijémonos solo en un caso particular, pero importante, en que tenemos puntos de interpolación $\big\{(x_j, y_j)\big\}_{j=0}^{n}$ con $x_{j+1} - x_j = h$ constante. Supongamos que n es gigantesco y que no estamos dispuestos a hacer el esfuerzo computacional de resolver el sistema requerido por los *splines*. Resulta que la función definida en $[x_0, x_{n-1}]$ mediante

$$f(t) = \sum_{k=1}^{4} Q_k\big((t - x_k)/h\big) y_{j+3-k} \qquad \text{para} \quad t \in [x_j, x_{j+1}], \quad 0 < j < n-1,$$

es un buen sustituto sencillo para los splines tomando

$$Q_1(t) = \frac{1}{6}t^3,\ Q_2(t) = \frac{1}{6}(t+1)^3 - \frac{2}{3}t^3,\ Q_3(t) = Q_2(1-t),\ Q_4(t) = Q_1(1-t).$$

Un cálculo muestra

$$f(x_j) = \frac{1}{6}\big(y_{j-1} + 4y_j + y_{j+1}\big).$$

Si h es pequeño y los y_j provienen de una función regular, los y_j presentarán variaciones pequeñas y se cumplirá aproximadamente $f(x_j) = y_j$. Aunque te

resulte chocante, operaciones de este tipo están funcionando todos los días en las entrañas de nuestros dispositivos digitales cuando ampliamos una foto [4, §2.1.3].

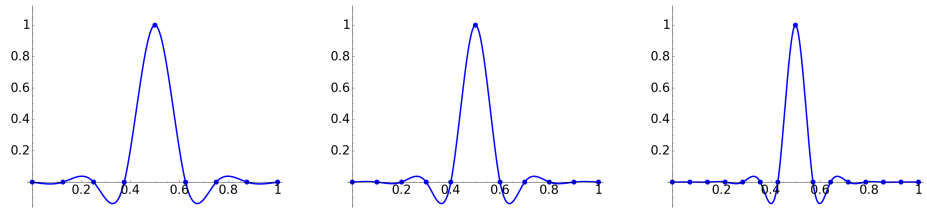
El siguiente código, no muy elegante, muestra el resultado para $n = 10$ y la función $y = \operatorname{sen} x$ en $[-\pi, \pi]$.

```
q1 = @(x) x.^3/6;
q2 = @(x) (x+1).^3/6-2*x.^3/3;
q3 = @(x) q2(1-x);
q4 = @(x) q1(1-x);

n = 10;
x = linspace(-pi,pi, n+1);
y = sin(x);
h = x(2)-x(1);

figure(1)
plot(x,y,'bo')
hold on
t = linspace(-pi,pi,100);
plot(t,sin(t), 'r--')
hold on

for j=1:n-2
        t = linspace(x(j+1), x(j+2),10 );
        tn = (t-x(j+1))/h;
        plot(t,q1(tn)*y(j+3) + q2(tn)*y(j+2) +
            ↪ q3(tn)*y(j+1) + q4(tn)*y(j))
        hold on
end

hold off
```

La función $b(t)$ definida por $Q_1(t+2)$, $Q_2(t+1)$, $Q_3(t)$ y $Q_4(t-1)$, respectivamente en $[-2,-1]$, $[-1,0]$, $[0,1]$ y $[1,2]$, y cero en el resto, es lo que se llama *B-spline*. Es un spline en el sentido de que es un polinomio cúbico a trozos con derivada segunda que además se anula en los extremos.

ACTIVIDAD 9.5.2. *Dibuja la gráfica de $b(t)$ en $[-2,2]$.*

Es importante notar que al utilizar B-splines los resultados son locales: el valor en un punto solo depende de la función en unos pocos nodos adyacentes, mientras que en el caso de los splines cúbicos que hemos visto antes, cualquier modificación de la función en un solo nodo cambia el sistema que hay que resolver y afecta a todo el resultado. La posibilidad de sustituir con éxito los splines por B-splines radica, en parte, en que si el número de nodos es grande, los splines son casi locales. Un cambio en un solo nodo tiene una influencia numéricamente pequeña en cuanto nos alejamos un poco.



$n = 8$, $y_4 = 1$, $y_j = 0$ $\quad$ $n = 10$, $y_5 = 1$, $y_j = 0$ $\quad$ $n = 14$, $y_7 = 1$, $y_j = 0$

Para terminar, algunos comentarios para el que tenga curiosidad en la nomenclatura y la historia.

Las curvas de Bézier deben su nombre al ingeniero francés que las usó extensivamente en el diseño de automóviles en la década de 1960. En realidad, desde el punto de vista matemático, su origen teórico está en los *polinomios de Bernstein*, que son bastante anteriores.

La función f construida a partir de los Q_k se puede expresar como combinación lineal de versiones convenientemente escaladas y trasladadas de b y, en conexión con esto, la terminología B-spline abrevia *basis spline* en inglés, pues a partir de b se obtiene una base del espacio vectorial de splines. Por otro lado, *spline* era el nombre que recibían las plantillas para trazar curvas usadas en ingeniería industrial. Hace años era habitual que los escolares tuvieran splines en este sentido, materializados en unas placas plásticas con formas caprichosas que llamaban plantillas de curvas (aunque, al parecer, el nombre técnico es *plantillas Burmester*), las cuales, usadas con pericia, permitían dibujar elipses y formas más complicadas. Posiblemente los gráficos con ordenador han hecho que ahora no sean comunes.

Apéndice A

Soluciones a las actividades

A.1. Actividades del capítulo 2

2.1.1

```
% h^5 debe ser mayor que 10*eps donde el 10 viene del mayor
        ↪ coef.

h = 2*0.0011730;
x = linspace(1-h,1+h,100);
y = x.^5 - 5*x.^4 + 10*x.^3 - 10*x.^2 + 5*x - 1;
plot(x,y)
```

2.1.2

```
% Discretización muy grande. Cuanta mayor es la
        ↪ discretización más son los puntos de
        ↪ impresiónentre los que se interpola y en general
        ↪ dan valores erróneos.

h = 1e-5;
x = linspace(1-h,1+h,1000);
y = x.^5 - 5*x.^4 + 10*x.^3 - 10*x.^2 + 5*x - 1;
plot(x,y)

% Cambio de signo (no hay cancelación)

h = 1e-5;
x = linspace(1-h,1+h,100);
y = x.^5 - 5*x.^4 + 10*x.^3 - 10*x.^2 + 5*x + 1;
plot(x,y)
```

2.1.3

```
% En el primero \sqrt{1+a_{n-1}^2}-1 es aproximadamente
% \sqrt(1+ \pi^2 2^{-2n}) -1 \approx
% (1+\pi^2 2^{-2n-1}) -1
% que debe dar errores relativos grandes cuando
% \pi^2 2^{-2n-1} \approx eps
% Despejando con eps = 2^{-52} se tiene n =27.151
```

2.1.4

```
function y = alg1(x)
```

```
    % Primera fórmula
    y = (sqrt(1+x^2)-1)/x;
end
```

```
function y = alg2(x)
    % Segunda fórmula
    y = x/(sqrt(1+x^2)+1);
end
```

