¿Y a mí qué me importa?

(aplicaciones del álgebra lineal)

con resúmenes de teoría

Fernando Chamizo Lorente

Álgebra II, 1º de ingeniería informática

Curso 2008-2009

Prólogo

Entre las preguntas dinamiteras que se suelen disparar contra las investigaciones y resultados matemáticos está el famoso "¿Y eso para qué sirve?". Los matemáticos tampoco ayudamos mucho pues por una parte creemos en general que las Matemáticas son un arte con una justificación primordialmente estética, y por otro lado no solemos estar involucrados en sus aplicaciones reales que consideramos propiedad privada de los ingenieros.

Evidentemente cualquiera con un mediano conocimiento sabe que las aplicaciones están ahí y que las Matemáticas viven en objetos cotidianos convenientemente agazapadas para no asustar a las buenas gentes y nuestra era tecnológica y su paladín el ordenador han multiplicado esa presencia. Por ejemplo, los códigos correctores de errores en tiempo real empleados en la reproducción de un CD se basan en espacios vectoriales y polinomios sobre $\mathbb{Z}_p$; la criptografía mayoritariamente empleada en las comunicaciones seguras por la red requiere propiedades de los números primos y otra cada vez más pujante propiedades de las curvas elípticas; en nuestro videojuego favorito lo que llega a nuestra pantalla proviene de unas transformaciones lineales que indican la posición de la cámara y el resto de la geometría de la escena; el método por el que las fotos extraídas de nuestra máquina fotográfica ocupan tan poco en el ordenador en relación con su detalle depende del análisis de Fourier... ¿No deberíamos saber más Matemáticas? Bajemos de las nubes. No podemos sugerir a los pacientes que estudien física cuántica porque las resonancias magnéticas se basen en el espín, ni tampoco debemos esperar una correlación entre la audiencia y las solicitudes de libros de electromagnetismo porque haya unas ecuaciones en derivadas parciales creadas por Maxwell y otros que permitan que se vea la televisión. Con esto llegamos a la pregunta clave: "¿Y a mí que me importa?". Si no eres ingeniero, físico, informático o ciudadano curioso probablemente nada. Sin embargo como estudiante de informática al escribir programas mínimamente complejos te encontrarás con problemas que tras un proceso más o menos arduo de traducción son ejercicios de las Matemáticas que has cursado y dominar esas asignaturas que te pueden parecer accesorias te dará un poder extraordinario. Además siempre es una inversión segura porque quizá dentro de unos años haya nuevos sistemas operativos, se cambien los protocolos de comunicación, se hable de la

Web n.0, haya formatos de audio y vídeo totalmente diferentes y tengas que estudiar muchas cosas de nuevo relegando a la inutilidad parte de los temarios de algunas asignaturas pero, no te preocupes, el teorema de Pitágoras lleva muchísimos siglos en los libros y sigue siendo extremadamente útil.

Trataremos de mostrar en las páginas siguientes algunas aplicaciones del álgebra lineal, posiblemente sesgadas, incompletas o inexactas por provenir de un matemático no aplicado en el sentido que se prefiera, esperando que sean suficientes para justificar las líneas anteriores. Debido al público al que está dirigido el nivel matemático está rebajado en perjuicio de los detalles, se incluye bibliografía para el que quiera conocerlos.

Fernando Chamizo
Departamento de Matemáticas
Universidad Autónoma de Madrid

Febrero 2008

Finalmente estos apuntes no suscitaron interés el curso 2007-2008. Nuestro propósito es retomar su redacción en éste.

Fernando Chamizo
Departamento de Matemáticas
Universidad Autónoma de Madrid

Febrero 2009

Índice general

Parte I

Aplicaciones

Aplicación 1

Valorando páginas

La popularización de la red ha estado ligada al propio desarrollo de la empresa Google y de su producto estrella, el motor de búsqueda conocido bajo el mismo nombre. Hay una gran diferencia para empresas y particulares entre que su página esté en los primeros lugares al buscar un término común o que aparezca en la plétora de páginas que nadie mira. Probablemente el funcionamiento preciso del motor de búsqueda o incluso el detalle de cómo ordena las páginas es un secreto celosamente guardado, pero el algoritmo básico usado en Google Directory que asigna la importancia de cada página para su ordenación en el proceso de búsqueda es público y constituye una idea sencilla, desafortunadamente patentada, para cualquiera que sepa álgebra lineal. Según Google este algoritmo es el corazón de su motor de búsqueda.

1.1. El modelo básico

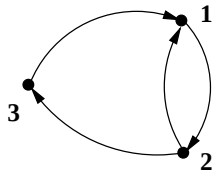
El objetivo es asignar a cada página un número positivo, una puntuación, que indica su importancia. Al igual que se puede puntuar sobre 10, sobre 30 o sobre 100, estos números sólo tienen valor relativo: una página con un 2,4 es el doble de importante que otra puntuada con un 1,2. Saber esto es suficiente para ordenarlas aunque no especifiquemos dónde está la Matrícula de Honor. Con una visión posiblemente muy simplista, Google tiene gigantescos índices de páginas y cuando buscamos un término los consulta resultando una infinidad de ellas. Para que el buscador sirva de algo hay que establecer una jerarquía de páginas y mostrar en primer lugar las importantes. Es ahí donde radica la potencia del buscador, si al introducir un término nos

diera una selección al azar de las páginas que lo contienen, sería bastante inútil. La puntuación de cada página se debe guardar en una gran base de datos que hay que consultar en las búsquedas y actualizar de vez en cuando.

Una página con muchos enlaces apuntando a ella seguramente es muy importante pero esta idea necesita precisarse si queremos tener relaciones cuantitativas. Por un lado no es lo mismo que haya un enlace a tu página desde la de Microsoft que desde la de tu primo (si no se apellida Gates) así que la importancia debe transmitirse por los enlaces. Por otra parte si en una página importante, por ejemplo un portal muy visitado, hay enlaces a mil páginas de usuarios, no es justo que éstas hereden toda la importancia del portal, a fin de cuentas como cada página es una entre mil, sólo debiera heredar la milésima parte de la importancia del portal.

Matemáticamente formulamos el modelo diciendo que la importancia de cada página es la suma ponderada de las importancias de las páginas que apuntan a ella, donde *ponderada* significa que debemos dividir estas importancias entre el número de enlaces salientes. El modelo refleja lo que ocurriría si uno navegase siguiendo enlaces al azar (en Matemáticas a esto se le llama una *cadena de Markov*), la importancia convenientemente normalizada sería el porcentaje de visitas.

Veamos un ejemplo muy simplificado de una red con sólo tres páginas que aclare el torrente de palabras anterior. Indicaremos las páginas con puntos y los enlaces con flechas (en Matemática Discreta de 2º se dirá que esto es un *grafo dirigido*).



x_1 = importancia de la página **1**
x_2 = importancia de la página **2**
x_3 = importancia de la página **3**

Según nuestro modelo $x_1 = x_2/2 + x_3$, la importancia de x_2 cuenta la mitad porque se reparte entre dos páginas; $x_2 = x_1$ porque desde **1** sólo se puede llegar a **2** y es la única forma de hacerlo, si nos obligan a navegar siempre que visitemos **1** también visitaremos **2**; finalmente $x_3 = x_2/2$ porque a **3** sólo llega media importancia de x_2, un enlace de dos.

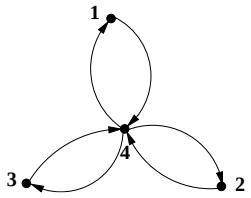
Si pensamos un poco antes de hacer nada, sólo mirando al esquema nos daremos cuenta de que x_3 es menos importante que las otras. Ahora con lápiz

y papel resolvamos el sistema, obteniendo

$$\left.\begin{array}{l} x_1 = x_2/2 + x_3 \\ x_2 = x_1 \\ x_3 = x_2/2 \end{array}\right\} \quad \Longrightarrow \quad \begin{array}{l} x_1 = 2t \\ x_2 = 2t \\ x_3 = t \end{array}$$

¿Es normal que salgan infinitas soluciones? Sí, porque ya hemos explicado que la importancia sólo tiene un valor relativo. Podemos decir que las importancias son $(x_1, x_2, x_3) = (2, 2, 1)$ o $(1, 1, 0{,}5)$ o $(20, 20, 10)$ dependiendo de sobre cuánto puntuemos y eso es irrelevante para la ordenación.

Otro ejemplo podría ser una red central donde todos los nodos necesitan pasar por uno especial para comunicarse. Intuitivamente ese nodo es tan importante como todos los otros juntos porque sin él no habría ninguna comunicación. El esquema, el sistema y su solución cuando hay 3 nodos satélites son los siguientes:



$$\begin{array}{l} x_1 = x_4/3 \\ x_2 = x_4/3 \\ x_3 = x_4/3 \\ x_4 = x_1 + x_2 + x_3 \end{array} \quad \Longrightarrow \quad \begin{array}{l} x_1 = x_2 = x_3 = t \\ x_4 = 3t \end{array}$$

Según lo que cabía esperar cada nodo satélite importa la tercera parte que el nodo central.

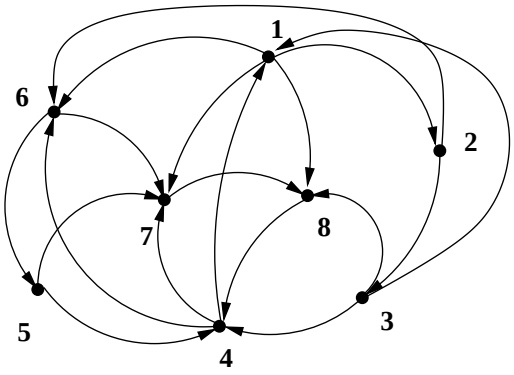
Ya que sabemos el truco, ejerzamos de científicos (o de SMSianos) empleando un poco de lenguaje sintético. Un momento de reflexión nos permite deducir que los sistemas que tenemos que resolver son de la forma $\vec{x} = A\vec{x}$ donde $A \in \mathcal{M}_{N\times N}(\mathbb{R})$ es la matriz que tiene como columna j el vector de probabilidades de ir a los otros nodos: sus coordenadas son 0 ó 1/nº enlaces desde j. El modelo nos sugiere que hay infinitas soluciones, entonces supongamos que nos podemos permitir $x_N = 1$. El nuevo sistema será $\vec{y} = B\vec{y} + \vec{v}$ donde B es el primer bloque $N-1 \times N-1$ de A y $\vec{v}$ son las $N-1$ primeras coordenadas de la última columna. El sistema de ecuaciones lineales a resolver es por tanto $(I - B)\vec{y} = \vec{v}$.

Como aspirantes a informáticos deberíamos ser capaces de escribir un programa que haga el trabajo por nosotros. Afortunadamente no tendremos que teclear y depurar líneas de código en C o en Java, o buscar bibliotecas,

si disponemos de alguno de los paquetes matemáticos que resuelven sistemas lineales, como Maple o Matlab, que están en algunas aulas de informática. Los *linuxeros* tacaños podemos emplear para esto Octave como clon de Matlab. El siguiente código funciona en ambos, basta escribirlo en un fichero de texto `fichero.m` y después teclear en la consola de Octave o Matlab `fichero` para tener la solución del último ejemplo:

```
% Matriz de probabilidades
A=      [0,0,0,1/3
         0,0,0,1/3
         0,0,0,1/3
         1,1,1,0];
N=length(A); % Tamaño de la matriz
B=A(1:N-1,1:N-1);  % Se queda con la submatriz N-1xN-1
v=A(1:N-1,N:N); % Toma la última columna como vector
x=[ (eye(N-1)-B)\v;1] % Resuelve el sistema (I-B)x=v y añade x_N=1
```

Modificando la definición de la matriz A se puede jugar con redes más complejas. Por ejemplo, para la maraña



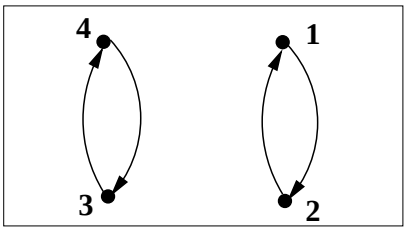
la matriz A y la solución son

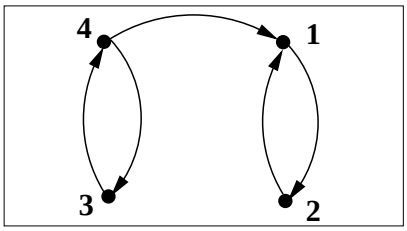
$$A = \begin{pmatrix} 0 & 0 & 1/3 & 1/3 & 0 & 0 & 0 & 0 \\ 1/4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 1/2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 0 \\ 1/4 & 1/2 & 0 & 1/3 & 0 & 0 & 0 & 0 \\ 1/4 & 0 & 0 & 1/3 & 1/2 & 1/2 & 0 & 0 \\ 1/4 & 0 & 1/3 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad \text{y} \quad \vec{x} = \begin{pmatrix} 0{,}400000 \\ 0{,}100000 \\ 0{,}050000 \\ 1{,}150000 \\ 0{,}266667 \\ 0{,}533333 \\ 0{,}883333 \\ 1{,}000000 \end{pmatrix}.$$

Es decir, que de acuerdo con su importancia las páginas vienen ordenadas como $\mathbf{3} < \mathbf{2} < \mathbf{5} < \mathbf{1} < \mathbf{6} < \mathbf{7} < \mathbf{8} < \mathbf{4}$. Nótese que no es la que recibe más enlaces. Por otro lado, la página **1** no está en el último puesto porque recibe un poderoso enlace de la **4**, la página más importante.

1.2. Limitaciones teóricas y prácticas

Hemos hecho unos ejemplos con unos pocos nodos y un programilla. ¿Se puede escalar y generalizar el procedimiento o es una simplificación ilusoria? Analicemos las dificultades teóricas y prácticas que presenta el modelo. Respecto a las primeras hay configuraciones especiales problemáticas. Por ejemplo las dos siguientes:





Si escribimos las matrices de estas dos configuraciones y las introducimos en nuestro programa Octave o Matlab nos dirán nanay, eso sí con refinadas formas inglesas e incluso después del aviso nos pueden dar un sustituto de solución por si queremos arriesgarnos. ¿Qué sucede? En el primer caso el espacio de soluciones depende de dos parámetros $x_1 = x_2 = t$, $x_3 = x_4 = u$; esto es lógico porque los bloques **1**-**2** y **3**-**4** no tienen nada en común, no se pueden comparar, no hay criterio común que podamos aplicar para juzgarlos relativamente. El segundo caso es más sutil: al conectar los dos bloques la mitad de la importancia de **4** se fuga al otro bloque y **3** recibe la otra mitad, pero por otra parte toda la importancia de **3** va a **4**, esto lleva a una contradicción excepto si **3** y **4** tienen importancia cero, lo cual parece extraño y además contradice la suposición $x_4 = 1$ que empleamos en el programa.

Inferimos que habrá problemas siempre que dadas dos páginas no podamos ir de una de ellas a la otra navegando por la red (se dice que el grafo dirigido no es *fuertemente conexo*). En el primer diagrama era imposible ir de **4** a **1** perdiendo con ello la valoración relativa mientras que en el segundo era imposible ir de **1** a **4** lo que implicaba un déficit de importancia en el bloque **3**-**4** que llevaba a cero. Si nos aferramos al modelo, ninguna de estas configuraciones es intrínsecamente conflictiva, únicamente choca con las hipótesis que hemos hecho implícitamente al escribir nuestro programa. A saber, que el primer bloque $N-1 \times N-1$ de $I - A$ es invertible.

Ocupémonos ahora de las limitaciones puramente prácticas. Según los libros de cálculo numérico o un simple análisis cuidadoso, aplicar eliminación de Gauss para N variables requiere del orden de $2N^3/3$ operaciones. Si consideramos que hay $8 \cdot 10^9$ páginas web (se ha dicho que éstas eran las visibles por Google en 2004, según sus propios informes ya se han triplicado) y trabajamos en el PC de casa que en condiciones ideales es capaz de hacer $3 \cdot 10^9$ operaciones por segundo (el sistema operativo y el compilador o el intérprete ya se encargan de reducir este número), la solución tardará en aparecer $1{,}14 \cdot 10^{20}$ segundos, que son unos cuantos billones de años, más que la edad estimada del universo. A modo de ilustración del alcance de nuestro programa, cambiemos la primera línea por `N=20000; A=rand(N);` que genera una matriz aleatoria $N \times N$ o quizá tomemos N un poco mayor dependiendo de la capacidad de nuestro ordenador. De nuevo nos dirá que nones. Todavía peor, para N del orden de unos miles el ordenador empezará a calcular y el programa acabará en un error fatal con sus capacidades excedidas. Las dificultades prácticas se tornan entonces muy serias si lo más que podemos diseñar es un motor de búsqueda para una red con unos pocos miles de páginas. Eso se quedaría muy corto incluso sólo para buscar en el sitio web de la UAM.

1.3. Un algoritmo iterativo

Si con todas estas dificultades Google funciona, ¿dónde está el truco? Consideremos el primero de los ejemplos estudiados, la red de tres páginas cuya matriz A tiene $a_{12} = a_{32} = 1/2$, $a_{21} = a_{13} = 1$ y el resto de las componentes nulas. Elijamos al azar un vector de importancia $\vec{x}$, para ser originales digamos $x_1 = \pi$, $x_2 = e$, $x_1 = \sqrt{10}$. Si calculamos $\vec{x}_1 = A\vec{x}$, $\vec{x}_2 = A\vec{x}_1$, $\vec{x}_3 = A\vec{x}_2$,... se obtiene

$$\vec{x}_1 = \begin{pmatrix} 4{,}2994 \\ 3{,}1416 \\ 1{,}5811 \end{pmatrix}, \ \vec{x}_2 = \begin{pmatrix} 3{,}1519 \\ 4{,}2994 \\ 1{,}5708 \end{pmatrix}, \ \vec{x}_3 = \begin{pmatrix} 3{,}7205 \\ 3{,}1519 \\ 2{,}1497 \end{pmatrix}, \ \ldots \ \vec{x}_{15} = \begin{pmatrix} 3{,}6071 \\ 3{,}6160 \\ 1{,}7990 \end{pmatrix}$$

y esto se acerca mucho a un vector de importancia válido que resuelve el sistema $\vec{x} = A\vec{x}$, dividiendo entre la última coordenada obtenemos algo muy parecido a $(2, 2, 1)$. ¿Es una casualidad? Procedemos de la misma forma con la matriz de la maraña y para no cansarnos calculando escribimos unas líneas con un pequeño bucle partiendo por ejemplo del vector con todo unos. Ya

puestos con las manos al teclado podemos normalizar el resultado para que la última coordenada sea uno y comparar con el resultado exacto.

```
% Matriz de probabilidades
A=[
0/4,0/2,1/3,1/3,0/2,0/2,0/1,0/1; 1/4,0/2,0/3,0/3,0/2,0/2,0/1,0/1;
0/4,1/2,0/3,0/3,0/2,0/2,0/1,0/1; 0/4,0/2,1/3,0/3,1/2,0/2,0/1,1/1;
0/4,0/2,0/3,0/3,0/2,1/2,0/1,0/1; 1/4,1/2,0/3,1/3,0/2,0/2,0/1,0/1;
1/4,0/2,0/3,1/3,1/2,1/2,0/1,0/1; 1/4,0/2,1/3,0/3,0/2,0/2,1/1,0/1;
];
niter=15; %número de iteraciones
x=ones(length(A),1); %todo unos
for n=1:niter %Bucle
        x=A*x;
endfor
x/x(length(A)) %divide entre la última coordenada
```

Con las 15 iteraciones elegidas la máxima diferencia de las coordenadas del valor exacto es 0,000615, un número despreciable porque no cambia la ordenación relativa.

No todo son noticias tan buenas. Si partimos por ejemplo de $x_1 = 4$, $x_2 = 3$, $x_3 = 2$, $x_4 = 1$ en las dos configuraciones especiales antes señaladas el resultado es oscilante y no lleva a soluciones:

$$\vec{x}_{\text{par}} = \begin{pmatrix} 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}, \ \vec{x}_{\text{impar}} = \begin{pmatrix} 3 \\ 4 \\ 1 \\ 2 \end{pmatrix} \qquad \text{y} \qquad \vec{x}_{\text{par}} \approx \begin{pmatrix} 6 \\ 4 \\ 0 \\ 0 \end{pmatrix}, \ \vec{x}_{\text{impar}} \approx \begin{pmatrix} 4 \\ 6 \\ 0 \\ 0 \end{pmatrix}.$$

Se pueden buscar otros ejemplos raros oscilantes con periodos más grandes.

Parece entonces que excepto en casos patológicos la sucesión $\vec{x}_n = A\vec{x}_{n-1}$ *converge* a una solución de $\vec{x} = A\vec{x}$. La palabra mágica suena a análisis y el profesor de esa asignatura te dirá que si supones que la sucesión converge a algo, digamos $\vec{l} = \lim \vec{x}_n$ entonces tomando límites obtendrás $\vec{l} = A\vec{l}$, es decir, que ese algo es una solución.

Usando la teoría de autovalores y autovectores que veremos en este curso, se infiere que en cierto sentido la "situación habitual" debería ser la convergencia. Experimentando un poco se observa que para redes suficientemente interconectadas éste es de hecho el caso.

Revisemos ahora nuestras estimaciones del número de operaciones. Si hay $8 \cdot 10^9$ páginas y cada una tiene en promedio, digamos, 10 enlaces $A\vec{x}$ requeriría $8 \cdot 10^{10}$ operaciones que es el número de componentes no nulas de A. Si hacemos 80 iteraciones (según los informes Google hace entre 50 y 100)

tenemos $6{,}4 \cdot 10^{12}$ operaciones que en el ordenador de nuestra casa requieren en condiciones óptimas menos de cuarenta minutos. Además el proceso es fácilmente susceptible de computación distribuida y con 3000 ordenadores (y Google tiene más) reduciríamos en principio el tiempo a una cantidad menor que un segundo. Por supuesto no hay que tomar en serio estos números porque el tiempo de acceso a bases de datos y otras cosas son la parte del león que estamos despreciando, se pretende mostrar únicamente que esta parte del proceso ya no es una barrera infranqueable. Según los informes, Google tarda algunos días en hacer el cálculo y se dice que recalcula el vector de importancias una vez al mes.