2.1.5

```
% aproximando tan(x) por x +x^3/3 (Taylor)
% 2^{n+1}a_n -\pi \approx \pi^3/3/2^{2n+2}

% La predicción funciona bien hasta la iteración 24
anp = 1;
for k = 2:28
    anp = anp/(sqrt(1+anp^2)+1);
    res = 2^(k+1)*anp;
        pred = pi^3/3/2^(2*k+2) ;
    disp(['Iteración ' num2str(k) ' -> ' num2str(res - pi,
        ↪ ' %.10e') ' -> ' num2str(pred, ' %.10e')])
end
```

2.2.1

```
% Matriz
A = [1,2,-1; 2, 5, 1; -1,1,11 ];
% Inversa con Gauss-Jordan sobre [A I]
E = rref( [A, eye(3)] );
% Esto muestra las últimas tres columnas
E(:,4:6)

% Inversa con un comando directo (también A^-1)
inv(A)
```

2.3.1

```
% Se busca que los pivotes sean grandes
A = [10,1; 1,1];
[L,U] = lu(A);
disp(L)

% Se busca que los pivotes sean grandes
A = [1,10; 3,1];
[L,U] = lu(A);
disp(L)
```

A.2. Actividades del capítulo 3

3.1.1

```
% Tomamos por ejemplo
A = [1,2,0;2,4,1;1,0,1]
% En octave el error es
```

```
4  % warning: division by zero
5  % En matlab simplifimente aparece  NaN e Inf en las matrices
```

3.2.1

```
1  % No funciona con v = k:n
2  % A(k+1,k) está por debajo de la diagonal y por tanto
3  % no corresponde a U. Estaríamos sobreescribiendo valores
       ↪ de L.
4  % En el algortimo original
5  % L(k+1,k) = U(k+1,k)/U(k,k) (línea 8)
6  % U(k+1,k) = U(k+1,k) - L(k+1,k)*U(k,k) (línea 10)
7  % hace que U(k+1,k)=0 y no hay problema.
```

3.3.1

```
1  function x = ltrs(A, b)
2      % Resuelve sistema con matriz triangular inferior
3      n = size(A,1);
4      x = zeros(n, 1);
5      x(1) = b(1)/A(1,1);
6      for ii = 2:n
7          x(ii) = (b(ii) - A(ii,1:ii-1)*x(1:ii-1) )/A(ii,ii);
8      end
9  end
```

```
1  function x = utrs(A, b)
2      % Resuelve sistema con matriz triangular superior
3      n = size(A,1);
4      x = zeros(n, 1);
5      x(n) = b(n)/A(n,n);
6      for ii = n-1:-1:1
7          x(ii) = (b(ii) - A(ii,(ii+1):n)*x((ii+1):n)
               ↪ )/A(ii,ii);
8      end
9  end
```

3.3.2

```
1  A = [1,2,1;3,4,5;5,8,1];
2  b = [4;12;14];
3  [L,U] = mlu(A);
4  y = ltrs(L,b);
5  x = utrs(U,y);
6  disp(x)
```

```
1  function [L,U] = mlu(A)
2      % Descomposición LU
3      n = size(A,1);
4      U = A;
5      L = eye(n);
6
7      for k = 1:n-1
8          v = k:n;
9          for ii = k+1:n
10             L(ii,k) = U(ii,k)/U(k,k);
11             U(ii,v) = U(ii,v) - L(ii,k)*U(k,v);
12         end
13     end
14 end
```

3.4.1

```
N = 1000;
x = linspace(0,4,N);
f = x.^2.*exp(-x);
[m,j] = max(f);
disp(m)
disp(x(j))
```

3.4.2

```
function [L,U,P] = plu(A)
% Descomposición LU con pivotaje
    n = size(A,1);
    U = A;
    L = eye(n);
    P = eye(n);

    for k = 1:n-1
        [m,j] = max( abs(U(k:n,k)) );

        if j~=1
                r = j+k-1;
                U([k,r], k:n) = U([r,k], k:n);
                L([k,r], 1:k-1) = L([r,k], 1:k-1);
                P([k,r], :) = P([r,k], :);
        end

        v = k:n;
        for ii = k+1:n
            L(ii,k) = U(ii,k)/U(k,k);
            U(ii,v) = U(ii,v) - L(ii,k)*U(k,v);
        end
    end
end
```

3.5.1

```
% Matlab online
% Elapsed time is 0.099433 seconds.
% Elapsed time is 1.427905 seconds.
%
% Recorrer todas las filas de una columna fijada es más
    ↪ rápido
% => se almacena por columnas, como en Fortran.
```

https://www.mathworks.com/help/matlab/matlab_external/matlab-data.html

3.5.2

```
% Matlab online
N = 1000;
A = rand(N);
tic
mlu(A);
toc

tic
mlut(A);
toc
```

```
function [L,U] = mlut(A)
    % Descomposición L^tU^t traspuestas
    n = size(A,1);
    U = transpose(A);
    L = eye(n);

    for k = 1:n-1
        v = k:n;
        for ii = k+1:n
            L(k,ii) = U(k,ii)/U(k,k);
            U(v,ii) = U(v,ii) -L(k,ii)*U(v,k);
        end
    end
end
```

```
% Matlab online
% Elapsed time is 7.671028 seconds.
% Elapsed time is 5.264230 seconds.
%
% El algoritmo es más rápido
```

A.3. Actividades del capítulo 4

4.1.1

```
N = 10;
A = rand(N) + i*rand(N);
disp( sqrt( max( eig(A'*A) ) ) )
disp( norm(A) )
```

4.1.2

```
N = 3;
maxp = 0;
for j= 1:1000
        A = rand(N);
        B = rand(N);
        res = norm(A*B,5)/norm(A,5)/norm(B,5);
        if res>maxp
                maxp = res;
        end
end
disp(['El máximo es ' num2str(maxp)])
```

4.2.1

```
format long
p = 1;
q = 1;
for k = 1:6
        ptemp = p^2+2*q^2;
        q = 2*p*q;
        p = ptemp;
        disp(['num/den = ' num2str(p) '/' num2str(q)])
end
% En la sexta iteración ya los números son
% mayores que 2^{63} y por tanto no fiables.
```

4.2.2

```
% Para a<1 converge para todo x_0.
% Para a>1 diverge (para x_0 ~= la solución).
% Para a=1 no hay ecuación.
N = 5;
x = 1;
p1 = [];
p2 = [];
for k = 1:N
        % (x_n,x_{n+1}), (x_{n+1},x_{n+1})
        p1 = [p1,x];
        x = (x+8)/3;
        p1 = [p1,x];
        p2 = [p2,x,x];
end

figure(1)
plot(p1,p2,'--o')
hold on
t = linspace(1,4,10);
plot(t,t, t,(t+8)/3)
hold off
```