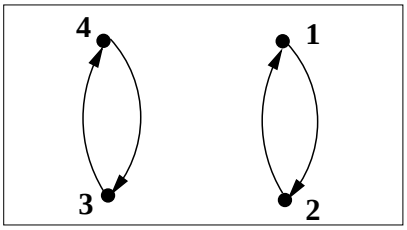
1.4. Un modelo revisado

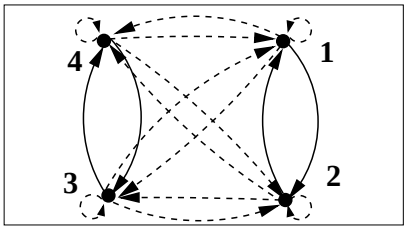
Tenemos un modelo matemático razonable y un algoritmo pero desafortunadamente no podemos afirmar con seguridad que funciones correctamente para el mapa de todas las webs y la eficiencia del algoritmo está extrapolada de lo que ocurre con algunos ejemplos demasiado simples. Podríamos cerrar los ojos y aplicarlo igualmente o utilizar algún truco como promediar los resultados para eliminar oscilaciones (nótese que en los dos contraejemplos los promedios dan soluciones). Google no toma ese riesgo y modifica levemente el modelo haciéndolo un poco menos realista para estar en situación de asegurarse el éxito gracias a un resultado matemático y de paso tener cierto control sobre la eficiencia.

El resultado matemático es un teorema de O. Perron y data de 1907 (quizá entonces le dijeron ¿y eso para qué sirve?). Traducido a nuestro lenguaje implica como caso particular que si A es una matriz cuyas columnas son vectores con coordenadas estrictamente positivas y de suma uno, entonces $\vec{x}_n = A\vec{x}_{n-1}$ converge para cualquier vector inicial (positivo). Ya hemos señalado que la convergencia implica que la sucesión tiende a la solución buscada de $\vec{x} = A\vec{x}$.

A primera vista el teorema parece poco aplicable porque en nuestro caso la matriz A está típicamente llena de ceros y eso era fundamental para que los cálculos fueran posibles en un tiempo razonable. La solución pasa por un cambio del modelo suponiendo que cierto porcentaje pequeño de veces nos cansamos y en vez de seguir la línea de enlaces saltamos a una página al azar (que puede ser incluso la propia de partida). Esto no suena mal aunque en rigor es poco realista que saltemos a páginas desconocidas. Tomemos por

ejemplo la primera configuración conflictiva. Con el modelo modificado es como si hubieran aparecido enlaces débiles nuevos.





Si suponemos arbitrariamente que la relación entre las probabilidades de un enlace débil y de uno normal es del 10 % al 90 %, la nueva matriz será el 90 % de la anterior y el 10 % restante de probabilidad se reparte entre los 4 enlaces débiles que salen de cada página, uno por página (algunos se superponen con los existentes). El cambio de matriz es entonces

$$A = \begin{pmatrix} 0&1&0&0 \\ 1&0&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{pmatrix} \quad \longrightarrow \quad \widetilde{A} = \begin{pmatrix} 0{,}1/4 & 0{,}9{+}0{,}1/4 & 0{,}1/4 & 0{,}1/4 \\ 0{,}9{+}0{,}1/4 & 0{,}1/4 & 0{,}1/4 & 0{,}1/4 \\ 0{,}1/4 & 0{,}1/4 & 0{,}1/4 & 0{,}9{+}0{,}1/4 \\ 0{,}1/4 & 0{,}1/4 & 0{,}9{+}0{,}1/4 & 0{,}1/4 \end{pmatrix}.$$

El método iterativo $\vec{x}_n = A\vec{x}_{n-1}$ ahora converge sin problemas a una solución de $\vec{x} = \widetilde{A}\vec{x}$ que es el objetivo del nuevo modelo. Como A y $\widetilde{A}$ son parecidas es de esperar que las soluciones de ambos modelos se parezcan.

Escribámoslo en general. Si U es una matriz $N \times N$ con $u_{ij} = 1$ lo que hemos hecho es aplicar la fórmula

$$\widetilde{A} = cA + \frac{1-c}{N}U \qquad \text{con } 0 < c < 1,$$

donde en el caso anterior $c = 0{,}9$. La relación de recurrencia $\vec{x}_n = A\vec{x}_{n-1}$ se traduce en $\vec{x}_n = cA\vec{x}_{n-1} + (1-c)s_{n-1}\vec{u}/N$ donde s_{n-1} es la suma de las coordenadas de $\vec{x}_{n-1}$ y $\vec{u}$ es el vector con todas sus componentes 1. Es un ejercicio comprobar que dicha suma es independiente de n. Entonces si s_0 es la suma de las componentes del vector inicial, el proceso iterativo con el nuevo modelo es

$$\vec{x}_n = cA\vec{x}_{n-1} + \frac{1-c}{N}s_0\vec{u}.$$

Con esta formulación el número de operaciones es prácticamente igual al del primer modelo.

El único cabo suelto es qué valor elegir para c. Se puede probar matemáticamente que la rapidez de convergencia se acelera si c se aleja de uno pero

por otra parte el nuevo modelo se parecerá más al original cuanto más cercano esté c a uno. Como solución de compromiso los creadores de Google tomaron $c = 0{,}85$. Si algún día decidieran tomar por ejemplo $c = 0{,}92$ para hacer el modelo más realista, argumentos con autovalores y autovectores sugieren que el número de iteraciones debería duplicarse para conseguir la misma precisión.

Respecto al vector inicial, sin información adicional lo más razonable es tomar un múltiplo de $\vec{u}$. Probablemente en la práctica se emplea el resultado de la anterior actualización porque lo más posible es que las importancias de la mayoría de las páginas no sufran cambios repentinos.

Modificando los aledaños del bucle del último programa a:

```
c=0.85;
for n=1:niter %Bucle
        x=c*A*x+(1-c);
endfor
```

se puede utilizar para el segundo modelo, obteniéndose un vector de importancias (con `niter` $=$ `15`) cuyas coordenadas son $x_1 = 0{,}45779$, $x_2 = 0{,}18673$, $x_3 = 0{,}16880$, $x_4 = 1{,}13119$, $x_5 = 0{,}33876$, $x_6 = 0{,}58660$, $x_7 = 0{,}90052$. $x_8 = 1{,}00000$, lo cual no está muy lejos de la solución hallada para el primer modelo.

Para mostrar fehacientemente que todo esto no son entelequias teóricas, incluimos un código en C que permite llevar a cabo el algoritmo en una red creada aleatoriamente con `N` $=$ `10`$^{\texttt{6}}$ páginas y `E` $=$ `10` enlaces salientes de cada una de ellas. Un buen reto para el lector es crear un programa eficiente que permita tratar el caso en el que número de enlaces no esté fijado de antemano.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 1000000
#define E 10
#define ITER 80
#define C 0.85

void redalea(unsigned int *ind){
        int i=N*E;
        while(i--)
                *(ind++)=(rand()%N);
}
```

```
void inicia(double *x,double r){
        int i=N;
        while(i--)
                *(x++)=r;
}

int main(){
        unsigned int ind[N*E]; /* matriz de índices */
        double x[N]; /* vector de importancia */
        double xn[N]; /* nuevo vector de importancia */
        int i,j,k; /* auxiliares */
        unsigned int t=time(NULL); /* tiempo */

        srand(t);

        redalea(ind); /* Crea red aleatoria */
        inicia(x,1.0/N); /* x=1/N */

        for(k=0; k<ITER; ++k){ /* Bucle principal */
                printf("iteraciones=%d, tiempo=%d\n",k, time(NULL)-t);
                inicia(xn,(1.0-C)/N); /* xn=(1-C)/N */
                for(i=0;i<N;++i) /* bucle cálculo de matriz por x */
                        for(j=0;j<E;++j)
                                xn[ind[i*E+j]]+=C*x[i]/E;
                for(i=0;i<N;++i) /* copia el resultado a x */
                        x[i]=xn[i];
        }

        return 0;
}
```

Nótese que no se almacena la matriz A sino la lista de las posiciones de los elementos no nulos. Probando el programa en un ordenador normal y corriente, tardó 70 segundos en realizar las 80 iteraciones para conseguir una aproximación del millón de coordenadas del vector de importancias.

NOTA: Para saber más se recomienda especialmente la lectura del artículo [Fe] que obtuvo un premio de divulgación de la Real Sociedad Matemática Española. [Br-Pa] es el artículo original de los creadores de Google.

1.5. Bibliografía

[Fe] P. Fernández Gallardo. *El secreto de Google y el Álgebra lineal.* Boletín de la Sociedad Española de Matemática Aplicada **30** (2004), 115-141. Disponible en `www.uam.es/pablo.fernandez` donde también hay una versión en inglés más concisa.

[Au] D. Austin. *How Google Finds Your Needle in the Web's Haystack*. Feature Column Archive, December 2006. American Mathematical Society. En `www.ams.org/featurecolumn/archive/pagerank.html`

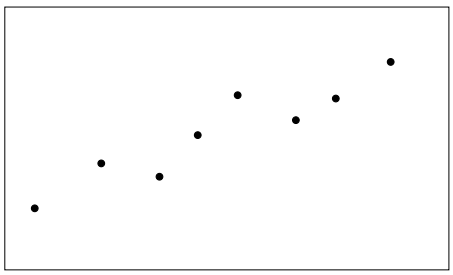
[Br-Le] K. Bryan, T. Leise. *The* $\$25,000,000,000$ *eigenvector: the linear algebra behind Google*. SIAM Rev. **48** (2006), no. 3, 569–581. Disponible en `www.rose-hulman.edu/$\sim$bryan/google.html`

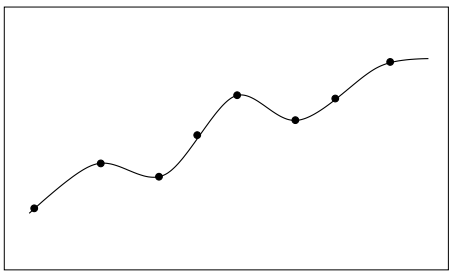
[Br-Pa] S. Brin, L. Page. *The anatomy of a large-scale hypertextual Web search engine*. Computer Networks and ISDN Systems **33**, 107-17, 1998. Disponible en `infolab.stanford.edu/pub/papers/google.pdf`

Aplicación 2

Une los puntos

Muchas veces uno se ve forzado a inventar datos de los que carece y no necesariamente para engañar. Por ejemplo, la inflación, las ganancias de una empresa, la altura de nuestro sobrino, las representamos como una línea continua, cuando es obvio que hemos tomado una cantidad discreta y pequeña de datos. El dibujo de unos pocos puntos sería menos interpretable para nosotros pues no daría una idea tan clara de la evolución.





Otro ejemplo es que si en un programita gráfico queremos que nuestro monigote pase por los puntos A, B, C, D y E, para crear automáticamente una animación suave tendrá que inventar todos los fotogramas intermedios. Una variación de ello son los programas de diseño gráfico. Los más sencillos editores vectoriales (como Xfig) tienen opciones para crear curvas ajustándolas a puntos fijos o modificables estirando unos puntos de control, otros más complicados de modelado tridimensional (como Blender) permiten trabajar con superficies dando impresionantes resultados. Incluso algo tan básico como ampliar una imagen de tamaño $x \times y$ a tamaño $x' \times y'$ requiere inventar *pixels* que no tenemos: en otro caso quedarían sustituidos por unos feos agujeros.

En una herramienta como GIMP al reescalar se admiten varias opciones de calidad con nombres tan misteriosos como interpolación lineal o interpolación cúbica. Todo ello cobrará sentido en este capítulo y en el próximo.

2.1. Unión exacta

Básicamente en las diferentes situaciones hay dos posibilidades: buscar una curva suave que pase exactamente por los puntos seleccionados o buscar una que sólo pase cerca. La segunda posibilidad parece imperfecta pero posiblemente es la que más veces se emplea en las aplicaciones. En este capítulo nos ocuparemos de la primera describiendo un par de métodos (el segundo es el bueno) sin entretenernos en optimizar los algoritmos ni en profundizar en propiedades teóricas interesantes (véase [Ki-Ch] y [St-Bu]).

Dados $n+1$ puntos $P_i = (x_i, y_i)$, $0 \leq i \leq n$, el problema general que nos planteamos es encontrar una curva $\sigma : [0,1] \longrightarrow \mathbb{R}^2$, $\sigma(t) = (x(t), y(t))$ que pase por el punto P_i cuando $t = t_i$. Para simplificar supondremos que los t_i, llamados *nodos* son $t_i = i/n$, esto se lo que se llama *interpolación uniforme*. En la práctica a veces es conveniente romper esta hipótesis y trabajar con una distribución de nodos no uniforme a cambio de admitir dificultades en la programación. Por ejemplo, si $n = 4$ y P_2 y P_3 están muy cercanos parece que dedicar $1/4$ del tiempo al arco $\widehat{P_2P_3}$ es un derroche y sería ridículo usar $\sigma(t)$ como trayectoria de una animación.
Sin embargo si quisiéramos hacerlo "bien" tendríamos que entrar en consideraciones acerca de la longitud de la curva, o examinar si en el punto hay mucha curvatura y quizá decidir eliminarlo. ¿Cómo introducir estos conceptos matemáticos de longitud y curvatura en un programa? ¿Cómo se generalizarían estas ideas al caso tridimensional (tan importante por las películas de animación y los videojuegos)? Las respuestas no son obvias [Bus], [Ha-St].

2.2. Interpolación de Lagrange

El primer método (poco usado para un número grande de nodos) busca $x(t)$ e $y(t)$ dentro de $\mathbb{R}_n[t]$, el espacio vectorial de los polinomios reales de grado menor o igual que n. Consideremos los *polinomios de Lagrange* de este

espacio asociados a los nodos definidos como

$$L_j(t) = \prod_{k \neq j} \frac{t - t_k}{t_j - t_k}$$

para $0 \leq j \leq n$. Lo que tienen de particular es la sencilla propiedad

$$L_j(t_i) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases} \tag{2.1}$$

que prueba casi inmediatamente que $\mathcal{B} = \{L_0(t), \ldots, L_n(t)\}$ es base de $\mathbb{R}_n[t]$ ya que $\lambda_0 L_0(t) + \cdots + \lambda_n L_n(t) = 0$ conduce a $\lambda_i = 0$ al sustituir $t = t_i$, con lo cual tenemos $n + 1$ vectores linealmente independientes en un espacio de dimensión $n + 1$.

La propiedad (2.1) asegura que la coordenada i-ésima de un polinomio de $\mathbb{R}_n[t]$ en la base $\mathcal{B}$ es justamente su valor en $t = t_i$. Releyendo esto con cuidado el número de veces que sea necesario, se tiene que la solución a nuestro problema es

$$\begin{cases} x(t) = x_0 L_0(t) + x_1 L_1(t) + \cdots + x_n L_n(t) \\ y(t) = y_0 L_0(t) + y_1 L_1(t) + \cdots + y_n L_n(t) \end{cases}$$

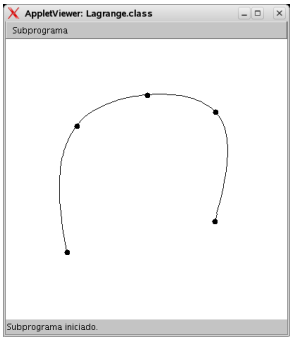
Para ver el resultado gráficamente creemos el (¿o la?) siguiente *applet* en Java[1] que realiza la interpolación de Lagrange de unos puntos que podemos mover con el ratón. La constante (¿por qué en Java no hay constantes?) que indica el número de puntos a interpolar es `N`.

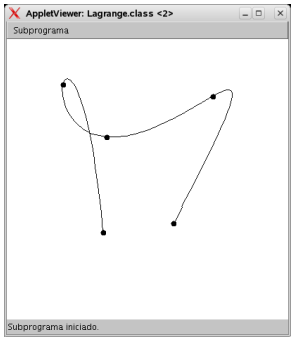
```
import java.awt.*;
import java.applet.Applet;

public class Lagrange extends Applet {
    Point[] puntos;    //Puntos de interpolación
    int pcurso;    // Número del punto en curso
    Image lienzo;
    Graphics grafico;
    // Constantes
    static final int N = 5; // Número de puntos
    static final int R = 8; // Radio de los puntos

    public void init() {
        puntos = new Point[N];
        pcurso = N; // Desactiva el punto en curso
        // Equidistribuye
        for(int i=1;i < N+1;++i)
            puntos[i-1] = new Point(i*(int)(size().width/(double)(N+1)),
```

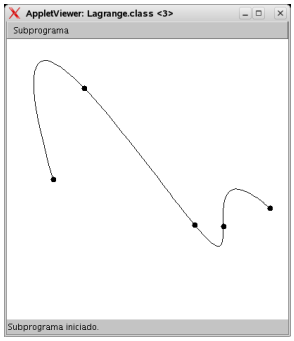
[1]Los *fans* de Java sabrán que algunas de las funciones empleadas están anticuadas o mal vistas (*deprecated*) pero ellos las corregirán fácilmente, y si se dan prisa antes de que las modernas se queden también anticuadas.

```
                        (int)(size().height/2));
        // Crea imagen ajustada al tamaño especificado en  <APPLET>
        lienzo = createImage(size().width,size().height);
        grafico = lienzo.getGraphics();
    }

    // Calcula el valor del polinomio de Lagrange j-ésimo
    private double lagr(int i, double t){
        double pr=1.0;
        for(int j=0; j < N; ++j)
            if(j != i) pr*=(t*(N-1.0)-j)/(i-j);
        return pr;
    }

    public void paint(Graphics g) {
        // Pinta el segmento (px,py), (qx,qy)
        double qx=puntos[0].x, qy=puntos[0].y, px, py;

        // Rellena de blanco
        grafico.setColor(Color.white);
        grafico.fillRect(0,0,size().width,size().height);
        grafico.setColor(Color.black);

        // Dibuja los puntos
        for(int i=0;i < N;++i)
            grafico.fillOval(puntos[i].x-R/2,puntos[i].y-R/2,R,R);

        // Los interpola
        for(double t=0.0; t<=1.0; t+=0.01){
            // nuevo-> antiguo
            px=qx; py=qy; qx=0.0; qy=0.0;
            // Calcula nuevo
            for(int i=0; i<N; ++i){
                qx+=(double)(puntos[i].x)*lagr(i,t);
                qy+=(double)(puntos[i].y)*lagr(i,t);
            }
            grafico.drawLine((int)(px-.5),(int)(py-.5),(int)(qx-.5),(int)(qy-.5));
        }
        grafico.drawLine((int)qx,(int)qy,puntos[N-1].x,puntos[N-1].y);
        g.drawImage(lienzo,0,0,this); // Al lienzo
    }

    public void update(Graphics g) {
        paint(g);
    }



    public boolean mouseDown(Event evt, int x, int y) {
        Point p = new Point(x,y);
        for(int i=0;i < N;++i)
            if( Math.abs(p.x-puntos[i].x) + Math.abs(p.y-puntos[i].y) < R )
                pcurso=i; // punto i en curso
        return true;
    }

    public boolean mouseDrag(Event evt, int x, int y) {
        if(pcurso < N) { // Si hay un punto activado
            puntos[pcurso].move(x,y);
            repaint(); // actualiza
        }
        return true;
    }

    public boolean mouseUp(Event evt, int x, int y) {
        pcurso = N; // Suelta punto
        return true;
    }
}
```

Con el programa tal como esta, con `N = 5` las cosas van relativamente bien:

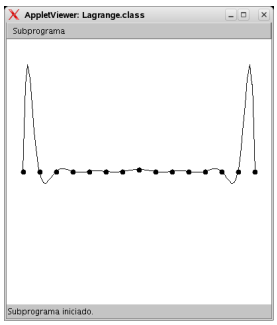


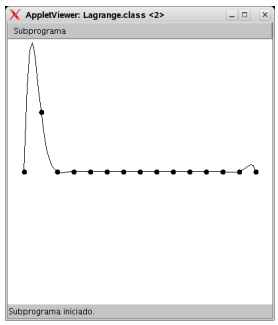




Puestos a quejarnos, en la última figura no hay razón aparente para continuar la primera ondulación tan arriba y las dos últimas ondulaciones parecen un poco superfluas.

Las deficiencias intolerables se muestran inequívocamente cuando el número de puntos crece. Por ejemplo si modificamos el programa anterior escribiendo `N = 15` al separar ligerísimamente el punto central de la altura inicial común a todos los puntos se obtiene la figura de la izquierda.





Menos drástica pero aun así insostenible es la situación representada en la figura derecha en la que una elevación del segundo punto de interpolación provoca una ondulación desmesurada al comienzo y otra menor inexplicable y superflua al final. Si a un niño le hubiéramos pedido que uniera los puntos con un trazo, no habría producido esas ondulaciones innecesarias tan acusadas.

Con mala idea y Matemáticas se pueden encontrar funciones normalitas tales que cuantos más puntos de su gráfica interpolemos con este método, peor sea la aproximación de la función. Esta situación problemática y extraña (¿más experimentos peores resultados?) se llama *fenómeno de Runge* (véase en [Ki-Ch] §6 lo sutil de la situación).

Una alternativa es agrupar los puntos de interpolación en grupos pequeños, digamos de tres en tres o de cuatro en cuatro y emplear en cada

trozo la interpolación de Lagrange, pero el resultado tendrá en general unas feas esquinas. El único caso empleado en la práctica es el que corresponde a tomar los nodos de dos en dos y por tanto polinomios lineales, llamado *interpolación lineal*. El resultado es un gráfico como el de la fiebre de los enfermos (al menos en los dibujos animados).

2.3. Splines cúbicos

Antes de la llegada de los programas de diseño gráfico, la palabra *spline* era la denominación en inglés de una plantilla ajustable para trazar curvas, sin embargo a día de hoy el significado predominante es el de un artificio matemático ampliamente usado en multitud de áreas de la ingeniería.

El objetivo es solventar la aparición de esquinas en la interpolación a trozos, donde cada trozo es un polinomio cúbico, para ello se pide que las funciones tengan derivadas continuas. Para ser más académico, lo que se exige es que $x(t)$ e $y(t)$ pertenezcan al espacio vectorial de *splines cúbicos* definido como

$$V = \left\{ f : [0,1] \longrightarrow \mathbb{R} \text{ tal que } f,\ f' \text{ son continuas y } f\big|_{[t_i,t_{i+1}]} \in \mathbb{R}_3[t] \right\}$$

donde $f\big|_I$ significa el trozo de f definido en el intervalo I.

Un teorema matemático asegura que entre todas las funciones de V usadas para interpolar, la que en algún sentido (que no indicamos) se curva menos es la única que cumple además que f'' es continua y que $f''(0) = f''(1) = 0$. En [Lo] hay un enunciado preciso del teorema y una demostración asequible en este curso. Lo que nosotros veremos aquí es que estas condiciones analíticas se traducen en un sistema lineal y una vez un problema ingenieril nos lleva la territorio del álgebra lineal.

Si f como antes interpola las coordenadas y_i, $0 \leq i \leq n$ de los puntos P_i, definamos los polinomios de $\mathbb{R}_3[t]$, $Q_i(t) = f\big|_{[t_i,t_{i+1}]}$, $0 \leq i < n$. Para dar un tratamiento uniforme a todos ellos, estiramos $[t_i, t_{i+1}]$ con un cambio lineal a $[0,1]$, matemáticamente consideramos $S_i(t) = Q_i\big((t+i)/n\big)$, de este modo $S_i(0) = Q_i(t_i) = y_i$ y $S_i(1) = Q_i(t_{i+1}) = y_{i+1}$. Digamos que

$$S_i(t) = s_{i1}t^3 + s_{i2}t^2 + s_{i3}t + s_{i4} \in \mathbb{R}_3[t]. \tag{2.2}$$

Nuestro objetivo es hallar los coeficientes s_{ij}. Para ello debemos utilizar las propiedades antes citadas de f. En primer lugar, como interpola los y_i, ya

hemos mencionado que $S_i(0) = y_i$ y $S_i(1) = y_{i+1}$, lo que se traduce en las ecuaciones:

$$s_{i4} = y_i \tag{2.3}$$

$$s_{i1} + s_{i2} + s_{i3} + s_{i4} = y_{i+1} \tag{2.4}$$

para $0 \leq i < n$. Por otra parte en cada uno de los nodos interiores, $0 < i < n$, se tiene que cumplir que las primeras y segundas derivadas son continuas y por tanto su cálculo por la derecha y por la izquierda debe coincidir. Es decir, $S'_{i-1}(1) = S'_i(0)$ y $S''_{i-1}(1) = S''_i(0)$ que en ecuaciones lineales es

$$3s_{i-1\,1} + 2s_{i-1\,2} + s_{i-1\,3} = s_{i3} \tag{2.5}$$

$$6s_{i-1\,1} + 2s_{i-1\,2} = 2s_{i2}. \tag{2.6}$$

Por último imponemos $S''_0(0) = S''_{n-1}(1) = 0$ que se escribe como

$$2s_{02} = 0 \tag{2.7}$$

$$6s_{n-1\,1} + 2s_{n-1\,2} = 0. \tag{2.8}$$

Tenemos que resolver el complejo sistema (2.3)–(2.8) donde los s_{ij} son las incógnitas y las y_i son los datos conocidos. Parece muy difícil pero no lo es tanto con una organización ingeniosa. Definamos $D_i = f'(t_i)$, entonces cualquiera de los dos miembros de (2.5) es D_i, en particular $s_{i3} = D_i$. Restando (2.4) y (2.3) se sigue $s_{i1} + s_{i2} + s_{i3} = y_{i+1} - y_i$, y por otra parte $D_{i+1} + D_i = (3s_{i1} + 2s_{i2} + s_{i3}) + s_{i3}$, por consiguiente $s_{i1} = D_{i+1} + D_i - 2(y_{i+1} - y_i)$. De forma similar se obtiene s_{i2}. En definitiva, si hallamos D_i tendremos resuelto el sistema con

$$\begin{cases} s_{i1} = D_{i+1} + D_i - 2(y_{i+1} - y_i), & s_{i3} = D_i \\ s_{i2} = -D_{i+1} - 2D_i + 3(y_{i+1} - y_i), & s_{i4} = y_i. \end{cases} \tag{2.9}$$

Nos falta una ecuación para despejar D_i y la única de las ecuaciones generales (2.3)–(2.6) que no hemos empleado es (2.6). Sustituyendo en ella (2.9) se sigue

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_i),$$

válido para $0 < i < n$, mientras que en los nodos extremos se ve modificada

ligeramente por (2.7) y (2.8). Lo que resulta es en notación matricial:

$$\begin{pmatrix} 2 & 1 & 0 & 0 & \vdots \\ 1 & 4 & 1 & 0 & \vdots \\ 0 & 1 & 4 & 1 & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \dots & 0 & 1 & 4 & 1 \\ \dots & \dots & 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_{n-1} \\ D_n \end{pmatrix} = 3 \begin{pmatrix} y_1 - y_0 \\ y_2 - y_0 \\ y_3 - y_1 \\ y_4 - y_2 \\ \vdots \\ y_n - y_{n-2} \\ y_n - y_{n-1} \end{pmatrix}.$$

Después a partir de los D_i se pueden construir los polinomios $S_i(t)$ sustituyendo en (2.9) y en la definición (2.2). En resumidas cuentas, aunque la deducción teórica ha sido un poco larga e incompleta, en la práctica la interpolación con splines cúbicos se reduce a resolver un sistema con una *matriz tridiagonal*, computacionalmente muy sencillo [St-Bu], y a hacer unas sustituciones.

Con el siguiente código Matlab u Octave[2] definimos una función que realiza el proceso y evalúa f en `param`. Hemos empleado prácticamente la misma notación que en el desarrollo anterior, con la excepción más notable de que la n está trasladada en 1 para evitar índices nulos (no permitidos en estos paquetes pero sí en C).

```
function resul = mifspline(param,y)
    n=length(y);

    % Crea los nodos
    t=linspace(0,1,n)';

    % Crea el sistema y lo resuelve
    incr=(y(2:end)-y(1:end-1));

    A=spdiags([ones(n,1),4*ones(n,1),ones(n,1)],-1:1,n,n);
    A(1,1)=2;
    A(n,n)=2;

    S=zeros(n,4);
    S(:,3)=A\(3*[y(2)-y(1);incr(2:end)+incr(1:end-1);y(n)-y(n-1)]);

    % Construye los coeficientes de los polinomios
    n=n-1;
```

[2]Si se dispone de una versión antigua de Octave que no manipule matrices dispersas, hay que decidirse entre actualizarla o sustituir instrucciones como `spdiags` por un código alternativo.

```
    nk=n;
    for m=1:n
      S(m,1)=-2*incr(m)+S(m,3)+S(m+1,3);
      S(m,2)=3*incr(m)-2*S(m,3)-S(m+1,3);
      S(m,4)=y(m);
    end

    S=S(1:n,1:4)*diag([nk*nk*nk,nk*nk,nk,1]);

    % Evalúa los polinomios
    resul=ppval(mkpp(t,S),param);
end
```

Las dos últimas instrucciones deshacen el cambio para pasar de S_i a Q_i y construyen la función f a partir de los coeficientes de los polinomios cúbicos.

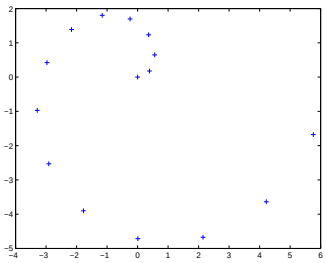
Por supuesto, para comprobar los resultados queremos ver dibujos más que leer números. La siguiente función dibuja con precisión suficiente la curva $\sigma(t)$ obtenida al interpolar los puntos P_i con coordenadas en la las listas `x` e `y`.

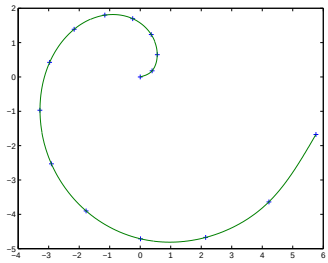
```
function dibujaspline(x,y)
    % Cien puntos entre nodos
    param=linspace(0,1,100*length(y));

    % Dibuja el resultado
    plot(x,y,'+',mifspline(param,x)',mifspline(param,y)')
end
```

Veamos en primer lugar que si dibujamos un número grande de puntos, el resultado de la interpolación es lógico, sigue la forma sin artificios. Por ejemplo, seleccionamos 15 puntos formando una espiral con

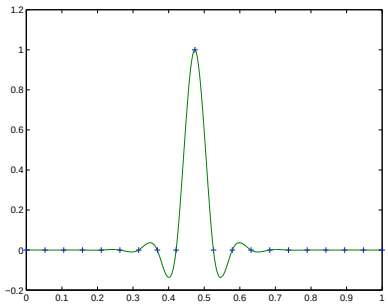
```
a=linspace(0,6.0,15)';
x = a.*cos(a);
y = a.*sin(a);
dibujaspline(x,y);
```

La figura de la derecha es el resultado de interpolar los puntos de la izquierda





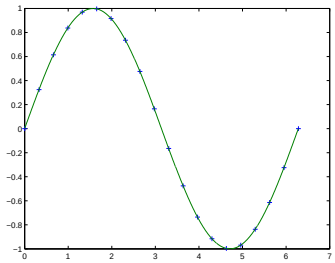
Buscando la analogía con el problema detectado en la interpolación de Lagrange veamos lo que ocurre al interpolar 19 puntos todos al mismo nivel y uno saltado en la posición central:

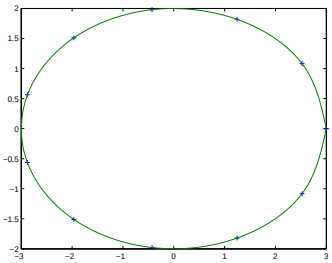


```
a=linspace(0,1,20)';
x = a;
y = 0*a;
y(length(y)/2) = 1;
dibujaspline(x,y);
```

Esta vez las oscilaciones se atenúan rápidamente y el punto saltado no actúa a larga distancia.

Al interpolar curvas conocidas como la gráfica de la función seno o una elipse, también se obtienen buenos resultados.



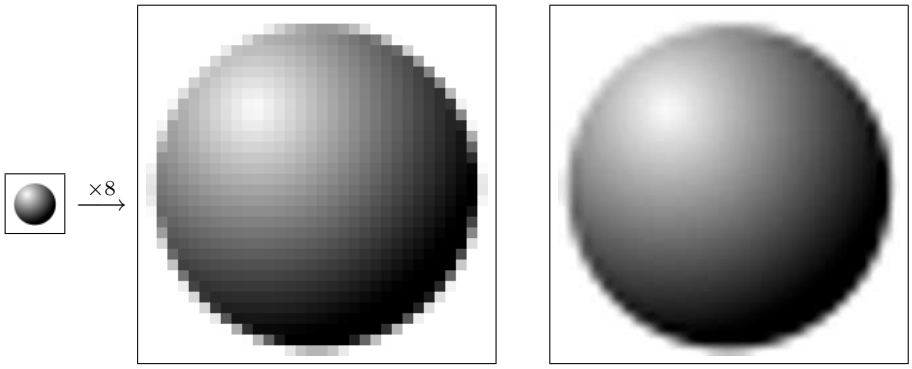


Puestos a poner pegas, hay una pequeña deficiencia en el segundo diagrama, un pico que ocurre en el primer y último punto. Hay una variante para curvas cerradas que en los razonamientos anteriores impone que la derivada sea continua en el punto de cierre, procediendo como si t_0 fuera adyacente a t_n. La implementación es ligeramente distinta, por ello los programas de diseño gráfico que muchos tenemos en nuestro ordenador tienen dos teclas diferentes para crear la curva que une varios puntos, una para curvas abiertas y otra para cerradas. Si usamos la primera en una curva cerrada, típicamente aparece una esquina en la unión del primer y el último punto.

2.4. Un ejemplo en más dimensiones

Nada impide que con las técnicas anteriores se interpolen igualmente puntos de tres, cuatro o cualquier número de coordenadas. Otra cosa distinta es que los nodos vivan en un espacio bidimensional, entonces el objeto con el que interpolamos es más bien una superficie para cada coordenada y hay que considerar polinomios en dos variables. Uno de los ejemplos más simples es la ampliación de imágenes planas. Los *pixels* de que disponemos son los puntos de interpolación que en este caso tendrán tres coordenadas dadas por sus tres componentes de color RGB (o cuatro, RGBA, si hay transparencias) mientras que los nodos son de la forma $\vec{t}_{ij} = (i/n, j/m)$ con $n \times m$ correspondiendo al tamaño original de la imagen.

Creando con GIMP una esferita 32×32 y escalándola a tamaño 256×256, el resultado visual es mucho mejor empleando la opción de interpolación cúbica[3] que usando la opción sin interpolación consistente en aumentar simplemente el tamaño de cada *pixel* (los informáticos, siempre tan amigos de la jerga, a vece3s llaman a esto de no hacer nada *nearest neighbor scaling* por razones que puede imaginar el lector).



Escalado directo de los *pixels* Interpolación cúbica

Evidentemente la interpolación no puede adivinar que queremos dibujar una esfera de bordes bien definidos y en la frontera toma una decisión que no es la óptima. Una alternativa para escalar imágenes con bordes "duros" es combinar la interpolación con un algoritmo de detección de bordes pero no entraremos en ese tema aquí.

[3]Sin revisar el código no sabemos si esta interpolación es por splines como los explicados aquí o por otra variante. Nos inclinamos por la segunda opción ya que en para esta situación no es necesaria mucha precisión y posiblemente los programadores se decidan por implementaciones sencillas utilizando los B-splines que se tratan en el siguiente capítulo.

La diferencia entre las opciones de interpolación lineal y cúbica que admite GIMP visualmente no son tan distintas en muchos ejemplos pero como regla general la segunda da mejores resultados. Por ejemplo, si ampliamos una porción de la frontera del ejemplo anterior, con la interpolación lineal hay unos extraños picos que son convenientemente limados con la cúbica.

2.5. Bibliografía

[Bus] S.R. Buss. *3D Computer Graphics: A Mathematical Introduction with OpenGL*. Cambridge University Press, 2003.

[Ha-St] A. Hardy, W.H. Steeb. *Mathematical tools in computer graphics with C# implementations*. World Scientific 2008.

[Ki-Ch] D. Kincaid, W. Cheney. *Numerical analysis. Mathematics of scientific computing*. Second edition. Brooks/Cole Publishing Co., 1996.

[Lo] B. López. *Cálculo Numérico I (2007/2008)*. Disponible en `http://www.uam.es/bernardo.lopez/CNI_07/cni_07.html`

[St-Bu] J. Stoer, R. Bulirsch. *Introduction to numerical analysis*. Third edition. Springer-Verlag, 2002.

Aplicación 3

No unas los puntos

En el capítulo anterior estudiamos como conseguir una curva suave que pase exactamente por una colección de puntos dados y los splines cúbicos permitían hacerlo de manera óptima en algún sentido. Supongamos que sacrificamos la exactitud por una mayor sencillez en la programación o por una interfaz más intuitiva para el usuario, exigiendo sólo cierto grado de aproximación a los puntos dados.

Por dar un ejemplo más preciso, la mayor parte de la veces que en un programa de diseño gráfico necesitamos trazar una curva sinuosa, deseamos una forma suave de cierto tipo más que un ajuste exacto a unos puntos destacados. En estos programas aparecen típicamente los *puntos de control* de los que estiraremos para deformar la curva de forma imprecisa pero muy intuitiva y local (en contraste con la interpolación donde mover un solo punto altera el aspecto global de la curva). Un ejemplo más complicado, que simplificamos con fines académicos y es más esquemático que real, podría ser la trayectoria de un *bot* (un personaje que se mueve autónomamente) en un videojuego[1]. Internamente el mapa de cada nivel está discretizado y hay algoritmos que seleccionan un camino de puntos que acaba en el punto elegido (el más famoso es el algoritmo A^*). Para que los *bots* amigos o enemigos se dirijan a cierto punto con una trayectoria natural, poco robótica, hay que impedir que el jugador note que hay ciertos puntos de paso comunes a todos ellos (los de la discretización) y es lógico construir una curva que pase sólo cerca.

[1]La situación real en un videojuego es normalmente más compleja porque hay que esquivar a otros personajes en movimiento y evitar situaciones sin salida. El tema se trata ampliamente en [Bu].

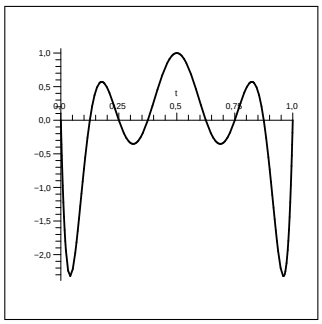
3.1. La idea general

Como hemos apuntado, deseamos determinar una curva $\sigma : [0,1] \longrightarrow \mathbb{R}^2$, $\sigma(t) = (x(t), y(t))$ a partir de ciertos puntos de control $P_i = (x_i, y_i)$, $0 \leq i \leq n$, de manera que al moverlos la curva se deforme de manera intuitiva. Indudablemente es difícil traducir algo tan sicológico como la intuición en términos matemáticos pero después de la experiencia con la interpolación de Lagrange sabemos al menos que ésta no lo es y podemos tratar de subsanar sus inconvenientes.

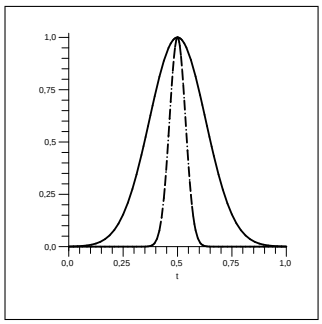
Guardando la analogía, consideremos σ con cada una de sus coordenadas en el espacio vectorial sobre $\mathbb{R}$ generado por cierto conjunto de funciones reales $\Lambda = \{\lambda_0(t), \lambda_1(t), \ldots, \lambda_n(t)\}$ de forma que la curva $\sigma(t) = (x(t), y(t))$ venga dada por la "combinación lineal"

$$\sigma(t) = \sum_{j=0}^{n} \lambda_j(t) P_j$$

donde $t \in [0,1]$ y se consideran $\sigma(t)$ y P_j como vectores de $\mathbb{R}^2$. Si $\lambda_j(t_j) = 1$ y $\lambda_j(t_i) = 0$ para $i \neq j$ se tiene que la curva descrita por σ pasa exactamente por los P_j ya que $\sigma(t_j) = P_j$ (conservamos la notación del capítulo anterior $t_j = j/n$). El problema al escoger λ_j igual a los polinomios de Lagrange es que son funciones muy oscilatorias para n grande y los resultados fuera de los nodos son incontrolables. Las oscilaciones son evidentes examinando el aspecto de su gráfica:



Polinomio de Lagrange $n = 8$



Funciones alternativas

Una opción natural es elegir como λ_j una función suave sin oscilaciones (como la representada en línea discontinua) pero necesariamente la condición $\lambda_j(t_j) = 1$, $\lambda_j(t_i) = 0$, $i \neq j$, obligará a que sea muy puntiaguda y la interpolación dará lugar a una curva con bultos poco naturales. Es mucho más

intuitivo pensar que un punto "tira" de los puntos adyacentes repartiendo parte del 1 que asignaría λ_j al nodo t_j con los nodos de alrededor preservando siempre $\sum \lambda_j(t_i) = 1$. Si ya no se cumple $\lambda_j(t_j) = 1$ entonces típicamente la curva no pasará por P_j pero si conserva una proporción suficiente del 1 que se ha repartido, todavía controlará la curva en sus alrededores atrayéndola hacia sí.

3.2. Curvas de Bézier generales

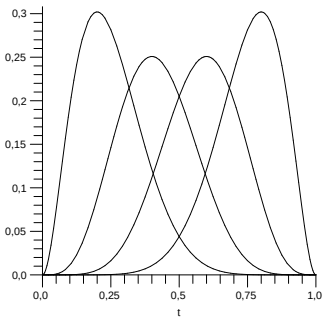
Antes de nada hay que hacer notar que las curvas de Bézier que se usan habitualmente en los programas de diseño gráfico o retoque fotográfico se tratarán en otro apartado posterior y constituyen una variante del caso particular de las estudiadas aquí.

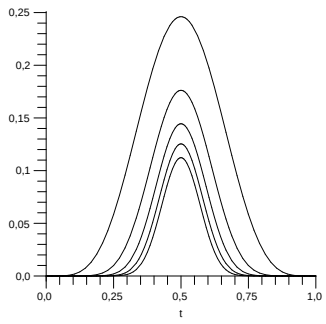
Retomando la idea general buscamos sustituir los polinomios de Lagrange por otros menos ondulantes que hagan una función parecida. Las curvas de Bézier emplean los *polinomios de Bernstein*:

$$B_j(t) = \binom{n}{j} t^j (1-t)^{n-j}$$

que pertenecen a $\mathbb{R}_n[t]$ y que sólo consideraremos para $t \in [0, 1]$. Unas líneas en Maple nos ayudan a visualizar las gráficas de algunos de los B_j para distintos valores de n y j:

```
F:=(N,j)->binomial(N,j)*t^(j)*(1-t)^(N-j);
plot([F(10,2),F(10,4),F(10,6),F(10,8)], t=0..1);
plot([F(10,5),F(20,10),F(30,15),F(40,20),F(50,25)], t=0..1);
```





El primer gráfico muestra que realmente los B_j avanzan con t_j, para n fijado. En el segundo se representa el polinomio central $B_{n/2}$ y deducimos que según crece n disminuye su altura. Este efecto se traduce en que para un número grande de nodos la influencia del punto central sobre el resultado será poca. Por otra parte, los polinomios $B_0 = (1-t)^n$ y $B_n(t) = t^n$ tienen una gráfica con una forma distinta y llegan a alcanzar el valor 1 en $t = 0$ y $t = 1$, lo que les sitúa fuera de la escala de los gráficos anteriores e implica que los puntos extremos tienen mucha influencia. Además las igualdades $B_0(0) = B_n(1) = 1$, y $B_j(0) = B_j(1) = 0$ para $0 < j < n$ implica que la curva siempre comenzará en P_0 y acabará en P_n.

Fácilmente se comprueba que cada función B_j es no negativa, alcanza un máximo en $t = t_j = j/n$ (Análisis Matemático I), es pequeña "lejos" de este valor (las potencias decrecen rápido) y además $\sum_{j=0}^{n} B_j(t)$ es idénticamente 1 (binomio de Newton).

La *curva de Bézier* asociada a los puntos de control $P_0, \ldots, P_n$ viene dada por la combinación lineal

$$\sigma(t) = \sum_{j=0}^{n} B_j(t) P_j, \qquad t \in [0,1].$$

Los gráficos anteriores sugieren que los polinomios B_j están concentrados en cierto grado alrededor de su máximo pero, evidentemente, no son de soporte compacto, por tanto un cambio en alguno de los puntos $P_i = (x_i, y_i)$ modifica toda la curva.

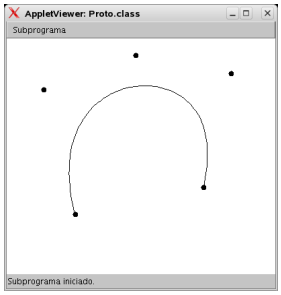
La prueba de fuego es, como siempre en informática, la práctica. Renombramos el *applet* `Lagrange` del capítulo anterior como `Bezier` y sustituimos las llamadas a `lagr` por otras a `bezi`, una función definida por

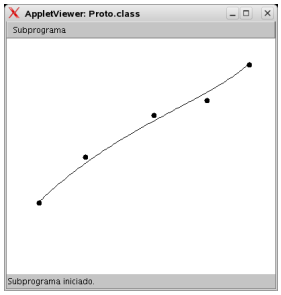
```
// Calcula binom(N-1,i)*t^{i}*(1-t)^{N-1-i}
private double bezi(int i, double t){
        double pr=1.0;
        int j;
        for(j=1;j<=i;++j)
                pr*=t*(N-1-i+j)/(double) j;
        for(j=1;j<=N-1-i;++j)
                pr*=1.0-t;
        return pr;
}
```

Nótese que el factor `(N-1-i+j)/j` en el primer bucle sirve para calcular el

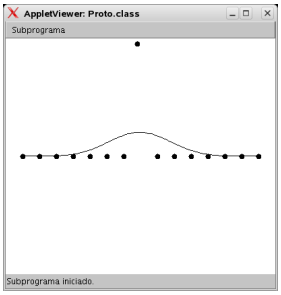
número combinatorio después de haber simplificado algunos factores comunes.

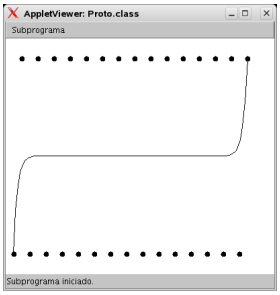
Experimentando con un número pequeño de nodos los resultados son aceptables. Se nota una falta de sensibilidad en algunos casos al movimiento de los puntos de control representada en la primera de las siguientes figuras por la separación entre la curva y los tres puntos de control intermedios. Por otra parte hay una aproximación bastante buena cuando los puntos están ordenados y próximos a una recta.





Para un número grande de nodos examinando uno de los casos problemáticos para la interpolación de Lagrange vemos en el primer gráfico que al "saltar" el punto central no hay una gran variación y de hecho hay que separarlo mucho para que tenga influencia notable.





En el segundo gráfico planteamos una situación que por lógica debiera dar oscilaciones: con la mitad de los puntos a una altura y la otra mitad a otra. Las oscilaciones no se producen y el resultado es extraño. Los puntos de los extremos tiran de la curva produciendo dos pendientes muy pronunciadas mientras que el resto de los puntos no parecen tener influencia alguna.

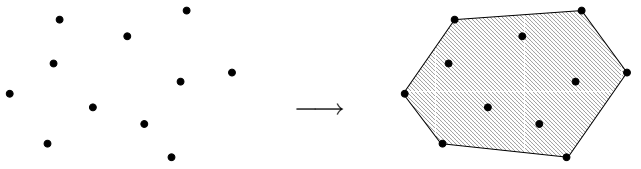
Las propiedades de los polinomios de Bernstein

$$\sum B_j(t) = 1 \qquad \text{y} \qquad B_j(t) \geq 0 \qquad \text{en } [0,1]$$

implican que el dibujo de una curva de Bézier siempre está dentro del conjunto llamado *envoltura convexa* de los puntos de interpolación P_i, que está definido por las siguientes combinaciones lineales restringidas (donde, como antes, consideramos los puntos como vectores):

$$\{\lambda_0 P_0 + \lambda_1 P_1 + \cdots + \lambda_n P_n \ : \ \lambda_i \geq 0, \ \sum_{i=0}^{n} \lambda_i = 1\}.$$

Geométricamente representa el menor polígono que contiene a todos los P_i en su interior o su frontera (es un buen ejercicio probar esta afirmación). Esto explica por qué las curvas de Bézier dan buenos resultados para puntos casi alineados y también por qué no tienen los inconvenientes tan acusados de la interpolación de Lagrange.



Conjunto de puntos — Su envoltura convexa

Según hemos visto, teórica y prácticamente el problema aquí es el contrario al de la interpolación de Lagrange: no hay mucha sensibilidad al movimiento de los puntos que no están cerca de los extremos y para un número grande de nodos es difícil ajustar intuitiva y cómodamente una curva de este manera.

3.3. B-splines cúbicos

La "B" de los B-splines es por *basis* (base en inglés) y viene de que consideraremos bases muy sencillas formadas por splines cúbicos (esencialmente una función y sus trasladados) que a pesar de perder la interpolación exactas son todavía suficientemente intuitivas como para ser prácticas.

Hay que añadir que la definición de B-splines tiene varias versiones y la más general incluye las curvas de Bézier y otros métodos que no mencionaremos aquí. Lo que examinaremos son los B-splines cúbicos uniformes[2].

Para remediar los problemas de las curvas de Bézier buscamos funciones más parecidas a los pulsos que dibujamos en la primera sección cuyo soporte se limite a unos cuantos nodos adyacentes. Además escogeremos como base una función y sus trasladados, lo que simplificará computacionalmente mucho el método y dará uniformidad al resultado.

En términos matemáticos consideramos:

$$\sigma(t) = \sum_j \phi(t - t_j) P_j \tag{3.1}$$

donde ϕ es una función que "vive" en un entorno de cero y por tanto $\phi(t-t_j)$ "vive" en un entorno de t_j. Además para que la masa total sea 1 queremos que $\sum_{j=-\infty}^{\infty} \phi(j/n) = 1$ (por supuesto esta suma sólo tiene una cantidad finita de elementos no nulos).

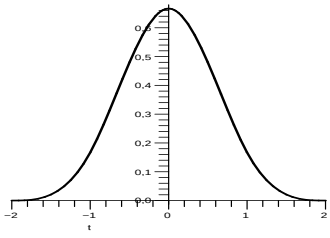
Por un lado deseamos que la gráfica de ϕ no sea muy picuda para no crear oscilaciones antinaturales y por otra no queremos que sea muy achatada para salvaguardar la influencia local.

En el capítulo anterior mencionamos que los splines cúbicos dan soluciones al problema de interpolación que, en cierto sentido, se curvan lo menos posible. Busquemos por tanto cada trasladado de ϕ, esto es, $\phi(t-t_j)$, en el espacio de splines cúbicos idénticamente nulos fuera del intervalo $I = [t_k, t_{k+r}]$. Aparentemente hay $r-1$ grados de libertad: los valores de la función en $t_{k+1}, \ldots, t_{k+r-1}$ pero la anulación fuera de I impone $f'(t_k) = 0$, $f'(t_{k+r}) = 0$, reduciendo estas dos condiciones la dimensión a $r-3 = (r-1)-2$. El primer caso en el que existirá una función no nula es por tanto $r = 4$ y recordando que la suma de sus valores en los nodos debe ser 1, la solución es única. Recapitulando, debemos construir entonces la única función C^2 nula fuera de $[t_k, t_{k+4}]$ definida por polinomios cúbicos en cada subintervalo $[t_j, t_{j+1}]$ cuyos valores en t_{k+1}, t_{k+2} y t_{k+3} sumen uno.

[2]Los usuarios de herramientas de diseño en 3D reconocerán la palabra NURBS, pues bien, es el acróstico de *Non-uniform rational B-splines*, más complicados que los que veremos aquí. Esencialmente *non-uniform* significa que se permiten nodos con diferentes espaciamientos y *rational* que se permiten cocientes de B-splines, lo cual es útil para dibujar en perspectiva.

Para simplificar escalemos el eje horizontal de manera que el intervalo $[t_k, t_{k+4}]$ pase a ser $[-2, 2]$. Unos cálculos prueban que la función que satisface las propiedades requeridas es:

$$S(t) = \begin{cases} Q_1(t+2) & \text{si } -2 \le t \le 1 \\ Q_2(t+1) & \text{si } -1 \le t \le 0 \\ Q_3(t) & \text{si } 0 \le t \le 1 \\ Q_4(t-1) & \text{si } 1 \le t \le 2 \\ 0 & \text{en otro caso.} \end{cases}$$



donde $Q_1(t) = t^3/6$, $Q_2(t) = (1 + 3t + 3t^2 - 3t^3)/6$, $Q_3(t) = Q_2(1-t)$ y $Q_4(t) = Q_1(1-t)$.

La relación con $\phi(t)$ será $\phi(t) = S(nt)$, de este modo cada trasladado $\phi(t - t_j) = S(n(t - t_j))$ es una función que se extiende dos nodos a la derecha y a la izquierda de t_j con el perfil indicado. Al sustituir en (3.1) nos percatamos de que el método es computacionalmente muy barato puesto que en el subintervalo $[t_j, t_{j+1}]$ se tiene

$$\sigma(t) = Q_4(u)P_{j-1} + Q_3(u)P_j + Q_2(u)P_{j+1} + Q_1(u)P_{j+2} \qquad \text{con } u = n(t - t_j).$$

En resumen, en la práctica todo se reduce a hacer cuatro cuentas. Un pequeño problema es que la fórmula anterior pierde sentido en los casos $j = 0$ y $j = n - 1$, para solucionarlo uno puede definir artificialmente $P_{-1} = P_0$ y $P_{n+1} = P_n$.

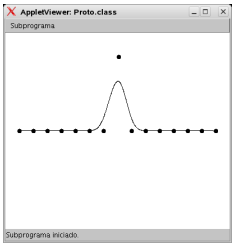
Para que el programa java que hemos usado para la interpolación de Lagrange y las curvas de Bezier permita representar interactivamente las curvas asociadas a los B-splines cúbicos basta reemplazar el bucle `for(double t=0.0; ...)` en `paint` por

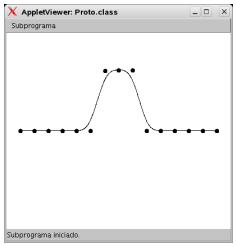
```
for(int i=0;i < N-1;++i){
    for(double t=0.0; t<=1.0; t+=0.01){
        // Polinomios
        q4=(1.0-t)*(1.0-t)*(1.0-t)/6.0;
        q3=(4.0-6.0*t*t+3.0*t*t*t)/6.0;
        q2=(1.0+3.0*t+3.0*t*t-3.0*t*t*t)/6.0;
        q1=t*t*t/6.0;
        // nuevo-> antiguo
        px=qx; py=qy;
```

```
            qx=(double)(puntos[i].x)*q3+(double)(puntos[i+1].x)*q2;
            qy=(double)(puntos[i].y)*q3+(double)(puntos[i+1].y)*q2;

            if( i==0 ){
                qx+=(double)(puntos[0].x)*q4;
                qy+=(double)(puntos[0].y)*q4;
            }
            else{
                qx+=(double)(puntos[i-1].x)*q4;
                qy+=(double)(puntos[i-1].y)*q4;
            }

            if( i==N-2 ){
                qx+=(double)(puntos[N-1].x)*q1;
                qy+=(double)(puntos[N-1].y)*q1;
            }
            else{
                qx+=(double)(puntos[i+2].x)*q1;
                qy+=(double)(puntos[i+2].y)*q1;
            }
            grafico.drawLine((int)(px-.5),(int)(py-.5),(int)(qx-.5),(int)(qy-.5));
        }
    }
```

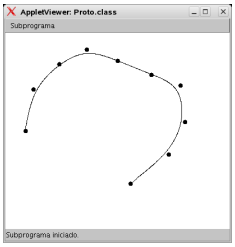
después de haber definido `double q1, q2, q3, q4;` al comienzo.

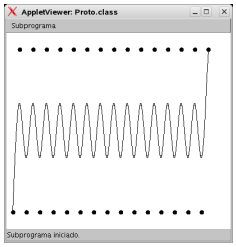
Jugando un poco con la aplicación resultante sentimos que los puntos realmente tiran de la curva independientemente del número de nodos. Visualmente es intuitivo y no cuesta nada aproximar curvas más o menos complicadas, sobre todo porque el resultado es local, afecta sólo a los alrededores, y bastante suave:





Los ejemplos con pocos nodos funcionan también muy bien, y el ejemplo alocado con la mitad de los puntos a cada lado da un resultado razonable:





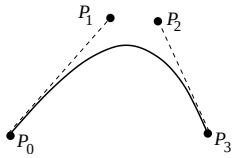
3.4. Las curvas de Bezier en la práctica

En esta sección combinamos los objetivos planteados en el capítulo anterior y en éste. Construiremos una curva que pase exactamente por una parte de los puntos mientras que el resto se utilizan para controlar la forma.

El método está basado en curvas de Bézier cúbicas, esto es, en

$$\sigma(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3.$$

Se cumple $\sigma(0) = P_0$ y $\sigma(1) = P_3$, de modo que la curva interpola estos puntos mientras que P_1 y P_2 controlan su forma. Además lo hacen de una forma intuitiva porque $\sigma'(0) = -3P_0 + 3P_1$, $\sigma'(1) = -3P_2 + 3P_3$, implican que $\overrightarrow{P_0P_1}$ y $\overrightarrow{P_2P_3}$ son vectores tangentes a la curva en P_0 y P_3.



Si añadimos a la derecha otra curva de Bézier cúbica resultará una curva que conecta P_0, P_3 y P_6 y se controla con P_1, P_2, P_4 y P_5. En general con k curvas de Bézier cúbicas unidas logramos conectar $k+1$ pntos por medio de una curva controlada por $2k$ puntos. Con el abuso de notación evidente se llama *curva de Bézier* a esta curva construida a trozos, cuyo uso se ha popularizado en programas de diseño gráfico.

Si tenemos el vicio o la virtud de leer las características técnicas o los avisos del proceso de configuración, sabremos que la mayor parte de las impresoras entienden o simulan *PostScript*. Si además utilizamos LaTeX es posible que incluyamos figuras PostScript por medio de ficheros con apellidos `.ps` o `.eps`. Lo que es menos conocido es que más que un formato, PostScript es un lenguaje de programación (muy simplificado y orientado a gráficos) y que dichos ficheros son simplemente documentos de texto con algunas instrucciones. A veces las instrucciones son tan burdas como indicar el contenido de todos los pixels, por ejemplo cuando el fichero representa una foto (para ello PostScript es poco aconsejable). Sin embargo es posible "programar" figuras sencillas usando unas pocas instrucciones (véase [Bo]) que permiten emplear curvas de Bézier. Si en el fichero `curvaps.eps` incluimos las líneas

```
%!PS-Adobe-2.0 EPSF-2.0
%%Title: curvaps.eps
%%BoundingBox: 0 0 300 300
%%EndComments
newpath
300 200 moveto
220 100
100 200
50 150
curveto

4 setlinewidth
stroke
%%EOF
```

entonces obtendremos la curva de la derecha, que es la curva de Bézier cúbica correspondiente a los puntos $(300, 200)$, $(220, 100)$, $(100, 200)$ y $(50, 150)$. Un resultado bastante bueno teniendo en cuenta que no hemos gastado ni 200 *bytes*, que podemos imprimirlo directamente y que no pierde calidad al reescalar. Para visualizar el resultado muchos de los usuarios de Linux no tendrán que instalar nada, el resto pueden buscar en la red *ghostview*. En [Ca] hay información acerca del trazado de curvas más complejas (ver también el ejemplo sencillo en [Ge]).

Nosotros lo que vamos a hacer es un programa en C que a partir de unos puntos de interpolación calcule unos puntos de control adecuados y genere como salida el texto de un fichero PostScript con la figura resultante (en Linux `./programa.out > figura.eps` creará este fichero). Comenzamos por una cabecera que se explica a sí misma, excepto el significado de `SUAV` sobre el que volveremos después.

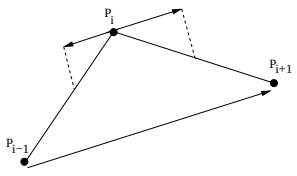
```
#include <stdio.h>
#include <stdlib.h>

#define ANCH 640 /* Ancho de la imagen */
#define ALT  400 /* Alto de la imagen */
#define SUAV 2.0 /* Control de la suavidad */
#define GROS 2 /* Grueso de línea */
#define N 10 /* Número de nodos */

/* Coordenadas de los puntos de interpolación */
int px[N]={ 60, 120, 180, 240, 300, 360, 420, 480, 540, 600};
int py[N]={ 60, 200, 300, 250,  60,  60, 160, 200, 250, 360};
```

La rutina `main` imprime primero la cabecera del fichero PostScript y después los comandos que dibujan círculos rellenos en los puntos de interpolación para que sean fácilmente identificables. El bucle principal llama a las funciones que crean los puntos de control e imprime el comando `curveto` que completará cada curva de Bézier cúbica.

```
int main(int argc, char *argv[]){
    double pcx, pcy; /* Punto de control */
    int i;

    /* Imprime cabecera */
    printf("%%!PS-Adobe-2.0 EPSF-2.0\n%%%%Title: curva.eps\n");
```

```
    printf("%%%%BoundingBox: 0 0 %d %d\n", ANCH, ALT);
    printf("%%%%EndComments\nnewpath\n%d setlinewidth\n",GROS);

    /* Imprime puntos */
    for(i=0; i<N; ++i)
        printf("%d %d 5 0 360 arc\nclosepath\nfill\nstroke\n", px[i], py[i]);

    /* Posición inicial */
    printf("%d %d moveto\n", px[0], py[0]);

    /* Bucle principal */
    for(i=0; i<N-1; ++i){
        /* Calcula el punto de control izquierdo */
        p_cont_izq(i,&pcx,&pcy);
        /* Lo imprime */
        printf("%d %d\n", (int) pcx, (int) pcy);
        /* Calcula el punto de control derecho */
        p_cont_dch(i,&pcx,&pcy);
        /* Lo imprime */
        printf("%d %d\n", (int) pcx, (int) pcy);
        /* Punto i+1 y traza curva */
        printf("%d %d\ncurveto\n", px[i+1], py[i+1]);
    }

    /* Imprime fin de fichero */
    printf("stroke\n%%%%EOF\n");
}
```

El método que usaremos para seleccionar los puntos de control intenta conservar la regularidad de la curva. Si tenemos tres puntos de interpolación consecutivos, P_{i-1}, P_i y P_{i+1}, buscaremos los puntos de control a la derecha y a la izquierda de P_i en la recta r paralela a $\overrightarrow{P_{i-1}P_{i+1}}$ que pasa por P_i. El alineamiento de los puntos de control asegurará que las rectas tangentes en P_i a la derecha y a la izquierda coinciden con r y por tanto la curva será C^1. Parece que una mayor longitud de $\overrightarrow{P_iP_{i-1}}$ o $\overrightarrow{P_iP_{i+1}}$ debería implicar un mayor alejamiento del punto de control correspondiente, por ello los escogeremos como las proyecciones de estos vectores sobre r divididos por `SUAV` (el valor por defecto las reduce a la mitad).

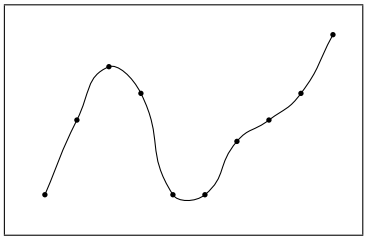


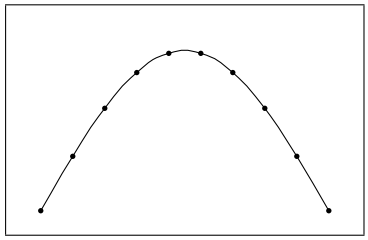
La proyección de un vector $\vec{v}$ en la dirección $\vec{d}$ viene dada por $(\vec{v}\cdot\vec{d})\vec{d}/\|\vec{d}\|^2$ donde $\vec{v}\cdot\vec{d}$ indica el producto escalar, y esto es lo que aparece en las funciones `p_cont_izq` y `p_cont_dch` con las excepciones lógicas para el primer y el último nodo.