4.3.1

```
N = 5;
A = [5,2,1; -3,1,1; 2,3,1];
b = [10;6;20];

n = size(A,1);
Ap = A - diag(diag(A));
Dinv = diag(A).^-1;

% Si el radio espectral es menor
% que 1, no converge en general
r = max(abs(inv(diag(diag(A)))*eig(Ap)))
disp(['Radio espectral = ', num2str(r)])

x = zeros(n,1);
for k = 1:N
        x = Dinv.*(b-Ap*x);
        disp(norm(x))
end
```

4.3.2

```
function [x,err] = mjac (A, b, Nmax, tol)
        n = size(A,1);
        Ap = A - diag(diag(A));
        Dinv = diag(A).^-1;

        x = zeros(n,1);
        for k = 1:Nmax
                x_old = x;
                x = Dinv.*(b-Ap*x);
                err = norm(x-x_old)/norm(x_old+eps);
                if err < tol
                        return
                end
        end
end
```

4.4.1

```
function [x,err] = mgase (A, b, Nmax, tol)
        n = size(A,1);

        x = zeros(n,1);
        for k = 1:Nmax
                x_old = x;
                for ii = 1:n
                        x(ii) = (b(ii) -
                                ↪ A(ii,1:ii-1)*x(1:ii-1) -
                                ↪ A(ii,ii+1:n)*x(ii+1:n)
                                ↪ )/A(ii,ii);
                end
                err = norm(x-x_old)/norm(x_old+eps);
                if err < tol
                disp(['Se han usado ',num2str(k),'
                        ↪ iteraciones'])
                        return
                end
        end
end
```

4.5.1

```
N = 10
n = 100
A = rand(n) + (n-1)*eye(n);
b = rand(n,1);
x_exact = A\b;

% xj -> Jacobi
% xgs -> Gauss-Seidel

xgs = zeros(n,1);
xj = zeros(n,1);
Ap = A - diag(diag(A));
Dinv = diag(A).^-1;

for k = 1:N
        xj = Dinv.*(b-Ap*xj);
        for ii = 1:n
                xgs(ii) = (b(ii) - A(ii,1:ii-1)*xgs(1:ii-1)
                        ↪ - A(ii,ii+1:n)*xgs(ii+1:n)
                        ↪ )/A(ii,ii);
        end
        e1 = norm(xj-x_exact);
        e2 = norm(xgs-x_exact);
        disp( ['k = ' num2str(k) ' -> J: '
                ↪ num2str(e1,'%.5e') ' -> G-S: '
                ↪ num2str(e2,'%.5e')]  )
end
```

4.6.1

```
% Número iteraciones
N = 100
% Sistema
A = [2,2,1; 8,4,1; 4,7,-4];
b = [6; 14; 3];

% número de filas
m = size(A,1);
% número de comunas
n = size(A,2);

% Vector inicial
x = zeros(n,1);

```

```
for k = 1:m
        no = 1/norm(A(k,:));
        A(k,:) = A(k,:)*no;
        b(k) = b(k)*no;
end

for k = 1:N
        ii = mod(k,m) +1;
        fila = A( ii , : );
        x = x + (b(ii) - fila*x)*fila';
end

disp(x)
```

A.4. Actividades del capítulo 5

5.1.1

```
N = 300

A = rand(N);

tic
m = size(A,2);
Q = A;
Q(:,1) = A(:,1)/norm(A(:,1));

for k = 2:m
        u_k = A(:,k);
        for j = 1:k-1
                u_k = u_k - Q(:,j)' * A(:,k) * Q(:,j);
        end
        Q(:,k) = u_k/norm( u_k );
end
toc

tic
orth(A);
toc
```

Salida (aleatoria) para varios valores de `N`:

```
N =  300
Elapsed time is 0.520701 seconds.
Elapsed time is 0.0822821 seconds.

N =  600
Elapsed time is 2.09508 seconds.
Elapsed time is 0.631554 seconds.

N =  900
Elapsed time is 4.96111 seconds.
Elapsed time is 2.29034 seconds.
```

5.1.2

```
A = [1,1,0;1,2,0;-1,3,1]
tol = 1e-10;
for j1 = 1:size(A,2)
        for j2 = j1+1:size(A,2)
                if abs(A(:,j1)'*A(:,j2))>tol
```

```
6                               disp([num2str(j1) ' y ' num2str(j2)
                                    ↪ ' no son ortogonales'])
7                       else
8                               disp([num2str(j1) ' y ' num2str(j2)
                                    ↪ ' son ortogonales'])
9                       end
10              end
11      end
```

5.2.1

```
1  N = 4
2  k = 2
3
4  v = zeros(N,1);
5  while v'*v==0
6          v = round(k*(2*rand(N,1)-1));
7  end
8  d = v'*v;
9  Q = d*eye(N)-2*v*v';
10 disp('La matriz es')
11 disp(Q)
12 disp(['dividida por ' num2str(d)])
```

5.2.2

```
1  function v = vh(x)
2          v = x;
3          if v(1)<0
4                  v(1) = v(1)-norm(x);
5          else
6                  v(1) = v(1)+norm(x);
7          end
8          v = v/norm(v);
```

5.3.1

```
1  times1 = [];
2  times2 = [];
3  rang = 100:10:500;
4
5  for N = rang
6          A = rand(N);
7
8          t0 = tic;
9          [Q,R] = qr(A);
10         times1 = [times1,toc(t0)];
11
12         t0 = tic;
13         [Q,R] = mQR(A);
14         times2 = [times2,toc(t0)];
15 end
16
17 plot( rang, times2./times1)
```

5.3.2

```
1  m = 3
2  n = 3*m
3
4  A = rand(n,m);
```

```
5  [Q,R] = mQR(A);
6
7  eig(Q'*Q)
8  eig(Q*Q')
```

Son m unos en el primes caso, completados con $n - m$ ceros en el segundo.

5.3.3

```
1  times1 = [];
2  times2 = [];
3  rang = 100:10:500;
4
5  for N = rang
6          A = rand(N);
7
8          t0 = tic;
9          [Q,R] = qr(A);
10         times1 = [times1,toc(t0)];
11
12         t0 = tic;
13         [Q,R] = mQR2(A);
14         times2 = [times2,toc(t0)];
15 end
16
17 plot( rang, times2./times1)
```

5.4.1

```
1  times1 = [];
2  times2 = [];
3  rang = 1000:100:3000;
4
5  for N = rang
6          A = rand(N);
7          b = rand(N,1);
8
9          t0 = tic;
10         [L,U,P] = lu(A);
11         times1 = [times1,toc(t0)];
12
13         t0 = tic;
14         [Q,R] = qr(A);
15         times2 = [times2,toc(t0)];
16 end
17
18 plot( rang, times2./times1)
```

5.4.2

```
1  N = 10
2  A = rand(N);
3  b = rand(N,1);
4
5  [L,U,P] = lu(A);
6  y = ltrs(L,P*b);
7  x = utrs(U,y);
8
9  norm(x-A\b)
```

A.5. Actividades del capítulo 6

6.1.1

```
A = [1,2; 1,1; 3,1];
b = [2;8;6];

[Q,R] = qr(A);
m = size(A,2);
T = R(1:m,:);
Qm = Q(:,1:m);

utrs(T, Qm'*b)
```