```
void p_cont_izq(int i, double *x, double *y){
    double pe;
    if(i!=0){
        *x=px[i+1]-px[i-1];
        *y=py[i+1]-py[i-1];
    }
```

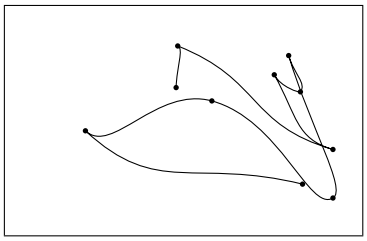
```
    else{
        *x=2*(px[1]-px[0]);
        *y=2*(py[1]-py[0]);
    }
    pe=(px[i+1]-px[i])*(*x) + (py[i+1]-py[i])*(*y);
    pe/=((*x)*(*x) + (*y)*(*y))*SUAV;
    *x=px[i]+pe*(*x);
    *y=py[i]+pe*(*y);
}

void p_cont_dch(int i, double *x, double *y){
    double pe;
    if(i!=N-2){
        *x=px[i+2]-px[i];
        *y=py[i+2]-py[i];
    }
    else{
        *x=2*(px[N-1]-px[N-2]);
        *y=2*(py[N-1]-py[N-2]);
    }
    pe=(px[i+1]-px[i])*(*x) + (py[i+1]-py[i])*(*y);
    pe/=((*x)*(*x) + (*y)*(*y))*SUAV;
    *x=px[i+1]-pe*(*x);
    *y=py[i+1]-pe*(*y);
}
```

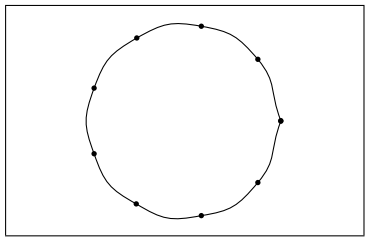
Para puntos de interpolación situados naturalmente los resultados son bastante buenos:





Por mostrar también algunos casos conflictivos, al generar los puntos de interpolación al azar obtenemos la primera figura que dista mucho de ser regular. En la segunda figura se muestra que al situar los puntos en una circunferencia aparece una fea esquina además de que la aproximación es mejorable.





En el primer caso la explicación es que proyecciones muy cortas dan lugar a puntos de control cercanos a los puntos de interpolación y por la propiedad de envoltura convexa las curvas estará demasiado “rectificadas”, a esto

se une que las coordenadas de los *pixels* son números enteros perdiéndose precisión. Este problema no se presenta cuando los vectores $\overrightarrow{P_{i-1}P_i}$ varían lentamente que es lo que ocurre cuando los puntos están situados en una curva regular. Con respecto al problema de la circunferencia, lograríamos una aproximación mejor con splines cúbicos y la esquina (que ya vimos menos acusada usando splines) se debe a que para tratar con éxito curvas cerradas deberíamos imponer que el primer punto de interpolación y el último (coincidentes) tuvieran puntos de control adyacentes alineados con ellos. El programa se podría modificar con este fin.

3.5. Bibliografía

[Bo] P. Bourke. *PostScript Tutorial*. Disponible en `http://local.wasp.uwa.edu.au/~pbourke/dataformats/postscript/`.

[Bu] M. Buckland. *Programming Game AI by Example*. Wordware Game Developer's Library. Wordware Publishing, Inc. 2005.

[Ca] B. Casselman. *Notes on Bezier curves*. Mathematics 308. `http://www.math.ubc.ca/~cass/gfx/curves.pdf`.

[Ge] N.A. Gershenfeld. *The Nature of Mathematical Modeling*. Cambridge University Press 1998.

[Ve-Bi] J.M. Van Verth, L.M. Bishop. *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide* (2nd edition). Morgan Kaufmann, 2008.

Aplicación 4

Comprobar, corregir y mezclar

Algunas de las aplicaciones más relevantes en el mundo de hoy de las matemáticas a la informática se enmarcan dentro de la teoría de códigos y la criptografía. En general se trata de cambiar la forma de un mensaje y el objetivo práctico que se busca es o bien producir redundancia para detectar y corregir errores, o bien eliminar redundancia para economizar memoria o bien ocultar la información para protegerla. Ejemplos cotidianos de estas situaciones son, respectivamente, los lectores de CD, que corrigen multitud de errores en tiempo real, los ficheros con formatos de compresión sin pérdidas (`*.gz`, `*.zip`, etc.) y las transacciones seguras por internet. En los últimos años estas técnicas han crecido de una manera espectacular y no son una parcela del álgebra lineal, por ello lo más que haremos aquí es poner algunos ejemplos para capturar la atención del lector que debe buscar entre la amplia bibliografía existente para tener información más completa.

4.1. Corrigiendo errores

Supongamos que tenemos palabras de cuatro bits $a_3a_2a_1a_0$ y queremos transmitirlas por un canal que comete a lo más un error por cada 12 bits que se transmiten. ¿Cómo podemos asegurarnos de que no hay error al transmitir una palabra. Muy fácil, basta con enviarla tres veces, entonces aunque haya un error en un bit, como lo tenemos repetido tres veces, los buenos ganan por mayoría. Por ejemplo, partiendo de la palabra 1101 podría ocurrir:

$$1101 \longrightarrow 110111011101 \overset{\text{canal defectuoso}}{\rightsquigarrow} 1101\underline{11}\mathbf{1}\underline{1}1101$$

Observando la salida deducimos que en el bloque marcado el tercer 1 debe ser incorrecto ya que no coincide con las otras dos repeticiones del bit y concluimos que la palabra inicial es 1101, la que aparece en las repeticiones correctas.

Nos preguntamos si podríamos traducir $a_3a_2a_1a_0$ en una palabra de menos de 12 bits y aun así ser capaces de detectar y corregir un error, por ejemplo queremos saber si un byte puede almacenar $a_3a_2a_1a_0$ con suficiente redundancia como para que una tasa máxima de error de un bit por byte no corrompa el mensaje. La respuesta parece negativa pues a primera vista el procedimiento anterior es inmejorable. Si un bit se transmite sólo dos veces y una de ellas es defectuoso, es imposible recomponerlo.

No obstante, consideremos el siguiente programa `codifica.c`, tan breve que ni da pereza teclearlo:

```
/* codifica.c */
#include <stdio.h>
#include <stdlib.h>

#define P argv[1]

int main(int argc, char **argv) {
    int x=0,i;
    int bas[4]={112,76,42,105};

    printf("%s -> ",P);
    for (i = 0; i < 4; ++i)
        if( P[i]=='1' ) x^=bas[i];

    i=128;
    while( i/=2 )
        if( x&i ) printf("1");
        else printf("0");
    printf("\n");

    return EXIT_SUCCESS;
}
```

Después de compilarlo con `gcc codifica.c -o codifica` en nuestra consola Linux ejecutamos `./codifica abcd` con `abcd` un número (en binario) de cuatro bits a nuestro antojo (los usuarios de Windows o los que prefieran no usar la consola deben introducir los parámetros con ayuda de su IDE, lo mismo se aplica a otros programas de esta sección). Por ejemplo, la salida tras ejecutar `./codifica 0110` es `1100110` y la de `./codifica 1101` es `1010101`. De esta forma los números de 4 bits se condifican en otros de 7 bits (para hacer el programa medianamente profesional el lector debería incluir como ejercicio una función comprobante de que la entrada es realmente de cuatro bits).

Según lo dicho anteriormente, si modificamos un bit de la salida no parece posible recuperar la entrada. Teclear para ver. Consideremos el programa `descodifica.c` casi tan breve como el anterior que compilaremos con `gcc descodifica.c -o descodifica` y que de nuevo deberíamos completar con una función comprobante de que el parámetro de entrada es un número de 7 bits:

```
/* descodifica.c */
#include <stdio.h>
#include <stdlib.h>

#define P argv[1]

int main(int argc, char **argv) {
    int i;
    int w=0;

    for (i = 0; i < 7; ++i)
        if( P[i]=='1' ) w^=i+1;

    printf("%s -> ",P);
    if ( w!=0 ) {
        printf("Error en el bit %dº (por la izqda)\n",w);
        if ( P[w-1]=='0' ) P[w-1]='1';
        else P[w-1]='0';
        printf("%s -> ",P);
    }
    printf("%c%c%c%c\n",P[2],P[4],P[5],P[6]);

    return EXIT_SUCCESS;
}
```

La salida de `./descodifica 1100110` es `0110` y la de `./descodifica 1010101` es `1101`. El programa entonces en estos ejemplos hace honor a su nombre invirtiendo a `codifica`. El milagro ocurre cuando introducimos un error en alguno de los resultados. Por ejemplo, si en vez de `./descodifica 1010101` escribimos `./descodifica 1010111`, el programa nos dice diligentemente `Error en el bit 6º (por la izqda)` y lo corrige y muestra la descodificación `1101` correspondiente a la entrada corregida. No es casualidad, podemos hacer todas las pruebas que queramos e incluso completar los programas o diseñar un *script* para que se comprueben todas las posibilidades. Lo creamos o no, nuestro ordenador siempre acierta dónde estamos intentando colar un error y es capaz de corregirlo. En contra de nuestra intuición no ha sido necesario ni siquiera duplicar el tamaño de la palabra inicial para ser capaces de corregir un error.

La pregunta desde el punto de vista matemático es: ¿dónde radican estos poderes adivinatorios de estos simples programas? y la respuesta será en el núcleo de una aplicación lineal entre ciertos espacios vectoriales.

Para un informático principiante la pregunta sería: ¿tiene esto sentido en

un mundo donde la tecnología ha llegado a tal perfeccionamiento que puedo mover de aquí a allá gigas y gigas sin un solo error? La respuesta es un rotundo sí. Es más, parte de la responsabilidad de que tal cosa sea posible radica en la teoría de códigos que con sus algoritmos ha permitido corregir errores en tiempo real. La grabación en soportes físicos como el DVD está al límite de la tecnología y es imposible asegurar que las marcas micrométricas que indican los bits están bien hechas. Un bit erróneo no será posiblemente desastroso al ver el vídeo del último efímero ídolo del pop pero imaginemos qué ocurriría si cambiase la dirección de llamada de la rutina `main` de nuestra obra de arte informática.

Los espacios vectoriales naturales para tratar problemas de codificación y de corrección de errores son los definidos sobre cuerpos finitos. Cuerpos de este tipo son los $\mathbb{Z}_p$ con p primo (clases de residuos módulo p) que se suelen denotar mediante $\mathbb{F}_p$ cuando se quiere hacer hincapié en que son cuerpos (***f**ield* es el término inglés para cuerpo) y de paso se elimina la posibilidad de confusión con otra cosa que también se llama $\mathbb{Z}_p$ en matemáticas. El caso $p = 2$ es de singular interés en informática porque $\mathbb{F}_2 = \{0, 1\}$ representa el posible valor de un bit y una palabra de n bits queda reflejada en un vector $\vec{x} \in \mathbb{F}_2^n$.

Los *códigos lineales* sobre $\mathbb{F}_2$ son vectores de $\mathbb{F}_2^n$ (palabras de n bits) que pertenecen al núcleo de una aplicación lineal sobreyectiva $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^{n-k}$. De la fórmula para la dimensión

$$\dim \mathrm{Nuc}(f) = n - \dim \mathrm{Im}(f) = n - (n - k) = k.$$

Cada elemento $\vec{x} \in \mathrm{Nuc}(f)$ está en $\mathbb{F}_2^n$ pero a su vez está caracterizado por los k bits que conforman sus coordenadas con respecto a una base de $\mathrm{Nuc}(f)$. En definitiva, $\mathrm{Nuc}(f)$ contiene 2^k vectores de n coordenadas (bits) y cada uno de ellos se puede considerar la codificación de una palabra de k bits.

Un error en $\vec{x}$ se representa con un cambio de $\vec{x}$ por $\vec{y} = \vec{x} + \vec{e}_i$ donde $\vec{e}_i \in \mathbb{F}_2^n$ es el vector que tiene un 1 en la coordenada i y ceros en el resto. Detectaremos que $\vec{y}$ es erróneo si $\vec{y} \notin \mathrm{Nuc}(f)$, es decir, si $f(\vec{e}_i) \neq \vec{0}$. Para poder corregir el error debemos asegurarnos además de que $f(\vec{e}_i)$ permite reconstruir $\vec{e}_i$, esto es, que no existe $j \neq i$ tal que $f(\vec{e}_i) = f(\vec{e}_j)$. Escribamos $f(\vec{x}) = H\vec{x}$ donde $H \in \mathcal{M}_{(n-k)\times n}(\mathbb{F}_2)$, a esta matriz de la aplicación lineal se le denomina *matriz de paridad* del código. Entonces la condición para que se pueda corregir un error es que todas las columnas de H sean distintas y no

nulas. La opción más simple es elegir como columnas todos los vectores no nulos de k coordenadas. En el ejemplo anterior $n = 7$, $k = 4$ y elegimos

$$H = \begin{pmatrix} 0&0&0&1&1&1&1 \\ 0&1&1&0&0&1&1 \\ 1&0&1&0&1&0&1 \end{pmatrix} \Rightarrow \text{Nuc}(f) = \left\langle \left\{ \begin{pmatrix} 1\\1\\1\\0\\0\\0\\0 \end{pmatrix}, \begin{pmatrix} 1\\0\\0\\1\\1\\0\\0 \end{pmatrix}, \begin{pmatrix} 0\\1\\0\\1\\0\\1\\0 \end{pmatrix}, \begin{pmatrix} 1\\1\\0\\1\\0\\0\\1 \end{pmatrix} \right\} \right\rangle.$$

Los vectores $\vec{v}_1$, $\vec{v}_2$, $\vec{v}_3$, $\vec{v}_4$ indicados como generadores del núcleo forman una base y lo único que hace el primer programa al ejecutar `./codifica abcd` es calcular la combinación lineal $\vec{x} = a\vec{v}_1 + b\vec{v}_2 + c\vec{v}_3 + d\vec{v}_4$.

Si un error ha reemplazado $\vec{x}$ por $\vec{y} = \vec{x} + \vec{e}_i$ entonces el segundo programa lo detecta porque $\vec{y} \notin \text{Nuc}(f)$, además $f(\vec{y})$ es la i-ésima columna de H y se puede saber fácilmente dónde está el error (el número binario que representa la i-ésima columna es i). Las coordenadas tercera, quinta, sexta y séptima de $\vec{x}$ son respectivamente a, b, c y d, entonces una vez subsanado el posible error la descodificación se reduce a una línea de programa: `printf("%c%c%c%c\n",P[2],P[4],P[5],P[6])`.

¿Cómo podríamos subsanar no uno, sino hasta dos errores en una palabra de n bits? Repitiendo el razonamiento anterior habría que verificar que no existen $\vec{e}_i$, $\vec{e}_j$, $\vec{e}_k$, $\vec{e}_l$ distintos tales que

$$f(\vec{e}_i) = \vec{0} \;\text{ o }\; f(\vec{e}_i) = f(\vec{e}_j) \;\text{ o }\; f(\vec{e}_i) = f(\vec{e}_j + \vec{e}_k) \;\text{ o }\; f(\vec{e}_i + \vec{e}_j) = f(\vec{e}_k + \vec{e}_l).$$

Gracias a la linealidad y a que en $\mathbb{F}_2$ sólo hay dos números, el cero y el 1, todo esto equivale a que cuatro columnas cualesquiera de H sean linealmente independientes sobre $\mathbb{F}_2$. En general hallar un código lineal sobre $\mathbb{F}_2$ que sea capaz de subsanar hasta t errores es lo mismo que hallar una matriz $H \in \mathcal{M}_{(n-k)\times n}(\mathbb{F}_2)$ tal que $2t$ columnas cualesquiera sean linealmente independientes sobre $\mathbb{F}_2$ (Th.3.2.11 [Mu-Mu]).

Es un buen ejercicio computacional diseñar un programa que busque por fuerza bruta códigos que corrijan muchos errores con n/k no demasiado grande (para economizar bits). Naturalmente la materia gris ha llegado a límites inalcanzables por la fuerza bruta y en la actualidad disponemos de varios métodos para crear códigos eficientes [Hi], [Li-Xi], [Mu-Mu].

4.2. Códigos de todos los días

Si en el esquema de la sección anterior $n - k = 1$ entonces el código es demasiado simple como para corregir errores sin embargo se utiliza en varios

contextos cotidianos cambiando $\mathbb{F}_2$ por otro cuerpo con más elementos para detectar errores con cierta probabilidad. Al terminar de leer más de un lector concederá que las matemáticas no quedan tan lejos de nuestra vida diaria, simplemente permanecen escondidas.

La letra del DNI. El Documento Nacional de Identidad contiene un número n de 8 dígitos acompañado de una letra. A cada n le asignaremos el elemento correspondiente de $\mathbb{F}_{23}$ calculando su resto al dividir por 23, mientras que a la letra le asignaremos el lugar que ocupa en la cadena

`TRWAGMPYDFXBNJZSQVHLCKE`

comenzando por cero, es decir, `T` $\mapsto 0$ y `E` $\mapsto 22$. De esta forma el número y la letra dan lugar a un elemento de $\mathbb{F}_{23}^2$. Se utiliza el código viene dado por la aplicación lineal

$$f : \mathbb{F}_{23}^2 \longrightarrow \mathbb{F}_{23}, \qquad f(\vec{x}) = H\vec{x}, \qquad \text{con} \quad H = \begin{pmatrix} 1 & -1 \end{pmatrix}.$$

En palabras sencillas, la letra está determinada por el resto módulo 23 del número como muestra el siguiente programa brevísimo:

```
#include <stdio.h>
#include <stdlib.h>

/*  Calcula la letra del DNI/NIF */
int main(int argc, char *argv[]){
    unsigned char a[23]="TRWAGMPYDFXBNJZSQVHLCKE";
        printf("Número=%s, Letra=%c\n",argv[1],a[ (atoi(argv[1]) %23) ]);
}
```

Ejecutando `./dni 12345678` averiguaremos que la letra que corresponde al número de DNI `12345678` es la `Z`. Entonces la letra funciona como un código de control que permite detectar (pero no corregir) errores en la mayor parte de las ocasiones. Por ejemplo, si al copiar el DNI anterior escribiéramos erróneamente `12345687 Z` nuestro sencillo programilla sabría que algo no va bien pues `./dni 12345687` indica que la letra es la `T` en vez de la `Z`.

El ISBN-10 y los códigos de barras. Todo libro se identifica con un número de diez dígitos (a menudo separadas en grupos por guiones o espacios) que aparece en la portada interior o en la cubierta o aledaños, en algunas ocasiones el último dígito no es tal, sino una `X`. Recientemente este código llamado ISBN se complementa o reemplaza por 13 dígitos acompañados de

un código de barras, de los que también hablaremos, y que posiblemente llegarán a sustituir al ISBN. Para distinguir ambos se especifica que el ISBN de toda la vida es el ISBN-10.

Cada ISBN se puede considerar un vector de $\mathbb{F}_{11}^{10}$ donde el dígito en el lugar i es la coordenada i-ésima y si el último carácter es una `X` en lugar de un dígito, suponemos que representa 10 (recuérdense los números romanos). El ISBN-10 está basado en la matriz de paridad:

$$H = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{pmatrix}.$$

que corresponde a una aplicación lineal $f : \mathbb{F}_{11}^{10} \longrightarrow \mathbb{F}_{11}$.

Con esta información, un *script* muy sencillo en *python* (seguramente no muy bien programado) permite comprobar si un ISBN (que se puede introducir con o sin guiones) es válido:

```
import string

tab={'X':10,'x':10}
for i in range(10):
    tab[string.digits[i]]=i

cod = str(raw_input("Posible ISBN: ")).replace('-', '')
if len(cod)!=10:
    print 'Debe tener 10 caracteres'
else:
    j=0
    for i in range(10):
        j+=(i+1)*tab[cod[i]]

    if j %11 == 0:
        print 'Es un ISBN-10 admisible'
    else:
        print 'No es un ISBN-10 admisible'
```

Los trece dígitos que posiblemente sustituyan al ISBN-10 responden al formato EAN-13 que se emplea para detectar errores en códigos de barras (de nuevo hay más de un formato de códigos de barras pero no entraremos en ello). Como las barras representan tradicionalmente dígitos nos debemos olvidar del carácter especial `X`. La matriz de paridad que se considera es:

$$H = \begin{pmatrix} 1 & 3 & 1 & 3 & 1 & 3 & 1 & 3 & 1 & 3 & 1 & 3 & 1 \end{pmatrix}.$$

sobre $\mathbb{Z}_{10}$. El problema (teórico) es que $\mathbb{Z}_{10}$ no es un cuerpo, ya que en él no se puede dividir: por ejemplo $2 \cdot 0 = 2 \cdot 5 = 0$ implica que $0/2$ debería dar a la vez 0 y 5. Si $\mathbb{Z}_{10}$ no es un cuerpo $\mathbb{Z}_{10}^{13}$ tampoco será estrictamente un

espacio vectorial. Ciertamente aun así es algo que se le parece (esta especie de espacios vectoriales en los que no es posible dividir por números se llaman en álgebra *módulos*) y en este ejemplo práctico tan sencillo no es muy relevante pero la teoría no se puede copiar ya que hay sorpresas desagradables por ejemplo cuando se trata de extender el concepto básico de independencia lineal.

El Código Cuenta Cliente. Nuestra cuenta corriente tiene asignada un número de 20 dígitos que se emplea en cualquier transacción, es el *Código Cuenta Cliente* (CCC). Los cuatro primeros dígitos indican la entidad, los cuatro siguientes la sucursal y los diez últimos constituyen el número de cuenta propiamente dicho. Los dos dígitos restantes son los *dígitos de control* (marcados con DC) y están determinados por el resto. Imaginemos que desde un cajero automático hacemos una transferencia y por una equivocación en un número enviamos el dinero a otra persona. El número de veces que nos preguntan si estamos seguros y los dígitos de control son la manera de que esta posibilidad sea remota.

Los 20 dígitos se pueden considerar conformando un vector de $\mathbb{F}_{11}^{20}$. Salvo una pequeña chapuza que indicaremos a continuación. El código lineal sobre $\mathbb{F}_{11}$ es el que corresponde a la matriz de paridad

$$H = \begin{pmatrix} 2^2 & 2^3 & 2^4 & 2^5 & 2^6 & 2^7 & 2^8 & 2^9 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2^0 & 2^1 & 2^2 & 2^3 & 2^4 & 2^5 & 2^6 & 2^7 & 2^8 & 2^9 \end{pmatrix}.$$

Naturalmente todas estas potencias de 2 pueden reducirse módulo 11.

Dado un banco, una sucursal y un número de cuenta, cada dígito de control es un número de 0 a 10, ambos incluidos. Entonces no puede representarse realmente con un dígito, recordemos que en el ISBN esto se resolvía introduciendo la `X`. Aquí se hace una chapucilla permitiendo que el 1 valga 1 o 10. De esta forma los dígitos de control que contienen al 1 son menos seguros pues en realidad representan más de una posibilidad.

El siguiente programa sirve para calcular los dígitos de control a partir del resto de los dígitos y comprobar si coinciden con los que se introducen en la entrada.