6.1.2

```
function res = rrerr(x,y)
    % Error en la recta de regresión
        n = size(x,1);
        X = [ones(n,1), x];
        c = inv(X'*X)*X'*y;
        res = max( abs(c(1)+c(2)*x-y) );
end
```

No se cumple `rrerr(x,y)=rrerr(y,x)` en general.

```
N = 6
x = (1:N)';
y = 2*x + [1;zeros(N-1,1)];

rrerr(x,y)
rrerr(y,x)
```

6.2.1

```
N = 10
A = rand(2*N,N)+i*rand(2*N,N);
b = rand(2*N,1)+i*rand(2*N,1);
[U,D,V] = svd(A);
E = diag( diag(D) );
norm( V*inv(E)*U(:,1:N)'*b - A\b )
```

6.2.2

```
A = rand(100,200);
res = zeros(99,1);

[U,D,V] = svd(A);

for r = 99:-1:1
        D(r+1,r+1)=0;
        Ar = U*D*V';
        res(r) =  norm(A-Ar,'fro');
end

figure(1)
plot(res);
```

6.2.3

```
A = [0.546, -0.722; 0.783, 0.427];

x = [2,3,3,2,2; 0,0,1,1,0];
y = A*x;

[U,D,V] = svd(A);

z = U*V'*x;

figure(1)
plot(x(1,:), x(2,:))
hold on
plot(y(1,:), y(2,:))
hold on
plot(z(1,:), z(2,:))
hold off
axis('equal')
```

6.2.4

```
N = 5;
M = 3;
A = rand(N,M)+i*rand(N,M);

[V,D] = eig(A'*A);
[U,D] = qr(A*V);

norm(A-U*D*V','fro')
```

A.6. Actividades del capítulo 7

7.1.1

```
% Crea la matriz
N = 100;
A = diag( ones(N-1,1), 1 );
A = 2*eye(N) + A + A';

% ordena los autovalores
v = sort(eig(A));

% los dibuja
plot(v)
```

7.2.1

```
A1 = [-7,6,-8,5; -16,10,-13,10; -5,2,-2,3; -8,6,-8,6];
A = -A1;
N = 8;
v = rand(size(A,1),1);
v = v/norm(v);

tol = 0.1;
fl = false;
for k = 1:N
```

```
10          vn = A*v;
11          [m,j] = max(abs(vn));
12          v = v/v(j);
13          vn = vn/vn(j);
14          if norm(v-vn) < tol*norm(vn)
15                  fl = true;
16                  v = vn/norm(vn);
17                  break
18          end
19          v = vn/norm(vn);
20  end
21
22
23  if fl
24          disp(['Se ha alcanzado la tolerancia exigida con N
                    ↪ = ' num2str(k)])
25  else
26          disp('No se ha alcanzado la tolerancia exigida')
27  end
28
29  disp('Aproximación del autovector:')
30  disp(v)
31
32  disp('Aproximación del autovalor:')
33  disp(v'*A*v)
```

7.3.1

```
% Basta añadir esto antes de axis('equal')
plot(real(eig(A)),imag(eig(A)),'o');
```

7.3.2

```
1  A3 = [5,0,-2,1; 0,5,-1,2; -2,-1,5,0; 1,2,0,5];
2  R = rand(4);
3  ep = 0.1;
4  P = ep*(R+R');
5  A = A3+P;
6
7  [C,D] = eig(A3);
8  [Cp,D] = eig(A);
9
10 disp('Diferencia real de los autovalores')
11 disp(eig(A)-eig(A3))
12
13 disp('Diferencia estimada de los autovalores')
14 disp( diag(C'*P*C) )
```

7.4.1

```
1  A1 = [-7,6,-8,5; -16,10,-13,10; -5,2,-2,3; -8,6,-8,6];
2  A = A1;
3  N = 5;
4  n = size(A,1);
5  v = rand(n,1);
6  v = v/norm(v);
7  w = v;
8
9  la = 0.8+0.8*i;
10 Ai = inv(A-la*eye(n));
11
12 for k = 1:N
13         fprintf('Iteración %d\n', k)
14
15         % Alg 1
```

```
16          wn = Ai*w;
17          w = wn/norm(wn);
18          fprintf('\tError alg1 %.6e\n', abs(w'*A*w-1-i))
19
20          % Alg 2
21          vn = (A-la*eye(n))\v;
22          v = vn/norm(vn);
23          la = v'*A*v;
24          fprintf('\tError alg2 %.6e\n', abs(la-1-i))
25  end
```

7.5.1

```
1   function [C,D] = eigt(A)
2       % Autovalores y autovectores de matriz
3       % triangular superior.
4       % Usa la función utrs
5           n = size(A,1);
6           C = zeros(n);
7
8           C(1,1) = 1;
9           for k=2:n
10                  C(k,k) = 1;
11                  T = A(1:k-1, 1:k-1) - A(k,k)*eye(k-1);
12                  C(1:k-1,k) = utrs( T, -A(1:k-1,k) );
13          end
14
15          D = diag(diag(A));
16  end
```

```
1   function x = utrs(A, b)
2       % Resuelve sistema con matriz triangular superior
3       n = size(A,1);
4       x = zeros(n, 1);
5       x(n) = b(n)/A(n,n);
6       for ii = n-1:-1:1
7           x(ii) = (b(ii) - A(ii,(ii+1):n)*x((ii+1):n)
                ↪ )/A(ii,ii);
8       end
9   end
```

7.5.2

```
1   % Dimensión
2   n = 100
3
4   % Número de iteraciones
5   N = 400;
6
7   % Matriz simétrica aleatoria
8   AA = rand(n);
9   AA = AA+AA';
10
11
12  tic
13  A = AA;
14  for k = 1:N
15          [Qp,R] = qr(A);
16          A =  R*Qp;
17  end
18  toc
19
20  % error
21  % disp( norm( sort(diag(A))-sort(eig(AA)) ) )
22
23
24  tic
25  A = AA;
26  [P,A] = hess(A);
```

```
for k = 1:N
        [Qp,R] = qr(A);
        A =  R*Qp;
end
toc

% error
% disp( norm( sort(diag(A))-sort(eig(AA)) ) )
```

A.7. Actividades del capítulo 8

8.1.1

```
N = 5;
c = -6/17
x = 1;
for k = 1:N
        x = c*(x^2 - 2) + x
end
disp(['Error absoluto (con signo) = ' num2str(x-sqrt(2))])
```

8.2.1

```
function xa = acel (x)
        xa = circshift(x,2) -
                ↪ (circshift(x,1)-circshift(x,2)).^2 ./
                ↪ (x-2*circshift(x,1)+circshift(x,2));
        xa(1) = x(1);
        xa(2) = x(2);
end
```

8.2.2

```
N = 10;

f = @(x) x-(x^2-2)/2;

x = ones(N,1);
x(1) = 1;
a(1) = 1;
for k = 1:N-1
        x(k+1) = f(x(k));
        a(k+1)= a(k)-( f(a(k))-a(k) )^2 / (
                ↪ f(f(a(k)))-2*f(a(k))+a(k) );
end

for k = 3:N
        disp('----------')
        disp(['Iteración ' num2str(k)])
        xa = x(k-2)-
                ↪ (x(k-1)-x(k-2))^2/(x(k)-2*x(k-1)+x(k-2));
        disp(['Error sin acelerar ' num2str(x(k)-sqrt(2))])
        disp(['Error acelerado ' num2str(xa-sqrt(2))])
        disp(['Error Aitken ' num2str(a(k)-sqrt(2))])
end
```

8.2.3

```
N = 4;

f = @(x) (x+2/x)/2;

x = ones(N,1);
x(1) = 1;
a(1) = 1;
for k = 1:N-1
        x(k+1) = f(x(k));
        a(k+1)= a(k)-( f(a(k))-a(k) )^2 / (
                ↪ f(f(a(k)))-2*f(a(k))+a(k) );
end

for k = 1:N
        disp('----------')
        disp(['Iteración ' num2str(k)])
        disp(['Error algoritmo ' num2str(x(k)-sqrt(2))])
        disp(['Error Aitken ' num2str(a(k)-sqrt(2))])
end
```

8.3.1

```
function x = bisec (f,a,b, err)
        while b-a>err
                c = (a+b)/2;
                if f(a)*f(c)<0
                        b = c;
                else
                        a = c;
                end
        end
        x = (a+b)/2;
end
```

8.4.1

```
figure(1)
A = juliak(3, 30, 400, 0.01);
subplot(1,3,1)
imshow(A);

A = juliak(4, 180, 400, 0.1);
subplot(1,3,2)
imshow(A);

subplot(1,3,3)
A = juliak(5, 380, 400, 0.1);
imshow(A);
```

```
function R = juliak (k, Nmax, N, tol)
        % k = degree
        % Nmax = number of steps in Newton-Raphson
        % N = number of pixels
        % tol = tolerance

        A = meshgrid(linspace(-3/2,3/2,N),1:N);
        A = A + i*A' + eps;
        z = 2^(1/k)*exp(2*pi*i*[0:k-1]/k);

        % Newton-Raphson
        for l = 1:Nmax
                A = (1-1/k)*A+2/k*A.^(1-k);
        end

        R = zeros(N);
        for l = 1:k
```

```
18                  R = R + l*(abs(A-z(l))<tol);
19          end
20
21          % normalization
22          R = im2double(mat2gray(R));
23  end
```

8.5.1

```
1  Nmax = 4;
2  x = zeros(Nmax,1);
3  x(1) = 1;
4  x(2) = 2;
5  for k = 3:Nmax
6          x(k) = x(k-1) - (x(k-1)^2-2)*(x(k-1)-x(k-2))
                  ↪ /(x(k-1)^2-x(k-2)^2);
7  end
8
9  figure(1)
10 t = linspace(x(1),x(2), 100);
11 plot( t, t.^2-2, 'g')
12 hold on
13 plot(x, x.^2-2)
14 hold off
```

A.8. Actividades del capítulo 9

9.1.1

```
1  K = 10
2  a = -K;
3  b = K;
4  x = -K:K;
5  y = x;
6  y(K+1) = 0.01;
7  n = 2*K;
8  w = zeros(n+1,1);
9  for j = 0:n
10         w(j+1)= (-1)^j*nchoosek(n,j)*2^n/(b-a)^n;
11 end
12
13 figure(1)
14 plot(x,y,'bo')
15 hold on
16 t = linspace(a,b,300);
17 plot( t,  lagr_bar(x, y, w, t) )
18 hold off
```

9.1.2

```
1  function w = wj (x)
2          n = length(x)-1;
3          w = zeros(1,n+1);
4          for j = 0:n
5                  temp = x(j+1)-x;
6                  temp(j+1) = [];
7                  w(j+1) = 1/prod(temp);
8          end
9  end
```

9.2.1

```
n = 4;
x = linspace(-1,1,n+1);

for j = 0:n
        temp = x;
        temp(j+1) = [];
        Lj = poly(temp)/prod(x(j+1)-temp);

        figure(j+1)
        plot(x,polyval(Lj,x),'bo')
        hold on
        t =  linspace(-1,1,200);
        plot(t, polyval(Lj,t))
        hold off
end
```

9.2.2

```
x = -2:2;
y = cos(x)+sin(x);

figure(1)
plot(x,y,'bo')
hold on
t = linspace(-2,2,100);
plot(t,cos(t)+sin(t), 'r--')
hold on
for n = 1:4
        cpi = polyfit(x,y,n);
        plot( t,  polyval(cpi,t) )
        hold on
end
hold off
```

9.3.1

```
for n=5:20
        x = linspace(-5,5, n+1);
        y = cos(x);
        cpi = polyfit(x,y,n);

        figure(1)
        t = linspace(-5,5,200);
        plot(t,cos(t), 'r--')
        hold on
        plot( t,  polyval(cpi,t) )
        hold on
end
hold off
```

9.3.2

```
n = 10;
x = linspace(-5,5, n+1);
g = poly(x);

figure(1)
t = linspace(-5,5,300);
p = abs(polyval(g,t));
plot( t,  p )
```

9.4.1

```
K = 10;
x = -K:K;
y = x;
y(K+1) = 0.01;

figure(1)
t = linspace(-K,K,200);
sc = spline(x,y);
plot(x,y,'bo', t, ppval(sc,t), '-')
```

9.5.1

```
J = 4;
P = zeros(6*J+1,2);
P(:,1) = 0:6*J;
P(:,2) = [repmat([0,1,1,0,-1,-1],1,J)'; 0];

figure(1)
n = (size(P,1)-4)/3;
for j = 0:n
        r = 3*j+1:3*j+4;
        plot( P(r,1), P(r,2), 'o', P(r,1), P(r,2), '--')
        hold on
        t = linspace(0,1, 300)';
        R = [(1-t).^3, 3*t.*(1-t).^2, 3*t.^2.*(1-t),
                ↪ t.^3]*P(r,:);
        plot( R(:,1), R(:,2) )
        hold on
end
hold off
```

9.5.2

```
q1 = @(x) x.^3/6;
q2 = @(x) (x+1).^3/6-2*x.^3/3;
q3 = @(x) q2(1-x);
q4 = @(x) q1(1-x);

t = linspace(0,1, 100);
plot(t-2,q1(t))
hold on
plot(t-1,q2(t))
hold on
plot(t,q3(t))
hold on
plot(t+1,q4(t))

hold off
```

Apéndice B

Entregas y sus soluciones

A lo largo del curso se propusieron tres entregas a completar en la hora de clase. Aquí están los enunciados y las soluciones. En cada caso hay tres modelos.

B.1. Enunciados de la primera entrega

Modelo 1 — Primera entrega de Cálculo Numérico I

Instrucciones. La duración de la prueba es una hora. Al terminar se deben subir a Moodle los tres ficheros correspondientes a los problemas. Se valorará que el código sea elegante y el menor gasto computacional.

---

1) Haz un programa en un fichero llamado `gra.m` que aproxime el dibujo de la gráfica de la función

$$f(x) = \frac{x+10}{x}\,\mathrm{sen}\left(\frac{3}{2}x\right) + e^{-x/(x+1)}$$

en $x \in [2, 6]$ con 200 puntos de interpolación y que además señale con un círculo la posición del máximo y del mínimo.