```
/* cocucu.c */
#include <stdio.h>
#include <stdlib.h>

#define P argv[1]
```

```
int main(int argc, char **argv) {
    int dc1=11000,dc2=11000;
    int i;

    /* Comprueba formato */
    for(i=0; i<20; ++i)
        if( (P[i]<'0')||(P[i]>9) ) i=100;
    if( (P[20]!=0)||(i==100) ){
        printf("Formato erróneo: Debería tener 20 dígitos 0-9\n");
        return EXIT_SUCCESS;
    }

    /* Calcula dígitos de control */
    for (i = 2; i < 10; ++i) dc1-=(1<<i)*(P[i-2]-48);
    for (i = 0; i < 10; ++i) dc2-=(1<<i)*(P[i+10]-48);
    if( (dc1%=11)==10 ) dc1=1;
    if( (dc2%=11)==10 ) dc2=1;

    /* Comprueba si coinciden */
    if( (dc1+48!=P[8])||(dc2+48!=P[9]) )
        printf("Los dígitos de control deberían ser %d y %d y son %c y %c\n",
                                                              dc1,dc2,P[8],P[9]);
    else
        printf("Código cuenta cliente %s validado\n",P);

    return EXIT_SUCCESS;
}
```

Tras crear el ejecutable `cocucu`, con `./cocucu 23571111210123456789` sabremos que el número indicado es un código cuenta cliente admisible. Si intercambiamos los dos últimos dígitos el programa protestará lanzando el mensaje `Los dígitos de control deberían ser 2 y 4 y son 2 y 1` que sugiere que el error está en el número de cuenta, pues el primer dígito de control es correcto. Si deseamos saber por ejemplo qué dígitos de control corresponden a la entidad 1927, la sucursal 0429 y al número de cuenta 2999409504 basta ejecutar `./cocucu 19270429872999409504` (los dígitos de control se han escrito al azar) y esperar a que los corrija.

4.3. Reordenar con matrices

La función f que asigna a cada n el resto de a^n al dividir por p (primo) es biyectiva de $\{1, 2, \ldots, p-1\}$ en sí mismo cuando a es lo que se llama una *raíz primitiva* (véase [Ro]). Computacionalmente es muy fácil hallar $f(n+1)$ a partir de $f(n)$ multiplicando por a y calculando el resto pero por otra parte es difícil reconocer a simple vista un patrón en $f(1),\ f(2),\ f(3), \ldots\ f(p-1)$. Por esta razón a la función f se utiliza en ocasiones como generador de

números (pseudo)aleatorios. Por ejemplo, la función `RND` del prehistórico ZX-Spectrum lo único que hacía era multiplicar por $a = 75$ y reducir módulo $p = 65537$, presentando el resultado normalizado en $(0, 1]$. Con números mayores más en línea con las tecnologías actuales el procedimiento sigue en vigor. La biyectividad de f implica que se puede emplear para hacer repartos evitando repeticiones sin tener que ocupar memoria para recordar los números ya asignados.

Aquí vamos a introducir una variante bidimensional de este proceso con fines académicos y la aplicaremos para crear un efecto de pulverizado en una imagen.

Trabajaremos con $p = 31$ y lo que buscamos es una manera de desordenar los elementos de $X = \mathbb{F}_{31}^2 - \{(0, 0)\}$, es decir, los puntos $\{(x, y) \ : \ 0 \leq x, y < 31\}$ salvo el origen. En analogía con lo anterior tomaremos

$$f(n) = A^n \vec{v}_0 \qquad \text{para algún} \quad \vec{v}_0 \in X$$

y queremos que $f : \{1, 2, 3, \ldots, 31^2 - 1\} \longrightarrow X$ sea biyectiva.

Si A diagonaliza en $\mathbb{F}_{31}^2$ entonces

$$A = C^{-1} \begin{pmatrix} \alpha & 0 \\ 0 & \beta \end{pmatrix} C \qquad \Longrightarrow \qquad A^n = C^{-1} \begin{pmatrix} \alpha^n & 0 \\ 0 & \beta^n \end{pmatrix} C$$

pero un resultado de Álgebra I (el pequeño teorema de Fermat) afirma que si $\alpha, \beta \in \mathbb{F}_{31} - \{0\}$ entonces $\alpha^{30} \equiv \beta^{30} \equiv 1$ módulo p y se estropea la biyectividad porque todo se repite de 30 en 30. ¿Y si consideramos A tal que no tenga autovalores en $\mathbb{F}_{31}$? Por ejemplo

$$\begin{pmatrix} 0 & 3 \\ 3 & 1 \end{pmatrix} \qquad \Longrightarrow \qquad |A - \lambda I| = \lambda^2 - \lambda - 9$$

y $\lambda^2 - \lambda - 9 = 0$ no tiene raíces en $\mathbb{F}_{31}$. ¿Dónde estan entonces las raíces? Pues en un cuerpo formado por una especie de números complejos módulo 31. Además la teoría asegura que siempre algunos de los elementos del cuerpo (los análogos de las raíces primitivas) cumplen que sus potencias tardan $31^2 - 1$ en repetirse. Como habrá adivinado el lector, la matriz A citada anteriormente tiene autovalores α y β con esta propiedad. En esta situación no es difícil ver que para $\vec{v}_0 \in X$ la colección finita

$$\vec{v}_1 = A\vec{v}_0, \quad \vec{v}_2 = A\vec{v}_1, \quad \vec{v}_3 = A\vec{v}_2, \ldots \qquad \vec{v}_{31^2-1} = A\vec{v}_{31^2-2}$$

da lugar a la reordenación buscada de los elementos de X.

Dicho sea de paso, ésta es una de esas *funciones hash* que son difíciles de invertir. Con un ordenador aquí sería trivial por fuerza bruta pero cambiando 31 por un primo de cien cifras el cálculo de $f(n)$ quedaría todavía dentro de límite computacionales razonables mientras que el de $f^{-1}(n)$ sería casi siempre impracticable. Este círculo de ideas tiene gran interés en criptografía [Ro].

Es muy fácil hacer un programa que con esta técnica cambie de forma aparentemente aleatoria y secuencialmente los puntos de una imagen de tamaño 31×31 del color de fondo, digamos el negro. Ello crea un efecto de que la imagen se pulveriza. Lo único malo es que el programa dependerá de la biblioteca gráfica favorita de cada cual. Una muy sencilla, eficiente y totalmente multiplataforma es SDL. El sitio oficial es `http://www.libsdl.org/` y [Ga] contiene una descripción sencilla y razonablemente exhaustiva al respecto.

Las siguientes líneas incorporan una variante que permite graduar el tamaño del pixel (parámetro `ZOOM`) para manejar imágnes mayores. Todo lo que hay que saber para entender esta función es que `SDL_FillRect(imd,&r,0)` rellena de negro el rectángulo `r` de `imd`, que es la pantalla en este caso, `SDL_Flip(imd)` actualiza la pantalla y `SDL_Delay(ESP)` espera unos milisegundos.

```
#include <SDL/SDL.h>
#include <stdio.h>
#include <stdlib.h>

#define ZOOM 2 /* Tamaño en comparación con 31 */
#define ESP 3 /* Tiempo de espera en ms */

void pulveriza(SDL_Surface *imd){
    int i,a,b;
    SDL_Rect r={0,0,ZOOM,ZOOM};

    a= rand() % 31; b= rand() % 30; /* Primer punto al azar */

    SDL_FillRect(imd,&r,0); /* Quita el origen */
    SDL_Flip (imd); /* Actualiza */

    r.x=a*ZOOM;    r.y=++b*ZOOM;

    /* Bucle quita puntos */
    for(i=0; i<960; ++i){
        SDL_FillRect(imd,&r,0);
        SDL_Flip (imd); /* Actualiza */
        SDL_Delay(ESP); /* Espera */
        r.x = (3*b) %31;
        r.y = b = (3*a+b) %31;
        a = r.x;
        r.x*=ZOOM;   r.y*=ZOOM;
    }

}
```

El resto del programa es la incialización de SDL y la carga de la imagen.

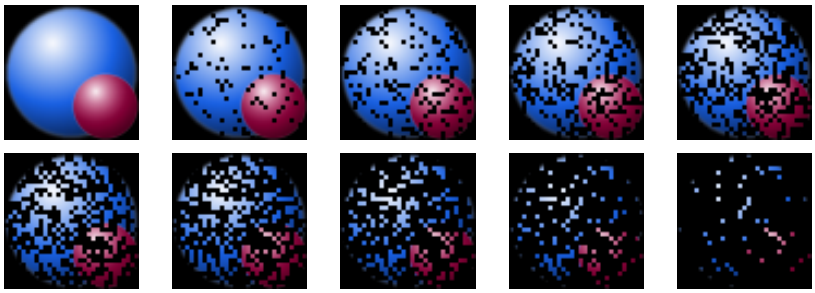
```
#define ANCHO (31*ZOOM) /* Ancho de pantalla */
#define ALTO (31*ZOOM) /* Alto de pantalla */
#define IMG "bolota.bmp" /* Nombre de la imagen */

int main(int argc, char** argv) {

    SDL_Surface *screen; /* La pantalla */
    SDL_Surface *image; /* La imagen */

    /* Inicia SDL */
    if((SDL_Init(SDL_INIT_VIDEO)==-1)) {
        printf("Error al iniciar SDL: %s.\n", SDL_GetError());
        exit(-1);
    }
    atexit(SDL_Quit);

    /* Crea pantalla */
    putenv("SDL_VIDEO_CENTERED=1"); /* Centra la ventana */
    if ( (screen = SDL_SetVideoMode(ANCHO, ALTO, 32, SDL_SWSURFACE)) == NULL ) {
        printf( "No se puede crear la pantalla: %s\n", SDL_GetError());
            exit(-1);
    }

    /* Carga la imagen en la pantalla */
    if ( (image = SDL_LoadBMP(IMG)) == NULL) {
        printf("Error al cargar " IMG ": %s\n", SDL_GetError());
        exit(-1);
    }
    SDL_BlitSurface(image, NULL, screen, NULL); /* A la pantalla */
    SDL_FreeSurface(image); /* Libera la imagen */

    pulveriza(screen);

    SDL_Delay(1000); /* Espera un segundo */
    SDL_Quit(); /* Cierra SDL */
    return EXIT_SUCCESS;
}
```

Una vez instalada la biblioteca SDL se compila con

```
gcc  pulveriza.c -o pulveriza ‘sdl-config --cflags --libs‘ -lm
```

(nótese que las comillas son comillas derechas, como acentos graves). Si al ejecutar `./pulveriza` se produjese un error relativo a un formato de pantalla no soportado, cámbiense `(31*ZOOM)` y `(31*ZOOM)` en las definiciones de `ANCHO` y `ALTO` por ejemplo por `320` y `240`.

Teniendo en el directorio la imagen `bolota.bmp` (en este caso de tamaño 62×62) el resultado fue:



La primera imagen es la original y después se muestra lo ocurrido tras 100, 200, etc. iteraciones del bucle. Al cabo de $960 = 31^2 - 1$ la imagen será completamente negra.

4.4. Ocultar información

El código ASCII (*American Standard Code for Information Interchange*) representa con números del 0 al 127 letras, cifras y algunos caracteres de control y símbolos. Su versión extendida contiene caracteres especiales de ciertas lenguas y más símbolos, empleando números de 0 a 255, un byte. Aquí trabajaremos en $\mathbb{F}_{251}$ que también cabe en un byte y cubre todos los caracteres habituales y muchos más. De este modo una cadena de caracteres es una cadena de números para un ordenador:

```
unsigned char a[]="Soy una cadena";
int i=-1;
while( a[++i]!=0 ) printf("%c %d\n",a[i],a[i]);
```

produce la tabla (que giramos por razones tipográficas):

S	o	y		u	n	a		c	a	d	e	n	a
83	111	121	32	117	110	97	32	99	97	100	101	110	97

Supongamos que tenemos un texto secreto que deseamos ocultar a ojos curiosos. Una solución sería recodificarlo por medio de cierta función f. Distingamos dos casos límite: Si el texto tiene N caracteres y f actúa globalmente sobre él, se tendría $f : \mathbb{F}_{251}^N \longrightarrow \mathbb{F}_{251}^N$, mientras que si f actúa sobre el texto modificándolo carácter a carácter, $f : \mathbb{F}_{251} \longrightarrow \mathbb{F}_{251}$. Ambas situaciones no son muy convenientes. Si N es grande $f : \mathbb{F}_{251}^N \longrightarrow \mathbb{F}_{251}^N$ podría ser muy costosa de manejar, además cuando se cambia la longitud del texto, N, hay que cambiar la función. Si actúa carácter a carácter entonces una simple comparación de la frecuencia de cada número con la de las letras en un texto típico permitiría al curioso desvelar un texto suficientemente grande.

Un modo muy sencillo de evitar el análisis de frecuencias es considerar una función $f : \mathbb{F}_{251}^k \longrightarrow \mathbb{F}_{251}^k$ con k mayor que la longitud media de una palabra. Aquí tomaremos $k = 4$ por simplicidad. No es muy gravoso suponer que N es múltiplo de k (siempre es posible añadir uno, dos o tres espacios aquí o allá si fuera necesario o completar la cadena de caracteres con unos pocos caracteres superfluos). En esta situación, los N caracteres que forman

el texto se representan con N/k vectores de $\mathbb{F}_{251}^k$. Cada uno de estos vectores $\vec{v}$ se codifica con $f(\vec{v})$ y se descodifica con $f^{-1}(\vec{v})$.

Las aplicaciones lineales son un buen instrumento para este fin, ya que es posible hallar su inversa algorítmicamente. Pero seamos todavía más exigentes: ¿es posible que el mismo algoritmo codifique y descodifique? Si $f(\vec{v}) = A\vec{v}$ con $A \in \mathcal{M}_{4\times4}(\mathbb{F}_{251})$ entonces $f = f^{-1}$ equivale a $A^2 = I$ ¿y cómo inventarse una matriz con esta propiedad? Obviamente una matriz diagonal D con $d_{ii} \in \{1, -1\}$ $(= \{1, 250\}$ en $\mathbb{F}_{251})$ es una solución trivial pero poco útil porque no oculta la información de manera significativa: simplemente deja los caracteres igual o los cambia de signo. Para esconder su aspecto basta cambiar de base:

$$A = C^{-1}DC \quad \text{con } D = \begin{pmatrix} 1 & & & \\ & -1 & & \\ & & 1 & \\ & & & -1 \end{pmatrix} \Rightarrow A^2 = I.$$

Sirve cualquier matriz C de cambio de base para la que A no sea sencilla. Lo mejor es escoger $C \in \mathcal{M}_{4\times4}(\mathbb{Z})$ con determinante uno y así no habrá que calcular ningún inverso módulo 251, ya que los elementos de C^{-1} serán automáticamente enteros. Crear matrices enteras de determinante uno no es difícil, por ejemplo usando transformaciones elementales sobre la matriz identidad o inventando elementos al azar dejando algunos de ellos como paramétros para ajustar el determinante. Elegimos

$$C = \begin{pmatrix} 7 & 3 & 4 & 9 \\ 5 & 5 & 2 & 8 \\ 7 & 3 & 3 & 8 \\ 3 & 0 & 5 & 6 \end{pmatrix} \Rightarrow C^{-1}DC = \begin{pmatrix} -71 & -30 & -82 & -132 \\ -182 & -79 & -202 & -332 \\ -210 & -90 & -235 & -384 \\ 210 & 90 & 236 & 385 \end{pmatrix} \xrightarrow[\mathbb{F}_{251}]{} A = \begin{pmatrix} 180 & 221 & 169 & 119 \\ 69 & 172 & 49 & 170 \\ 41 & 161 & 16 & 118 \\ 210 & 90 & 236 & 134 \end{pmatrix}.$$

A pesar de su feo aspecto esta matriz cumple $A^2 = I$ en $\mathbb{F}_{251}$ por construcción.

Digamos que nuestro texto misterioso comienza con "En un lugar de la Mancha...", en ASCII

```
 69,  110,  32,  117,  110,  32,  108,  117,  103,  97,  114,  32,
100,  101,  32,  108,   97,  32,   77,   97,  110,  99,  104,  97,  ...
```

Para codificar agrupamos de 4 en 4 y aplicamos A. Los tres primeros bloques dan lugar a

$$A\begin{pmatrix} 69 \\ 110 \\ 32 \\ 117 \end{pmatrix} = \begin{pmatrix} 88 \\ 210 \\ 219 \\ 181 \end{pmatrix}, \quad A\begin{pmatrix} 110 \\ 32 \\ 108 \\ 117 \end{pmatrix} = \begin{pmatrix} 62 \\ 124 \\ 96 \\ 129 \end{pmatrix}, \quad A\begin{pmatrix} 103 \\ 97 \\ 114 \\ 32 \end{pmatrix} = \begin{pmatrix} 50 \\ 179 \\ 89 \\ 57 \end{pmatrix}$$

y los tres siguientes a:

$$A\begin{pmatrix}100\\101\\32\\108\end{pmatrix}=\begin{pmatrix}98\\24\\234\\157\end{pmatrix},\quad A\begin{pmatrix}97\\32\\77\\97\end{pmatrix}=\begin{pmatrix}143\\81\\221\\204\end{pmatrix},\quad A\begin{pmatrix}110\\99\\104\\97\end{pmatrix}=\begin{pmatrix}16\\20\\176\\25\end{pmatrix}.$$

Entonces la codificación del texto comenzaría por

```
88,  210,  219,  181,   62,  124,   96,  129,  50,  179,   89,  57,
98,   24,  234,  157,  143,   81,  221,  204,  16,   20,  176,  25,  ...
```

En ASCII esto corresponde a la obra de arte:

XÒÛÛμ>|` [129]2³Y9b[24]ê[157][143]QÝÌ[16][20]°[25]

donde los números recuadrados indican los caracteres no imprimibles con el código ASCII señalado.

Para ver mejor los caracteres podemos escribir todo en hexadecimal. El siguiente conversor actúa sobre un fichero de texto y muestra la traducción hexadecimal en pantalla:

```
/*  Texto a hexadecimal */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]){
    char c; /* Carácter */
    FILE *ifp; /* Fichero de entrada */

    if(argc<2){
        printf("Hay que especificar un fichero.\n");
        return EXIT_FAILURE;
    }

    ifp=fopen(argv[1],"r");
    if(ifp==NULL){
        printf("No se puede abrir el fichero \"%s\"\n",argv[1]);
        return EXIT_FAILURE;
    }

    while ((c=getc(ifp))!=EOF) printf("%X",c);

    fclose(ifp);
    printf("\n");
    return EXIT_SUCCESS;
}
```

Si el ejecutable es `texto2hex` hay que teclear `./texto2hex texto.txt` para hacerlo actuar sobre el fichero `texto.txt`. Este proceso se puede invertir con un programa `hex2texto` idéntico salvo definir `char c[]="00";` reemplazar el bucle `while` por

```
 while ((c[0]=getc(ifp))!=EOF){
     c[1]=getc(ifp);
     printf("%c",strtol(c,NULL,16));
}
```

De esta forma, al aplicar `texto2hex` sobre nuestro texto se obtiene

456E20756E206C7567617220646520 6C61204D616E636861...

La acción del algoritmo antes descrito viene representado por el siguiente programa que llamaremos `autoinv` ya que es "autoinvertible". Lo que hace es aplicar la matriz A a la traducción hexadecimal de cada grupo de 4 códigos ASCII. La función `mat_vec` lleva a cabo la multiplicación matricial mientras que en `main` se encuentra el bucle que recorre la cadena e imprime los resultados:

```
/*  Código autoinverso */
#include <stdio.h>
#include <stdlib.h>

#define P argv[1]

void mat_vec(int *v){
    int a,b,c;
    /* Matriz de orden 2 por vector */
    a=180*v[0]+221*v[1]+169*v[2]+119*v[3];
    b= 69*v[0]+172*v[1]+ 49*v[2]+170*v[3];
    c= 41*v[0]+161*v[1]+ 16*v[2]+118*v[3];
    v[3]= (210*v[0]+ 90*v[1]+236*v[2]+134*v[3])%251;
    v[0]= a%251;
    v[1]= b%251;
    v[2]= c%251;
}

int main(int argc, char *argv[]){
    int i=0;
    int j;
    char s[2]="00";
    int v[4];

    while( P[i]!=0 ){
        for(j=0; j<4; ++j){
            s[0]=P[i+2*j];
            s[1]=P[i+2*j+1];
            v[j]= strtol(s,NULL,16);
        }
        i+=8;
        mat_vec(v);
        printf("%X%X%X%X",v[0],v[1],v[2],v[3]);
    }
    printf("\n");
}
```

Aplicar `./autoinv` sobre la cadena anterior conduce a:

```
58D2DBB53E7C608132B359396218EA9D8F51DDCC1014B019...
```

que es justamente la traducción hexadecimal del texto ilegible. Aplicando de nuevo `./autoinv`, ahora sobre esta cadena, obtenemos la cadena inicial que se descodifica con `hex2texto` para obtener el texto. En comandos de consola `./autoinv` *(cadena anterior)* `>hextexto.txt` y `./hex2texto hextexto.txt` mostrarían en pantalla "En un lugar de la Mancha...", el texto descodificado, mientras que en `hextexto.txt` se guardaría su traducción hexadecimal.

4.5. Bibliografía

[Ga] A. García Serrano. *Programacion de videojuegos con SDL en Windows y Linux*. Ediversitas multimedia, Sevilla, 2003.

[Hi] R. Hill. *A first course in coding theory*. The Clarendon Press, Oxford University Press, New York, 1986.

[Li-Xi] S. Ling, C. Xing. *Coding theory. A first course*. Cambridge University Press, Cambridge, 2004.

[Mu-Mu] G.L. Mullen, C. Mummert. *Finite fields and applications*. American Mathematical Society, Providence, 2007.

[Ro] K.H. Rosen. *Elementary number theory and its applications*. Fourth edition. Addison-Wesley, Reading, MA, 2000.

Parte II

Resúmenes de teoría

Resúmenes 5

Temario

El temario comentado empleado en los últimos años está recogido en la siguiente lista:

1. **Sistemas de ecuaciones lineales. Reducción de Gauss. Reducción de Gauss-Jordan.**
   Métodos básicos para resolver sistemas de ecuaciones lineales. El resto del curso los usa para resolver distintos problemas dentro del álgebra lineal.
2. **Matrices. Operaciones entre matrices. Matriz inversa. Matrices y sistemas de ecuaciones lineales.**
   Las matrices son el instrumento básico para realizar cálculos efectivos en álgebra lineal. Se incluye el algoritmo para calcular la matriz inversa usando la reducción de Gauss-Jordan.
3. **Espacios y subespacios vectoriales. Aplicaciones lineales.**
   Definición y propiedades básicas de los espacios vectoriales y sus subespacios. Todo subespacio vectorial de k^n dado mediante generadores se puede también describir mediante ecuaciones lineales.
4. **Bases y dimensión.**
   Definiciones y cálculo de bases y dimensiones de subespacios vectoriales de k^n. Matriz de una aplicación lineal en una base dada. Número mínimo de ecuaciones lineales que describen un subespacio.
5. **Suma e intersección de subespacios.**
   Se incluye la fórmula de Grassmann y se discute cómo calcular la suma y la intersección de subespacios.

6. **Rango. Teorema de Rouché-Frobenius. Cambio de base para matrices de aplicaciones lineales.**
   Se estudia el teorema del rango y se utiliza para caracterizar la resolubilidad de los sistemas lineales.
7. **Determinantes. Regla de Cramer.**
   Definición y propiedades de los determinantes. Caracterización de la independencia lineal y de la invertibilidad de matrices mediante determinantes.
8. **Valores y vectores propios. Diagonalización. Aplicaciones.**
   Se incluye el teorema de diagonalización, con aplicación a la resolución de los sistemas de ecuaciones diferenciales lineales de primer orden con coeficientes constantes. También se tratará el caso discreto.
9. **Producto escalar. Formas bilineales.**
   Definiciones y desigualdades básicas (Cauchy-Schwarz, Minkowski y triangular). Matriz de una forma bilineal y cambio de base para la matriz de una forma bilineal.
10. **Ortogonalización. Proyección ortogonal.**
    Algoritmo de Gram-Schmidt y cálculo de la aplicación de proyección ortogonal sobre un subespacio.
11. **Aplicaciones ortogonales y autoadjuntas. Teorema espectral.**
    Diagonalización de aplicaciones autoadjuntas.
12. **Formas cuadráticas**
    Algoritmo de Gauss para la reducción de formas cuadráticas. Signatura.

5.1. Ecuaciones lineales y su resolución

Sistemas de ecuaciones lineales Un sistema de ecuaciones lineales es de la forma indicada a la izquierda y se suele representar por una matriz (una tabla) como la de la derecha:

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = & b_2 \\ \cdots \quad \cdots \quad \cdots \quad \cdots & = & \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n & = & b_m \end{array} \quad \leftrightarrow \quad \left(\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & \ldots & a_{2n} & b_2 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} & b_m \end{array}\right)$$

Como veremos más adelante en el curso, un sistema de este tipo puede tener solución única (compatible determinado), infinitas soluciones (compatible indeterminado) o no tener solución (incompatible).

Reducción de Gauss La matriz de un sistema es una *matriz escalonada* (o el sistema está en *forma escalonada*) si cada fila no nula tiene siempre más ceros a la izquierda que la que está por encima y las filas nulas, si las hubiera, están colocadas al final.

Siempre es posible reducir un sistema a forma escalonada empleando tres *transformaciones elementales* sobre las ecuaciones (o equivalentemente sobre las filas de la matriz):

1. Sumar a una ecuación un múltiplo de otra.
2. Multiplicar una ecuación por un número no nulo.
3. Intercambiar dos ecuaciones.

Todas ellas se pueden invertir, así que no se pierden soluciones del sistema al aplicarlas.

El algoritmo de *reducción de Gauss* consiste en aplicar estos tres procesos (el segundo no es estrictamente necesario) para producir ceros por columnas en la matriz y llegar a la forma escalonada.

Por ejemplo, consideremos el sistema:

$$\begin{array}{rrrl} x & +2y & +3z & = 2 \\ x & -y & +z & = 0 \\ x & +3y & -z & = -2 \\ 3x & +4y & +3z & = 0 \end{array}$$

Primero se crean los ceros en la primera columna bajo el primer elemento:

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 2 \\ 1 & -1 & 1 & 0 \\ 1 & 3 & -1 & -2 \\ 3 & 4 & 3 & 0 \end{array}\right) \begin{array}{c} \longrightarrow \\ {\scriptstyle f_2 \mapsto f_2 - f_1} \\ {\scriptstyle f_3 \mapsto f_3 - f_1} \\ {\scriptstyle f_4 \mapsto f_4 - 3f_1} \end{array} \left(\begin{array}{ccc|c} 1 & 2 & 3 & 2 \\ 0 & -3 & -2 & -2 \\ 0 & 1 & -4 & -4 \\ 0 & -2 & -6 & -6 \end{array}\right)$$

Intercambiar las filas es superfluo pero nos permite evitar los cálculos con fracciones al crear los ceros de la segunda columna:

$$\begin{array}{c} \longrightarrow \\ {\scriptstyle f_2 \leftrightarrow f_3} \end{array} \left(\begin{array}{ccc|c} 1 & 2 & 3 & 2 \\ 0 & 1 & -4 & -4 \\ 0 & -3 & -2 & -2 \\ 0 & -2 & -6 & -6 \end{array}\right) \begin{array}{c} \longrightarrow \\ {\scriptstyle f_3 \mapsto f_3 + 3f_2} \\ {\scriptstyle f_4 \mapsto f_4 + 2f_2} \end{array} \left(\begin{array}{ccc|c} 1 & 2 & 3 & 2 \\ 0 & 1 & -4 & -4 \\ 0 & 0 & -14 & -14 \\ 0 & 0 & -14 & -14 \end{array}\right)$$

Con un paso más llegamos a la forma escalonada:

$$\begin{array}{c} \longrightarrow \\ {\scriptstyle f_4 \mapsto f_4 - f_3} \end{array} \left(\begin{array}{ccc|c} |\underline{1} & 2 & 3 & 2 \\ 0 & |\underline{1} & -4 & -4 \\ 0 & 0 & |\underline{-14} & -14 \\ 0 & 0 & 0 & 0 \end{array}\right)$$

Los elementos señalados se llaman *elementos pivote* y señalan el principio de los "escalones". Con más rigor, un elemento pivote en una matriz escalonada es un elemento no nulo que tiene ceros a la izquierda. Las columnas que contienen a los elementos pivote se llaman *columnas pivote*.

Una vez que se ha llegado a la forma escalonada es fácil resolver el sistema (o deducir que no tiene solución), despejando de abajo a arriba las ecuaciones. Así en el ejemplo anterior la tercera ecuación de la forma escalonada implica $z = 1$, sustituyendo en la segunda se tiene $y = 0$ y estos resultados en la primera dan $x = -1$.

Reducción de Gauss-Jordan El algoritmo de *reducción de Gauss-Jordan* es similar al del Gauss pero cuando se ha finalizado éste se procede a crear ceros encima de los elementos pivote empleando las filas de abajo a arriba sin modificar la estructura escalonada. Multiplicando por un número adecuado (transformación 2) también se consigue que los elementos pivote sean unos. Esta forma escalonada en la que los elementos pivote son unos y el resto

de los elementos de las columnas pivote son ceros a veces se llama *forma escalonada reducida*.

En el ejemplo anterior dividiendo entre -14 en la tercera fila los pivotes serán unos:

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 2 \\ 0 & 1 & -4 & -4 \\ 0 & 0 & -14 & -14 \\ 0 & 0 & 0 & 0 \end{array}\right) \underset{f_3 \mapsto -f_3/14}{\longrightarrow} \left(\begin{array}{ccc|c} 1 & 2 & 3 & 2 \\ 0 & 1 & -4 & -4 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right)$$

Ahora creamos ceros encima del tercer elemento pivote y después del segundo:

$$\underset{\substack{f_2 \mapsto f_2+4f_3 \\ f_1 \mapsto f_1-3f_3}}{\longrightarrow} \left(\begin{array}{ccc|c} 1 & 2 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right) \underset{f_1 \mapsto f_1-2f_2}{\longrightarrow} \left(\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right)$$

Al emplear la reducción de Gauss-Jordan en la columna de la derecha leeremos la solución del sistema, si es que es única. Si una de la últimas ecuaciones fuera "$0 =$*constante no nula*" entonces se llegaría a una contradicción y no habría solución. En otro caso, si hay columnas que no son columnas pivote las incógnitas correspondientes se pueden elegir como parámetros arbitrarios.

El algorimto de Gauss Jordan es conveniente para resolver simultáneamente varios sistemas que comparten la misma *matriz de coeficientes* (la formada por los a_{ij}). Para ello simplemente se añaden nuevas columnas correspondientes a los diversos sistemas.

Por ejemplo, para resolver simultáneamente

$$\begin{array}{rrrl} x & -2y & +z & = 0 \\ 3x & -6y & +2z & = 0 \end{array} \quad \text{y} \quad \begin{array}{rrrl} x & -2y & +z & = 2 \\ 3x & -6y & +2z & = 1 \end{array}$$

La matriz a considerar sería

$$\left(\begin{array}{ccc|cc} 1 & -2 & 1 & 0 & 2 \\ 3 & -6 & 2 & 0 & 1 \end{array}\right)$$

que se reduce a forma escalonada en un solo paso

$$\left(\begin{array}{ccc|cc} 1 & -2 & 1 & 0 & 2 \\ 3 & -6 & 2 & 0 & 1 \end{array}\right) \underset{f_2 \mapsto f_2-3f_1}{\longrightarrow} \left(\begin{array}{ccc|cc} 1 & -2 & 1 & 0 & 2 \\ 0 & 0 & -1 & 0 & -5 \end{array}\right).$$

Creamos ahora un cero encima del segundo elemento pivote y lo reducimos a uno:

$$\underset{f_2\mapsto -f_2}{\longrightarrow} \left(\begin{array}{ccc|cc} 1 & -2 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 & 5 \end{array}\right) \underset{f_1\mapsto f_1-f_2}{\longrightarrow} \left(\begin{array}{ccc|cc} 1 & -2 & 0 & 0 & -3 \\ 0 & 0 & 1 & 0 & 5 \end{array}\right).$$

En ambos casos la segunda variable es un parámetro arbitrario, digamos $y = \lambda$ y se tiene como soluciones del primer y del segundo sistema, respectivamente:

$$\left\{\begin{array}{l} x = 2\lambda \\ y = \lambda \\ z = 0 \end{array}\right. \qquad \text{y} \qquad \left\{\begin{array}{l} x = -3 + 2\lambda \\ y = \lambda \\ z = 5. \end{array}\right.$$

5.2. Álgebra matricial

Matrices y sus operaciones Ya habíamos visto que una *matriz* A es simplemente una tabla de números. La notación habitual es llamar a_{ij} al elemento que está en la fila i y en la columna j.

Denotaremos mediante $\mathcal{M}_{m\times n}(K)$ al conjunto de todas las matrices de m filas y n columnas con coeficientes en un cuerpo[1] K.

Las matrices de $\mathcal{M}_{m\times n}(K)$ se suman de la forma esperada: sumando los elementos en las mismas posiciones. Para multiplicarlas por $\lambda \in K$ se multiplica cada elemento por λ.

La multiplicación de dos matrices sólo se define si el número de columnas de la primera coincide con el número de filas de la segunda. Si $A \in \mathcal{M}_{m\times n}(K)$ y $B \in \mathcal{M}_{n\times l}(K)$ entonces $AB \in \mathcal{M}_{m\times l}(K)$. El elemento ij del producto se calcula con la fórmula $\sum a_{ik}b_{kj}$. Esto equivale a decir que se hace el producto escalar habitual de la fila i de A por la columna j de B.

Por ejemplo:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 3 & 5 \end{pmatrix}, \qquad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ -1 & 0 \\ -2 & 1 \end{pmatrix} = \begin{pmatrix} -8 & 3 \\ -17 & 6 \end{pmatrix}.$$

[1] Un cuerpo es, intuitivamente, un conjunto en el que se puede sumar, restar, multiplicar y dividir (salvo por cero) con las propiedades habituales. En este curso el cuerpo más común será $\mathbb{R}$ pero también están $\mathbb{Q}$, $\mathbb{C}$ o las clases de restos módulo un número primo.

Dada una matriz A, se llama *matriz traspuesta*, y se denota con A^t, a la obtenida al intercambiar las filas por las columnas en A.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \in \mathcal{M}_{2\times 3}(\mathbb{R}), \qquad A^t = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} \in \mathcal{M}_{3\times 2}(\mathbb{R}).$$

Con respecto a la suma y el producto, la traspuesta tiene las propiedades:

$$1)\ (A+B)^t = A^t + B^t, \qquad 2)\ (AB)^t = B^t A^t$$

Tipos de matrices y matrices especiales:

Las matrices de $\mathcal{M}_{n\times n}(K)$, por razones obvias, se dice que son *matrices cuadradas*.

Las matrices cuadradas A tales que $a_{ij} = 0$ cuando $i \neq j$ se denominan *matrices diagonales*.

La matriz diagonal $A \in \mathcal{M}_{n\times n}(K)$ tal que $a_{ii} = 1$ para todo i se dice que es la *matriz identidad* y se suele denotar con I. Es el elemento neutro de la multiplicación en $\mathcal{M}_{n\times n}(K)$, es decir $A = IA = AI$ para cualquier $A \in \mathcal{M}_{n\times n}(K)$.

La matriz de $\mathcal{M}_{m\times n}(K)$ que tiene todos sus elementos cero se llama *matriz nula* y a veces se denota con O. Es el elemento neutro de la suma, es decir $A = A + O = O + A$.

Matriz inversa Dada una matriz cuadrada $A \in \mathcal{M}_{n\times n}(K)$ se dice que es *invertible* y que su *matriz inversa* es $B \in \mathcal{M}_{n\times n}(K)$ si $I = AB = BA$. A la matriz inversa de A se la denota con A^{-1}.

La inversa puede no existir, pero si existe es única.

Algunas propiedades de la inversa:

$$1)\ (AB)^{-1} = B^{-1}A^{-1}, \qquad 2)\ (A^t)^{-1} = (A^{-1})^t, \qquad 3)\ (A^{-1})^{-1} = A.$$

Para hallar la matriz inversa de $A \in \mathcal{M}_{n\times n}(K)$ uno puede tratar de resolver la ecuación matricial $AX = I$ donde X es una matriz $n \times n$ cuyos elementos son incógnitas. Esto conduce a n sistemas de ecuaciones, todos ellos con la misma matriz de coeficientes e igualados a cada una de las columnas de I. Con ello se deduce que el cálculo de la inversa equivale a aplicar el algoritmo de Gauss-Jordan a $(A|I)$. Si existe la inversa (y sólo en ese caso) el final del algoritmo será $(I|A^{-1})$.

Por ejemplo, calculemos la inversa de

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 3 \\ 2 & -1 & -8 \end{pmatrix}.$$

Los pasos del algoritmo de Gauss-Jordan son:

$$\underset{f_3 \mapsto f_3 - 2f_1}{\longrightarrow} \left(\begin{array}{ccc|ccc} 1 & 2 & 0 & 1 & 0 & 0 \\ 0 & 1 & 3 & 0 & 1 & 0 \\ 0 & -5 & -8 & -2 & 0 & 1 \end{array}\right) \underset{f_3 \mapsto f_3 + 5f_2}{\longrightarrow} \left(\begin{array}{ccc|ccc} 1 & 2 & 0 & 1 & 0 & 0 \\ 0 & 1 & 3 & 0 & 1 & 0 \\ 0 & 0 & 7 & -2 & 5 & 1 \end{array}\right)$$

$$\underset{f_3 \mapsto f_3/7}{\longrightarrow} \left(\begin{array}{ccc|ccc} 1 & 2 & 0 & 1 & 0 & 0 \\ 0 & 1 & 3 & 0 & 1 & 0 \\ 0 & 0 & 1 & -2/7 & 5/7 & 1/7 \end{array}\right) \underset{f_2 \mapsto f_2 - 3f_3}{\longrightarrow} \left(\begin{array}{ccc|ccc} 1 & 2 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 6/7 & -8/7 & -3/7 \\ 0 & 0 & 1 & -2/7 & 5/7 & 1/7 \end{array}\right)$$

$$\underset{f_1 \mapsto f_1 - 2f_2}{\longrightarrow} \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & -5/7 & 16/7 & 6/7 \\ 0 & 1 & 0 & 6/7 & -8/7 & -3/7 \\ 0 & 0 & 1 & -2/7 & 5/7 & 1/7 \end{array}\right) \Longrightarrow A^{-1} = \begin{pmatrix} -5/7 & 16/7 & 6/7 \\ 6/7 & -8/7 & -3/7 \\ -2/7 & 5/7 & 1/7 \end{pmatrix}.$$

Matrices y sistemas de ecuaciones lineales Los sistemas de ecuaciones lineales se pueden escribir como una simple ecuación matricial $A\vec{x} = \vec{b}$ donde

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}, \qquad \vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \qquad \vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}.$$

Si A es una matriz invertible entonces el sistema tiene solución única dada por $\vec{x} = A^{-1}\vec{b}$.

Las transformaciones elementales en los algoritmos de Gauss y Gauss-Jordan pueden escribirse en términos de multiplicaciones de matrices.

5.3. Espacios vectoriales

Espacios y subespacios vectoriales. Un *espacio vectorial* sobre un cuerpo es intuitivamente un conjunto en el que tenemos definida una suma y una multiplicación por números (los elementos de un cuerpo) con las propiedades

habituales. La definición rigurosa es más complicada requiriendo la estructura algebraica de grupo abeliano con la suma y cuatro propiedades que ligan la suma y la multiplicación.

Los elementos de un espacio vectorial se llaman *vectores* y los elementos del cuerpo (los números) *escalares*.

Los ejemplos de espacios vectoriales de espacios vectoriales sobre un cuerpo K más importantes en este curso son:

1. K^n que es el producto cartesiano $K \times \overset{n \text{ veces}}{\ldots} \times K$ con la suma y el producto por elementos de K definidos de la manera obvia. Por razones que serán claras más adelante, escribiremos en columna la colección de n elementos de K que representan cada vector de K^n. El ejemplo más común en el curso será $\mathbb{R}^n$.
2. $\mathbb{P}[x]$, el conjunto de polinomios en la variable x con coeficientes en K.
3. $\mathcal{M}_{m\times n}(K)$.

Un *subespacio vectorial* es un espacio vectorial incluido en otro con las mismas operaciones. Las propiedades de espacio vectorial se cumplen inmediatamente en un subconjunto siempre que las operaciones estén bien definidas, por ello para demostrar que cierto subconjunto S de un espacio vectorial sobre K es un subespacio basta comprobar:

$$1)\ \vec{u}, \vec{v} \in S \Rightarrow \vec{u} + \vec{v} \in S \qquad \text{y} \qquad 2)\ \vec{u} \in S,\ \lambda \in K \Rightarrow \lambda\vec{u} \in S$$

Por ejemplo, $\mathbb{R}_n[x]$, el conjunto de todos los polinomios con coeficientes reales de grado menor o igual que n (incluyendo al polinomio cero) son un subespacio de $\mathbb{P}[x]$ y en particular un espacio vectorial. Sin embargo los polinomios de grado exactamente 3 no lo son porque $1 - x^3$ y $x + x^3$ están en este conjunto pero su suma no.

Un ejemplo de subespacio vectorial muy frecuente este curso es el subconjunto de K^n dado por las soluciones $\vec{x} \in K^n$ del sistema $A\vec{x} = \vec{0}$ donde $A \in \mathcal{M}_{m\times n}(K)$ (el convenio de los vectores columna hace que el producto de matrices tenga sentido).

Dados vectores $\vec{v}_1, \vec{v}_2, \ldots \vec{v}_n$, una *combinación lineal* de ellos es cualquier expresión del tipo $\lambda_1\vec{v}_1 + \lambda_2\vec{v}_2 + \cdots + \lambda_n\vec{v}_n$ con $\lambda_1, \ldots \lambda_n \in K$.

Los vectores que se obtienen como combinaciones lineales de elementos de un conjunto de vectores C forman el *subespacio generado por* C que se denota con $\langle C\rangle$ ó $\mathcal{L}(C)$. Es fácil ver que realmente es un subespacio vectorial.

Dado un subconjunto finito de C de K^n se puede escribir $\langle C\rangle$ "con ecuaciones", es decir, como $\{\vec{x} \in K^n \ : \ A\vec{x} = \vec{0}\}$ eliminando parámetros por reducción de Gauss en una combinación lineal genérica de los elementos de C.

Por ejemplo

$$C = \left\{ \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix} \right\} \quad \Rightarrow \quad \langle C\rangle = \left\{ \begin{pmatrix} x \\ y \\ x \end{pmatrix} \in \mathbb{R}^3 \ : \ \begin{pmatrix} x \\ y \\ x \end{pmatrix} = \lambda \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} + \mu \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix} \right\}.$$

Empleando la reducción de Gauss

$$\left(\begin{array}{cc|c} 1 & 1 & x \\ 0 & \text{-2} & y \\ \text{-1} & 3 & z \end{array}\right) \to \left(\begin{array}{cc|c} 1 & 1 & x \\ 0 & \text{-2} & y \\ 0 & 4 & x+z \end{array}\right) \to \left(\begin{array}{cc|c} 1 & 1 & x \\ 0 & \text{-2} & y \\ 0 & 0 & x+2y+z \end{array}\right).$$

Entonces la ecuación que define $\langle C\rangle$ es $x + 2y + z = 0$ (la matriz A sería en este caso (1 2 1), una matriz fila).

Recíprocamente, dado un subespacio de la forma $S = \{\vec{x} \in K^n \ : \ A\vec{x} = \vec{0}\}$, resolviendo esta ecuación se llegará a una solución en términos de parámetros (o simplemente la solución $\vec{0}$ que da lugar al subespacio trivial) que puede escribirse como una combinación lineal de ciertos vectores de K^n. El conjunto formado por dichos vectores satisface $\langle C\rangle = S$.

Aplicaciones lineales. Una *aplicación lineal* es una función entre espacios vectoriales que preserva las operaciones. Es decir, es una función $f : V \longrightarrow W$ tal que

1) $\vec{u}, \vec{v} \in V \Rightarrow f(\vec{u}+\vec{v}) = f(\vec{u})+f(\vec{v})$ y 2) $\vec{u} \in V,\ \lambda \in K \Rightarrow f(\lambda\vec{u}) = \lambda f(\vec{u})$.

El ejemplo que más utilizaremos en el curso (y al cual reduciremos prácticamente todos los ejemplos) es $f : K^n \longrightarrow K^m$ dada por $f(\vec{x}) = A\vec{x}$ con $A \in \mathcal{M}_{m\times n}(K)$.