2) Escribe una función llamada `parmat` tal que su entrada sea una matriz A real $n \times 2n$ y su salida una matriz B de las mismas dimensiones tal que la primera fila de B sea la primera columna de A seguida de la segunda, la segunda fila de B sea la tercera columna de A seguida de la cuarta, y así sucesivamente. Añade además un condicional de modo que si A no es $n \times 2n$ la función muestre un aviso en pantalla y devuelva la matriz vacía `[]`.

3) Escribe una función llamada `utrs2` tal que aplicada a un vector columna $\vec{b}$ y a una matriz triangular superior A no singular 2-banda, en el sentido de que $a_{ij} = 0$ para $j \geq i + 2$, resuelva el sistema $A\vec{x} = \vec{b}$ por sustitución regresiva.

Modelo 2 — Primera entrega de Cálculo Numérico I

Instrucciones. La duración de la prueba es una hora. Al terminar se deben subir a Moodle los tres ficheros correspondientes a los problemas. Se valorará que el código sea elegante y el menor gasto computacional.

---

1) Escribe una función llamada `utrs2` tal que aplicada a un vector columna $\vec{b}$ y a una matriz triangular superior A no singular 2-banda, en el sentido de que $a_{ij} = 0$ para $j \geq i + 2$, resuelva el sistema $A\vec{x} = \vec{b}$ por sustitución regresiva.

2) Haz un programa en un fichero llamado `graf.m` que aproxime el dibujo de la gráfica de la función

$$f(x) = e^{x/(x+4)} \operatorname{sen}(2x) + \log(1 + x^2)$$

en $x \in [3/2, 5]$ con 200 puntos de interpolación y que además señale con un círculo la posición del máximo y del mínimo.

3) Escribe una función llamada `matpar` tal que su entrada sea una matriz A real $2n \times n$ y su salida una matriz B de las mismas dimensiones tal que la primera columna de B sea la primera fila de A seguida de la segunda, la segunda columna de B sea la tercera fila de A seguida de la cuarta, y así sucesivamente. Añade además un condicional de modo que si A no es $2n \times n$ la función muestre un aviso en pantalla y devuelva la matriz vacía `[]`.

Modelo 3 — Primera entrega de Cálculo Numérico I

Instrucciones. La duración de la prueba es una hora. Al terminar se deben subir a Moodle los tres ficheros correspondientes a los problemas. Se valorará que el código sea elegante y el menor gasto computacional.

1) Escribe una función llamada `parmat` tal que su entrada sea una matriz A real $n \times 2n$ y su salida una matriz B de las mismas dimensiones tal que la primera fila de B sea la primera columna de A seguida de la segunda, la segunda fila de B sea la tercera columna de A seguida de la cuarta, y así sucesivamente. Añade además un condicional de modo que si A no es $n \times 2n$ la función muestre un aviso en pantalla y devuelva la matriz vacía `[]`.

2) Escribe una función llamada `utrs2` tal que aplicada a un vector columna $\vec{b}$ y a una matriz triangular superior A no singular 2-banda, en el sentido de que $a_{ij} = 0$ para $j \geq i + 2$, resuelva el sistema $A\vec{x} = \vec{b}$ por sustitución regresiva.

3) Haz un programa en un fichero llamado `grafi.m` que aproxime el dibujo de la gráfica de la función

$$f(x) = \cos\left(\frac{x}{x+4}\right)\operatorname{sen}(3x) + \frac{x^2+1}{x+1}$$

en $x \in [3, 11/2]$ con 200 puntos de interpolación y que además señale con un círculo la posición del máximo y del mínimo.

B.2. Soluciones de la primera entrega

utrs2.m

```
1  function x = utrs2 ( b , A)
2    N = size(A, 1 ) ;
3    x = zeros(N, 1 ) ;
4    x(N) = b(N) /A(N, N) ;
5    for j = N-1: -1:1
6      x (j) = ( b (j) - A(j,j+1)*x( j+1)) /A( j , j ) ;
7    end
8  end
```

parmat.m

```
1  function B = parmat (A)
2    if 2* size(A, 1 ) == size(A, 2 )
3      % Extrae de A las columnas impares y pares
4      col_impar = A ( : , 1 : 2 : size(A, 2 ) ) ;
5      col_par = A ( : , 2 : 2 : size(A, 2 ) ) ;
6      % Construye B
7      B = [ col_impar' , col_par' ] ;
8    else
9      disp ( 'Dimensiones incorrectas' )
10     B = [];
11   end
12 end
```

matpar.m

```
function B = matpar (A)
  if size(A, 1 ) == 2* size(A, 2 )
    % Extrae de A las filas impares y pares
      fil_impar = A ( 1 : 2 : size(A, 1 ) , : );
      fil_par = A ( 2 : 2 : size(A, 1 ) , : );
      % Construye B
    B = [ fil_impar'; fil_par' ] ;
  else
    disp ( 'Dimensiones incorrectas' )
    B = [];
  end
end
```

---

gra.m

```
x = linspace (2 ,6 ,200) ;
y = (x+10)./x.* sin( 3/2 * x ) + exp( -x./( x+1) );
figure (1)
[ mi , jmi ] = min( y ) ;
[ ma , jma ] = max ( y ) ;
plot(x, y, '-', [ x(jmi), x(jma) ], [ mi, ma ], 'o' )
```

---

graf.m

```
x = linspace (3/2 ,5 ,200) ;
y = exp( x./(x+4) ).* sin( 2 * x ) + log(1+x.^2 ) ;
figure (1)
[ mi , jmi ] = min( y ) ;
[ ma , jma ] = max ( y ) ;
plot(x, y, '-', [ x(jmi), x(jma) ], [ mi, ma ], 'o' )
```

---

grafi.m

```
x = linspace (3 ,11/2 ,200) ;
y = cos( x./(x+4) ).* sin( 3 * x ) + ( x.^2 + 1 )./(x+1) ;
figure (1)
[ mi , jmi ] = min( y ) ;
[ ma , jma ] = max ( y ) ;
plot(x, y, '-', [ x(jmi), x(jma) ], [ mi, ma ], 'o' )
```

B.3. Enunciados de la segunda entrega

Modelo 1 — Segunda entrega de Cálculo Numérico I

Instrucciones. La duración de la prueba es una hora. Al terminar se deben subir a Moodle los tres ficheros correspondientes a los problemas. Se valorará que el código sea elegante.

---

1) Haz un programa en un fichero llamado `regr.m` que dibuje los puntos $\{(x_n, y_n)\}_{n=1}^{20}$ donde $x_n = 10 + \frac{11}{20}n$ e $y_n = x_n + \sin(x_n^2\sqrt{2022})$, marcándolos con un circulito, y, en la misma gráfica, dibuje la recta de regresión.

2) Escribe una función llamada `cursvd` cuyos argumentos sean una matriz cuadrada A y un parámetro $t > 0$, por ese orden, de forma que la salida sea una matriz con la misma descomposición SVD salvo que los valores singulares menores que t, los $d_{ii} < t$, están sustituidos por cero.

3) Escribe una función llamada `jab2` cuyos argumentos sean dos vectores reales $\vec{v}, \vec{b} \in \mathbb{R}^N$, $N > 2$, y su salida sea el resultado de aplicar el método de Jacobi con 10 iteraciones, partiendo de $\vec{x}^{(0)} = \vec{0}$, al sistema $A\vec{x} = \vec{b}$ donde los elementos de A son

$$a_{ij} = \begin{cases} 2 + v_i^2 & \text{si } i = j, \\ v_i & \text{si } |j - i| = 1, \\ 0 & \text{en el resto de los casos.} \end{cases}$$

Para mayor eficiencia, no debes crear la matriz A completa (que está llena de ceros). Es decir, la función no debe contener ninguna línea `A = ...` o similar.

Modelo 2 — Segunda entrega de Cálculo Numérico I

Instrucciones. La duración de la prueba es una hora. Al terminar se deben subir a Moodle los tres ficheros correspondientes a los problemas. Se valorará que el código sea elegante.

---

1) Escribe una función llamada `cursvd` cuyos argumentos sean una matriz cuadrada A y un parámetro $t > 0$, por ese orden, de forma que la salida sea una matriz con la misma descomposición SVD salvo que los valores singulares menores que t, los $d_{ii} < t$, están sustituidos por cero.

2) Escribe una función llamada `jab2` cuyos argumentos sean dos vectores reales $\vec{v}, \vec{b} \in \mathbb{R}^N$, $N > 2$, y su salida sea el resultado de aplicar el método de Jacobi con 10 iteraciones, partiendo de $\vec{x}^{(0)} = \vec{0}$, al sistema $A\vec{x} = \vec{b}$ donde

los elementos de A son

$$a_{ij} = \begin{cases} 2 + v_i^2 & \text{si } i = j, \\ v_i & \text{si } |j - i| = 1, \\ 0 & \text{en el resto de los casos.} \end{cases}$$

Para mayor eficiencia, no debes crear la matriz A completa (que está llena de ceros). Es decir, la función no debe contener ninguna línea `A = ...` o similar.

3) Haz un programa en un fichero llamado `regr.m` que dibuje los puntos $\{(x_n, y_n)\}_{n=1}^{20}$ donde $x_n = 10 + \frac{11}{20}n$ e $y_n = x_n + \sin(x_n^2\sqrt{2022})$, marcándolos con un circulito, y, en la misma gráfica, dibuje la recta de regresión.

Modelo 3 — Segunda entrega de Cálculo Numérico I

Instrucciones. La duración de la prueba es una hora. Al terminar se deben subir a Moodle los tres ficheros correspondientes a los problemas. Se valorará que el código sea elegante.

---

1) Escribe una función llamada `jab2` cuyos argumentos sean dos vectores reales $\vec{v}, \vec{b} \in \mathbb{R}^N$, $N > 2$, y su salida sea el resultado de aplicar el método de Jacobi con 10 iteraciones, partiendo de $\vec{x}^{(0)} = \vec{0}$, al sistema $A\vec{x} = \vec{b}$ donde los elementos de A son

$$a_{ij} = \begin{cases} 2 + v_i^2 & \text{si } i = j, \\ v_i & \text{si } |j - i| = 1, \\ 0 & \text{en el resto de los casos.} \end{cases}$$

Para mayor eficiencia, no debes crear la matriz A completa (que está llena de ceros). Es decir, la función no debe contener ninguna línea `A = ...` o similar.

2) Haz un programa en un fichero llamado `regr.m` que dibuje los puntos $\{(x_n, y_n)\}_{n=1}^{20}$ donde $x_n = 10 + \frac{11}{20}n$ e $y_n = x_n + \sin(x_n^2\sqrt{2022})$, marcándolos con un circulito, y, en la misma gráfica, dibuje la recta de regresión.

3) Escribe una función llamada `cursvd` cuyos argumentos sean una matriz cuadrada A y un parámetro $t > 0$, por ese orden, de forma que la salida sea una matriz con la misma descomposición SVD salvo que los valores singulares menores que t, los $d_{ii} < t$, están sustituidos por cero.

B.4. Soluciones de la segunda entrega

regr.m

```
x = 10+11/20*[1:20]';
y = x + sin( x.^2*sqrt(2022) );

X = [ones(20,1),x];
c = inv(X'*X)*X'*y;

figure(1)
plot(x,y,'o',x,c(1)+c(2)*x)
```

cursvd.m

```
function B = cursvd (A,t)
    [U,D,V] = svd(A);
    D = D.*(D>=t);
    B = U*D*V';
end
```

jab2.m

```
function x = jab2 (v,b)
    N = size(v,1);

    bm = b./(v.^2+2);
    am = v./(v.^2+2);
    x = zeros(N,1);

    for k = 1:10
        x = bm - am.*( [0;x(1:N-1)] + [x(2:N);0] )
    end
end
```

B.5. Enunciados de la tercera entrega

Modelo 1 — Tercera entrega de Cálculo Numérico I

Instrucciones. La duración de la prueba es una hora. Al terminar se deben subir a Moodle los tres ficheros correspondientes a los problemas. Se valorará que el código sea elegante.

1) Sea A la matriz cuadrada 100×100 que, para $1 \leq i, j \leq 100$, tiene $a_{ii} = 6$, $a_{ij} = -4$ si $|i-j| = 1$, $a_{ij} = 1$ si $|i-j| = 2$ y el resto de sus elementos nulos. Escribe un programa llamado `autov.m` que muestre el resultado de conectar los puntos $\{(x_j, y_j)\}_{j=1}^{100}$ donde $\vec{x}$ es el cuarto autovector e $\vec{y}$ es el sexto autovector con la ordenación y normalización que da el comando `eig`.

2) Escribe una función llamada `msec` con argumentos una función anónima `f` y tres números reales `x0`, `x1`, `tol`, que aplique el método de la secante a `f` partiendo de `x0`, `x1` y devuelva el primer x_{n+1} con $n \geq 1$ tal que $|x_{n+1}-x_n|$ sea menor que `tol`.

3) Recuerda que en la forma de Lagrange de la interpolación polinómica, dados los nodos $x_0 < x_1 < \ldots < x_n$, para cada $0 \leq j \leq n$ se definía

$$L_j(t) = \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{t - x_k}{x_j - x_k}.$$

Haz un programa llamado `ljs.m` que dibuje en una misma gráfica todos los L_j en el intervalo $[x_0, x_n] = [-1/2, 2]$ cuando $n = 5$ y los nodos están equiespaciados ($x_{j+1} - x_j$ constante).

Modelo 2 — Tercera entrega de Cálculo Numérico I

Instrucciones. La duración de la prueba es una hora. Al terminar se deben subir a Moodle los tres ficheros correspondientes a los problemas. Se valorará que el código sea elegante.

---

1) Escribe una función llamada `msec` con argumentos una función anónima `f` y tres números reales `x0`, `x1`, `tol`, que aplique el método de la secante a `f` partiendo de `x0`, `x1` y devuelva el primer x_{n+1} con $n \geq 1$ tal que $|x_{n+1}-x_n|$ sea menor que `tol`.

2) Recuerda que en la forma de Lagrange de la interpolación polinómica, dados los nodos $x_0 < x_1 < \ldots < x_n$, para cada $0 \leq j \leq n$ se definía

$$L_j(t) = \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{t - x_k}{x_j - x_k}.$$

Haz un programa llamado `ljs.m` que dibuje en una misma gráfica todos los L_j en el intervalo $[x_0, x_n] = [-1/2, 2]$ cuando $n = 5$ y los nodos están equiespaciados ($x_{j+1} - x_j$ constante).

3) Sea A la matriz cuadrada 100×100 que, para $1 \leq i, j \leq 100$, tiene $a_{ii} = 6$, $a_{ij} = -4$ si $|i-j| = 1$, $a_{ij} = 1$ si $|i-j| = 2$ y el resto de sus elementos nulos. Escribe un programa llamado `autov.m` que muestre el resultado de conectar los puntos $\{(x_j, y_j)\}_{j=1}^{100}$ donde $\vec{x}$ es el cuarto autovector e $\vec{y}$ es el sexto autovector con la ordenación y normalización que da el comando `eig`.

Modelo 3 — Tercera entrega de Cálculo Numérico I

Instrucciones. La duración de la prueba es una hora. Al terminar se deben subir a Moodle los tres ficheros correspondientes a los problemas. Se valorará que el código sea elegante.

---

1) Recuerda que en la forma de Lagrange de la interpolación polinómica, dados los nodos $x_0 < x_1 < \ldots < x_n$, para cada $0 \leq j \leq n$ se definía

$$L_j(t) = \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{t - x_k}{x_j - x_k}.$$

Haz un programa llamado `ljs.m` que dibuje en una misma gráfica todos los L_j en el intervalo $[x_0, x_n] = [-1/2, 2]$ cuando $n = 5$ y los nodos están equiespaciados ($x_{j+1} - x_j$ constante).

2) Sea A la matriz cuadrada 100×100 que, para $1 \leq i, j \leq 100$, tiene $a_{ii} = 6$, $a_{ij} = -4$ si $|i-j| = 1$, $a_{ij} = 1$ si $|i-j| = 2$ y el resto de sus elementos nulos. Escribe un programa llamado `autov.m` que muestre el resultado de conectar los puntos $\{(x_j, y_j)\}_{j=1}^{100}$ donde $\vec{x}$ es el cuarto autovector e $\vec{y}$ es el sexto autovector con la ordenación y normalización que da el comando `eig`.

3) Escribe una función llamada `msec` con argumentos una función anónima `f` y tres números reales `x0`, `x1`, `tol`, que aplique el método de la secante a `f` partiendo de `x0`, `x1` y devuelva el primer x_{n+1} con $n \geq 1$ tal que $|x_{n+1} - x_n|$ sea menor que `tol`.

B.6. Soluciones de la tercera entrega

`autov.m`

```
1 A = 3*eye(100) - 4*diag(ones(99,1),1) + diag(ones(98,1),2);
```

```
2  A = A + A';
3
4  [C,D] = eig(A);
5  figure(1)
6  plot( C(:,4), C(:,6) )
```

---

ljs.m

```
1  % Relacionado con la actividad 9.2.1
2  % Véase el código que la precede.
3
4  figure(1)
5  n = 5;
6  x = linspace(-1/2,2,n+1);
7  t = linspace(-1/2,2,200);
8  for j = 0:n
9          temp = x;
10         temp(j+1) = [];
11         Lj = poly(temp)/prod( x(j+1) - temp );
12         plot( t, polyval(Lj,t) )
13         hold on
14 end
15 hold off
```

---

msec.m

```
1  function x2 = msec(f,x0,x1,tol)
2          x2 = x0 + 2*tol;
3          % Dentro del bucle x0, x1 y x2 son x_{n-1}, x_{n} y
               ↪ x_{n+1}
4          while abs(x2-x0)>tol
5                  f1 = f(x1);
6                  x2 = x1 - f1*( x1 - x0 )/( f1 - f(x0) );
7                  x0 = x1;
8                  x1 = x2;
9          end
10 return
```

Bibliografía

[1] K. E. Atkinson. *An introduction to numerical analysis*. John Wiley & Sons, Inc., New York, second edition, 1989.

[2] J.-P. Berrut and L. N. Trefethen. Barycentric Lagrange interpolation. *SIAM Rev.*, 46(3):501–517, 2004.

[3] F. Chamizo. Modelización II (un pase de modelos). `http://matematicas.uam.es/~fernando.chamizo/libreria/libreria.html`, 2004.

[4] F. Chamizo. A course on signal processing. `http://matematicas.uam.es/~fernando.chamizo/libreria/libreria.html`, 2020.

[5] J. De La Horra. *Estadística Aplicada*. Díaz de Santos, tercera edición, 2003.

[6] J. F. Epperson. On the Runge example. *Amer. Math. Monthly*, 94(4):329–341, 1987.

[7] A. Gilat. *Matlab: una introducción con ejemplos prácticos*. Reverté, 2006.

[8] D. F. Griffiths. An Introduction to Matlab. `http://www.maths.dundee.ac.uk/ftp/na-reports/MatlabNotes.pdf`, 2015.

[9] E. Hernández, M. J. Vázquez, and M. A. Zurro. *Álgebra lineal y geometría*. Pearson, Madrid, 2012. 3a ed.

[10] E. Isaacson and H. B. Keller. *Analysis of numerical methods*. John Wiley & Sons, Inc., New York-London-Sydney, 1966.

[11] D. Kulkarni, D. Schmidt, and S.-K. Tsui. Eigenvalues of tridiagonal pseudo-Toeplitz matrices. *Linear Algebra Appl.*, 297(1-3):63–80, 1999.

[12] J.H. Mathews and K.D. Fink. *Métodos numéricos con MATLAB*. Prentice Hall, 1999.

[13] L. Mirsky. Symmetric gauge functions and unitarily invariant norms. *Quart. J. Math. Oxford Ser. (2)*, 11:50–59, 1960.

[14] R. Pratap. *Getting Started with MATLAB: A Quick Introduction for Scientists and Engineers*. Oxford University Press, 2002.

[15] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C*. Cambridge University Press, Cambridge, second edition, 1992. The art of scientific computing.

[16] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37 of *Texts in Applied Mathematics*. Springer-Verlag, Berlin, second edition, 2007.

[17] A. Quarteroni and F. Saleri. *Cálculo Científico con MATLAB y Octave*. Springer, Milan, 2007.

[18] A. Quarteroni, F. Saleri, and P. Gervasio. *Scientific computing with MATLAB and Octave*, volume 2 of *Texts in Computational Science and Engineering*. Springer, Heidelberg, 2014. Fourth edition [of MR2253397].

[19] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*, volume 12 of *Texts in Applied Mathematics*. Springer-Verlag, New York, second edition, 1993. Translated from the German by R. Bartels, W. Gautschi and C. Witzgall.

[20] Wikipedia contributors. Richardson extrapolation — Wikipedia, the free encyclopedia, 2020. [Online; accessed 1-May-2021].

Índice alfabético