Hay dos subespacios asociados a una aplicación lineal $f : V \longrightarrow W$, el *núcleo* y la *imagen*, definidos respectivamente como:

$$\mathrm{Nuc}(f) = \{\vec{v} \in V \ : \ f(\vec{v}) = \vec{0}\}, \qquad \mathrm{Im}(f) = \{\vec{w} \in W \ : \ \exists \vec{v} \text{ con } f(\vec{v}) = \vec{w}\}.$$

La aplicación lineal $f : V \longrightarrow W$ es inyectiva si y sólo si $\text{Nuc}(f) = \{\vec{0}\}$ y es sobreyectiva si y sólo si $\text{Im}(f) = W$. Las funciones que son inyectivas y sobreyectivas se dice que son biyectivas (y el caso de aplicaciones lineales se dice que establecen un *isomorfismo* entre V y W). Las funciones biyectivas tiene una *función inversa*, que en nuestro caso será la aplicación lineal $f^{-1} : W \longrightarrow V$ tal que $f^{-1} \circ f$ es la identidad en V y $f \circ f^{-1}$ es la identidad en W (esto significa que dejan todos los elementos invariantes en estos espacios).

Por ejemplo, si $f : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ viene dada por $f(\vec{x}) = A\vec{x}$ con $A \in \mathcal{M}_{n\times n}(\mathbb{R})$ invertible, entonces la función inversa es $f^{-1}(\vec{x}) = A^{-1}\vec{x}$.

En el caso $f : K^n \longrightarrow K^m$ con $A \in \mathcal{M}_{m\times n}(K)$, utilizando la definición obtenemos una descripción de $\text{Nuc}(f)$ con ecuaciones, $A\vec{x} = \vec{0}$, que una vez resueltas conduce a un conjunto que genera este subespacio. Por otro lado, las columnas de A siempre generan $\text{Im}(f)$ porque la estructura de la multiplicación de matrices hace que $A\vec{x} = \sum x_i\vec{c}_i$ con $\vec{c}_i$ la columna i-ésima de A. Con el procedimiento introducido antes, a partir del conjunto de columnas se pueden obtener las ecuaciones que determinan $\text{Im}(f)$.

Por ejemplo, dada la aplicación lineal $f : \mathbb{R}^2 \longrightarrow \mathbb{R}^3$ definida por

$$f\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & -2 \\ -1 & 3 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix},$$

hallemos su núcleo y su imagen.

Al igualar a $\vec{0}$ y resolver el sistema se obtiene únicamente la solución trivial $x = y = 0$, por tanto $\text{Nuc}(f) = \{\vec{0}\}$. Las columnas forman un conjunto C que genera un subespacio de $\mathbb{R}^3$ que según hemos visto antes responde a la ecuación $x + 2y + z = 0$. De aquí concluimos que f es una aplicación lineal inyectiva pero no sobreyectiva, en particular no tiene inversa.

5.4. Bases y dimensión

Bases y dimensión Una *base* B de un espacio vectorial V es un subconjunto que verifica $\langle B\rangle = V$ (*sistema de generadores*) y que es linealmente independiente.

En este curso sólo trataremos el caso de bases con un número finito de elementos. Es conveniente considerar las bases como conjuntos ordenados (es decir, no cambiaremos la ordenación de los vectores que la componen).

Dada una base $B = \{\vec{b}_1, \ldots, \vec{b}_n\}$ cada vector $\vec{v}$ se puede escribir de forma única como combinación lineal $\vec{v} = \lambda_1\vec{b}_1 + \cdots + \lambda_n\vec{b}_n$. Los números $\lambda_1, \ldots, \lambda_n$ se llaman *coordenadas* o *componentes* de $\vec{v}$ en la base B.

Un espacio vectorial V puede tener muchas bases pero todas ellas tienen el mismo número de elementos, llamado *dimensión* que se indica con $\dim V$.

En K^n se llama *base canónica* a $\{\vec{e}_1, \ldots, \vec{e}_n\}$ donde $\vec{e}_i$ es el vector que tiene en el i-ésimo lugar un uno y en el resto ceros. Esta base canónica se puede extender al espacio de matrices $\mathcal{M}_{m\times n}(K)$ considerando $\{\vec{e}_1, \ldots, \vec{e}_{mn}\}$ donde $\vec{e}_i$ es la matriz que tiene en el i-ésimo lugar un uno y en el resto ceros, donde se cuentan los lugares de izquierda a derecha y de arriba a abajo. En el espacio $K_n[t]$ formado por los polinomios de grado menor o igual que n en la variable t, la base llamada canónica es $\{1, t, \ldots, t^n\}$. Con estos ejemplos se tiene $\dim K^n = n$, $\dim \mathcal{M}_{m\times n}(K) = mn$ y $\dim K_n[t] = n + 1$. El espacio vectorial trivial $V = \{\vec{0}\}$ tiene dimensión cero.

En un espacio vectorial de dimensión n siempre n vectores linealmente independientes forman una base. Así por ejemplo $\{1 + x - x^2, 7 + x, 5 - 7x\}$ es una base de $\mathbb{R}_2[x]$ porque claramente estos polinomios son linealmente independientes y sabemos que $\dim \mathbb{R}_2[x] = 3$.

Una aplicación lineal $f : K^n \longrightarrow K^m$ se puede escribir siempre como $f(\vec{x}) = A\vec{x}$ con $A \in \mathcal{M}_{m\times n}(K)$. Las columnas de A son las imágenes de los vectores de la base canónica.

Fijada una base B de un espacio vectorial V de dimensión n, la aplicación que asigna a cada vector de V sus coordenadas en la base B define un isomorfismo (una aplicación lineal biyectiva) de V en K^n. Esto prueba que cualquier espacio vectorial (de dimensión finita) es isomorfo a algún K^n, es decir, es igual salvo renombrar sus vectores. Con ello podemos asignar una matriz a una aplicación lineal entre espacios que no son necesariamente K^n siempre que hayamos fijado bases.

Por ejemplo $f(P) = P + P'$ (con P' la derivada de P) es una aplicación lineal $f : \mathbb{R}_2[x] \longrightarrow \mathbb{R}_2[x]$ y si fijamos la base canónica, cada polinomio $a + bx + cx^2$ estará asociado al vector de $\mathbb{R}^3$ con coordenadas a, b y c. El efecto de la aplicación f se puede traducir a $\mathbb{R}^3$:

$$f(a+bx+cx^2) = (a+b)+(b+2c)x+cx^2 \quad \leftrightarrow \quad \widetilde{f}\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} a+b \\ b+2c \\ c \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} a \\ b \\ c \end{pmatrix}.$$

Esta última matriz 3×3 diremos que es la matriz asociada a f cuando se emplea la base B.

Problemas acerca del núcleo y la imagen pueden ser más naturales o mecánicos en K^n. En el ejemplo anterior es fácil ver que $\mathrm{Nuc}(\widetilde{f}) = \{\vec{0}\}$ y de ello se concluye $\mathrm{Nuc}(f) = \{0\}$. De hecho, la matriz obtenida es invertible y así $\widetilde{f}$ es un isomorfismo (una aplicación $\vec{x} \mapsto A\vec{x}$ define un isomorfismo si y sólo si A es cuadrada e invertible) y se deduce que f también lo es.

Dada una aplicación lineal $f : V \longrightarrow W$ siempre se cumple la relación:

$$\dim V = \dim \mathrm{Nuc}(f) + \dim \mathrm{Im}(f).$$

En el ejemplo anterior a partir de $\mathrm{Nuc}(f) = \{0\}$ se tiene por esta fórmula $\dim \mathrm{Im}(f) = 3$ sin necesidad de hallar la imagen, porque $\dim \mathbb{R}_2[x] = 3$, que con la inclusión $\mathrm{Im}(f) \subset \mathbb{R}_2[x]$ implica $\mathrm{Im}(f) = \mathbb{R}_2[x]$ (porque los elementos de la base de $\mathrm{Im}(f)$ serán tres vectores linealmente independientes en $\mathbb{R}_2[x]$ y por consiguiente deben generar este espacio).

5.5. Operaciones con espacios vectoriales

Suma e intersección de subespacios Hay dos operaciones naturales que se pueden aplicar sobre los subespacios para producir otros nuevos.

1. Dados dos subespacios V y W de un mismo espacio vectorial, su intersección $V \cap W$ también es un subespacio.

2. Dados V y W como antes, se define el *subespacio suma* como

$$V + W = \{\vec{v} + \vec{w} \ : \ \vec{v} \in V, \ \vec{w} \in W\}.$$

Para calcular la intersección de dos subespacios de K^n dados por ecuaciones, simplemente se añaden las ecuaciones del primero a las del segundo.

Para calcular la suma de dos subespacios dados por generadores, simplemente se añaden los generadores del primero a los del segundo.

Por ejemplo, digamos que queremos calcular la intersección y la suma de los subespacios de $\mathbb{R}^4$, V y W, que tienen bases $B_V = \{\vec{u}_1, \vec{u}_2\}$ y $B_W = \{\vec{u}_3, \vec{u}_4\}$ con

$$\vec{u}_1 = \begin{pmatrix} 1 \\ -2 \\ 1 \\ 0 \end{pmatrix}, \quad \vec{u}_2 = \begin{pmatrix} 0 \\ -1 \\ 0 \\ 1 \end{pmatrix}, \quad \vec{u}_3 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 2 \end{pmatrix}, \quad \vec{u}_4 = \begin{pmatrix} 2 \\ -1 \\ 1 \\ 1 \end{pmatrix}.$$

La suma es el subespacio de $\mathbb{R}^4$ generado por $\{\vec{u}_1, \vec{u}_2, \vec{u}_3, \vec{u}_4\}$. Un cálculo, que no se incluye, prueba que no es linealmente independiente pero $\{\vec{u}_1, \vec{u}_2, \vec{u}_3\}$ sí lo es. Así pues este último conjunto es una base de $V + W$ y se tiene $\dim(V + W) = 3$.

Para hallar la intersección hallamos las ecuaciones de V y W por el procedimiento ya estudiado, obteniéndose

$$V = \{\vec{x} \in \mathbb{R}^4 \,:\, \begin{pmatrix} -1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{pmatrix} \vec{x} = \vec{0}\}, \qquad W = \{\vec{x} \in \mathbb{R}^4 \,:\, \begin{pmatrix} 0 & 1 & 1 & 0 \\ -2 & -3 & 0 & 1 \end{pmatrix} \vec{x} = \vec{0}\}.$$

El subespacio $V \cap W$ se obtiene poniendo estas cuatro ecuaciones juntas. Resolviendo el sistema cuatro por cuatro resultante, se sigue que hay un sólo vector que genera el subespacio, por tanto $\dim(V \cap W) = 1$.

Se verifica siempre la *fórmula de Grassman*:

$$\dim V + \dim W = \dim(V \cap W) + \dim(V + W).$$

En el ejemplo anterior, a partir de $\dim V = 2$, $\dim W = 2$, $\dim(V \cap W) = 1$ podríamos haber deducido que $\dim(V+W) = 3$ y por tanto que en el conjunto $\{\vec{u}_1, \vec{u}_2, \vec{u}_3, \vec{u}_4\}$ sobraba un vector para llegar a una base.

Si $E = V + W$ y $V \cap W = \{\vec{0}\}$, se dice que E es *suma directa* de V y W y se suele escribir $E = V \oplus W$. En ese caso cada vector $\vec{e} \in E$ se escribe de forma única como $\vec{e} = \vec{v} + \vec{w}$ con $\vec{v} \in V$, $\vec{w} \in W$.

5.6. Rango y cambio de base

Rango Dada una matriz $A \in \mathcal{M}_{m\times n}(K)$ se define su *rango*, y se denota con $\mathrm{rg}(A)$, como el número máximo de columnas de A linealmente independientes.

Según hemos visto, si $f : K^n \longrightarrow K^m$ viene dada por $f(\vec{x}) = A\vec{x}$ entonces $\mathrm{rg}(A) = \dim \mathrm{Im}(f)$.

El rango se puede calcular aplicando el algoritmo de reducción de Gauss sobre la matriz. El número de columnas pivotes (que es el número de escalones) coincide con el rango y además las columnas de A en los lugares de las columnas pivotes dan una base del subespacio de K^m generado por las columnas. Esto es útil para calcular bases de $\mathrm{Im}(f)$ y en general de cualquier subespacio de K^m expresado mediante generadores.

Por ejemplo, sea la matriz de $\mathcal{M}_{4\times5}(\mathbb{R})$

$$A = \begin{pmatrix} 1 & 3 & -5 & 1 & 5 \\ -2 & -5 & 8 & 0 & -17 \\ 3 & 11 & -19 & 7 & 1 \\ 1 & 7 & -13 & 5 & -3 \end{pmatrix}$$

y llamamemos $\vec{v}_1$, $\vec{v}_2$, $\vec{v}_3$, $\vec{v}_4$, $\vec{v}_5$ a sus columnas, que generan $\mathrm{Im}(f)$ para f como antes. Con unos cálculos la reducción de Gauss lleva a

$$\begin{pmatrix} 1 & 3 & -5 & 1 & 5 \\ 0 & 1 & -2 & 2 & -7 \\ 0 & 0 & 0 & -4 & 20 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

El rango es tres y como las columnas pivote son la primera, la segunda y la cuarta, se tiene que $\{\vec{v}_1, \vec{v}_2, \vec{v}_4\}$ es una base de $\mathrm{Im}(f)$.

Nótese entonces que para cualquier aplicación lineal $f : K^n \longrightarrow K^m$ (y todas entre espacios vectoriales de dimensión finita se podían reducir a este caso), los cálculos de la reducción de Gauss para hallar el núcleo se pueden aprovechar para hallar una base de la imagen.

Teorema de Rouché–Frobenius Consideremos un sistema de ecuaciones lineales $A\vec{x} = \vec{b}$ con $A \in \mathcal{M}_{m\times n}(K)$ e introduzcamos la *matriz aumentada* $A^+ = (A|\vec{b})$ obtenida al añadir $\vec{b}$ a A como última columna.

El teorema de Rouché–Frobenius[2] afirma que el sistema es

1. *Compatible determinado* (solución única) si y sólo si $\mathrm{rg}(A) = \mathrm{rg}(A^+) = n$.
2. *Compatible indeterminado* (tiene solución pero no es única) si y sólo si $\mathrm{rg}(A) = \mathrm{rg}(A^+) \neq n$.
3. *Incompatible* (no tiene solución) si y sólo si $\mathrm{rg}(A) \neq \mathrm{rg}(A^+)$.

En ejemplos numéricos el mismo cálculo que lleva al rango también lleva a la solución o a que no existe, por tanto su interés práctico en estos casos es limitado.

Cambio de base para matrices de aplicaciones lineales Si en V (espacio vectorial de dimensión finita) se fija una base B entonces cada aplicación lineal $f : V \longrightarrow V$ tiene asociada una matriz cuadrada A. Al cambiar la

[2]Este nombre prácticamente sólo se usa en los textos escritos por autores hispanos, parece ser que por influencia del matemático hispanoargentino J. Rey Pastor (1888–1962).

base B por otra base B', la matriz cambia a una nueva matriz A' mediante la fórmula:

$$A' = M_{BB'}\, A\, M_{B'B}$$

donde $M_{B'B}$ es la llamada *matriz de cambio de base* de B' a B. Sus columnas son las coordenadas en la base B de los elementos de la base B'. Se cumple $M_{BB'} = M_{B'B}^{-1}$.

Por ejemplo, si tenemos la aplicación lineal

$$f : \mathbb{R}^2 \longrightarrow \mathbb{R}^2, \qquad f(\vec{x}) = A\vec{x}, \quad \text{con } A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix},$$

entonces A es la matriz de f en la base canónica B. Si queremos averiguar la matriz A' de f en la base $B' = \left\{ \binom{2}{1}, \binom{3}{2} \right\}$ (éstas son las coordenadas con respecto a la base canónica) se tiene según la fórmula

$$M_{B'B} = \begin{pmatrix} 2 & 3 \\ 1 & 2 \end{pmatrix} \quad \Rightarrow \quad A' = \begin{pmatrix} 2 & 3 \\ 1 & 2 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 2 & 3 \\ 1 & 2 \end{pmatrix} = \begin{pmatrix} -22 & -37 \\ 16 & 27 \end{pmatrix}.$$

Si llamamos $\vec{v}_1$ y $\vec{v}_2$ a los vectores de B', una forma alternativa de proceder es calcular $f(\vec{v}_1)$ y $f(\vec{v}_2)$ es expresarlos como combinación lineal de $\vec{v}_1$ y $\vec{v}_2$. Las coordenadas serán las columnas de A', así $f(\vec{v}_1) = \binom{4}{10} = -22\vec{v}_1 + 16\vec{v}_2$, $f(\vec{v}_2) = \binom{7}{17} = -37\vec{v}_1 + 27\vec{v}_2$.

5.7. Determinantes y sus aplicaciones

Determinantes A cada matriz cuadrada $A \in \mathcal{M}_{n\times n}(K)$ se le asigna un elemento de K llamado *determinante* que se denota con $|A|$ o $\det(A)$.

La definición del determinante de A es recursiva. Si A es una matriz 1×1 se define como el único elemento de A y en dimensiones superiores como

$$|A| = a_{11}|A_{11}| - a_{21}|A_{21}| + a_{31}|A_{31}| - \cdots + (-1)^{n+1} a_{n1}|A_{n1}|$$

donde A_{ij} es la matriz obtenida a partir de A tachando la fila i y la columna j.

A partir de esta definición es fácil deducir algunas fórmulas comunes como los determinantes 2×2 o el determinante de una matriz triangular

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc, \qquad \begin{vmatrix} a_{11} & * & * & \dots & * \\ 0 & a_{22} & * & \dots & * \\ 0 & 0 & a_{33} & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn} \end{vmatrix} = a_{11}a_{22}a_{33}\cdots a_{nn}.$$

Desde el punto de vista computacional la fórmula que define el determinante es muy costosa para matrices generales de tamaño moderadamente grande.

Algunas propiedades de los determinantes son:
1. $|A| = |A^t|$. 2. $|AB| = |A|\,|B|$.
3. Al intercambiar dos columnas (o dos filas) el determinante cambia de signo.

Regla de Cramer Los determinantes están ligados a la resolución de sistemas de ecuaciones lineales $n \times n$. La relación más conocida (aunque muy poco práctica) es la llamada *regla de Cramer*:

Sea el sistema $A\vec{x} = \vec{b}$ con $A \in \mathcal{M}_{n\times n}(K)$ y $|A| \neq 0$, entonces la solución (única) viene dada por la fórmula

$$x_1 = \frac{|A_1|}{|A|}, \quad x_2 = \frac{|A_2|}{|A|}, \quad x_3 = \frac{|A_3|}{|A|}, \quad \ldots \quad x_n = \frac{|A_n|}{|A|},$$

donde A_i es la matriz A sustituyendo la columna i por $\vec{b}$.

La relación entre resolución de sistemas y determinantes queda clara sabiendo que frente a los tres procesos de la reducción de Gauss, un determinante

1. queda invariante al sumar a una fila un múltiplo de otra.
2. se multiplica por α al multiplicar una fila por α.
3. cambia de signo al intercambiar dos filas.

De ello se podrían deducir todas las propiedades de los determinantes y permite dar un algoritmo basado en el de Gauss para calcularlos. Por ejemplo:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 2 & 2 & 3 \\ 4 & 2 & 0 & 3 \\ 2 & 2 & 1 & 1 \end{pmatrix} \quad \Rightarrow \quad |A| = \begin{vmatrix} 1 & 2 & 3 & 4 \\ 0 & 2 & 2 & 3 \\ 0 & -6 & -12 & -13 \\ 0 & -2 & -5 & -7 \end{vmatrix} = \begin{vmatrix} 1 & 2 & 3 & 4 \\ 0 & 2 & 2 & 3 \\ 0 & 0 & -6 & -4 \\ 0 & 0 & -3 & -4 \end{vmatrix},$$

aunque no es estrictamente necesario intercambiamos ahora las dos últimas columnas para hacer los cálculos con mayor comodidad:

$$|A| = -\begin{vmatrix} 1 & 2 & 3 & 4 \\ 0 & 2 & 2 & 3 \\ 0 & 0 & -3 & -4 \\ 0 & 0 & -6 & -4 \end{vmatrix} = -\begin{vmatrix} 1 & 2 & 3 & 4 \\ 0 & 2 & 2 & 3 \\ 0 & 0 & -3 & -4 \\ 0 & 0 & 0 & 4 \end{vmatrix} = -1 \cdot 2 \cdot (-3) \cdot 4 = 24.$$

La relación con la reducción de Gauss también sirve para probar que las columnas de una matriz cuadrada A son linealmente independientes si y sólo si $|A| \neq 0$ y esta última condición equivale a que A sea invertible.

Se pueden emplear determinantes para hallar la matriz inversa. Si $|A| \neq 0$, se tiene

$$A^{-1} = \frac{1}{|A|} C^t$$

donde C es la matriz de cofactores, es decir $c_{ij} = (-1)^{i+j}|A_{ij}|$. Por ejemplo, si $A = \left(\begin{smallmatrix} 3 & 2 \\ 1 & 1 \end{smallmatrix}\right)$ se tiene $|A_{11}| = 1$, $|A_{12}| = 1$, $|A_{21}| = 2$, $|A_{22}| = 3$, y la matriz de cofactores resulta $C = \left(\begin{smallmatrix} 1 & -1 \\ -2 & 3 \end{smallmatrix}\right)$ y la inversa es $A^{-1} = \left(\begin{smallmatrix} 1 & -2 \\ -1 & 3 \end{smallmatrix}\right)$.

5.8. Diagonalización de aplicaciones lineales

Valores y vectores propios Dado un endomorfismo, esto es, una aplicación lineal $f : V \longrightarrow V$, se dice que un vector $\vec{v} \in V$ no nulo es un *autovector* o *vector propio* si verifica $f(\vec{v}) = \lambda\vec{v}$ para algún $\lambda \in K$. Este número λ se llama *autovalor* o *valor propio*.

Cualquier múltiplo no nulo de un vector propio es también vector propio.

Si se tiene una aplicación $f : K^n \longrightarrow K^n$ dada por $f(\vec{x}) = A\vec{x}$ con $A \in \mathcal{M}_{n\times n}(K)$, entonces el sistema $A\vec{x} = \lambda\vec{x}$ debe tener soluciones no triviales (que son autovectores) cuando λ sea autovalor. Esto prueba que los valores propios son las raíces de la ecuación algebraica $|A-\lambda I| = 0$, llamada *ecuación característica*. Una vez hallados los autovalores, los autovectores se obtienen resolviendo el sistema $(A - \lambda I)\vec{x} = \vec{0}$.

Todas las aplicaciones lineales $f : V \longrightarrow V$ tienen una matriz, una vez fijada una base, y se corresponden con el caso anterior. A veces, con un poco de falta de rigor, se habla de los valores y vectores propios de una matrices para referirse a los de la aplicación del tipo anterior con A especificada.

Calculemos todos los autovalores y autovectores de $f : \mathbb{R}^3 \longrightarrow \mathbb{R}^3$ dada por $f(\vec{x}) = A\vec{x}$ donde

$$A = \begin{pmatrix} 4 & -1 & 6 \\ 2 & 1 & 6 \\ 2 & -1 & 8 \end{pmatrix}.$$

Unos cálculos, que se pueden simplificar con las propiedades de los determinantes, prueban que la ecuación característica es

$$|A - \lambda I| = -\lambda^3 + 13\lambda^2 - 40\lambda + 36 = (9 - \lambda)(\lambda - 2)^2 = 0.$$

Por tanto hay dos autovalores: $\lambda_1 = 9$ y $\lambda_2 = 2$. Resolviendo los sistemas lineales $(A - 9I)\vec{x} = \vec{0}$ y $(A - 2I)\vec{x} = \vec{0}$ se tiene que los autovectores con autovalores λ_1 y λ_2 son, respectivamente, los vectores no nulos de los subespacios:

$$V_1 = \left\langle \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\rangle \qquad \text{y} \qquad V_2 = \left\langle \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 6 \\ 1 \end{pmatrix} \right\rangle.$$

Un teorema asegura que los autovectores correspondientes a diferentes autovalores son siempre linealmente independientes.

Diagonalización Se dice que una aplicación lineal $f : V \longrightarrow V$ es *diagonalizable* si existe una base B de V en la que su matriz es diagonal.

Una aplicación lineal es diagonalizable si y sólo si existe una base formada por autovectores (y ésta es una base B válida en la definición anterior).

Con el abuso de notación obvio muchas veces se dice que una matriz cuadrada $A \in \mathcal{M}_{n\times n}(K)$ es diagonalizable para indicar que la aplicación $f(\vec{x}) = A\vec{x}$, definida de K^n en K^n, lo es.

Por ejemplo, la matriz A indicada anteriormente es diagonalizable porque

$$\left\{ \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 6 \\ 1 \end{pmatrix} \right\}$$

es una base de $\mathbb{R}^3$ formada por autovectores.

No todas las aplicaciones lineales son diagonalizables. El caso más obvio ocurre cuando no podemos resolver la ecuación característica en el cuerpo K en el que trabajamos. Por ejemplo, la aplicación lineal $f : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$ dada por $f\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) = \left(\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right)\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right)$ lleva a la ecuación característica $\lambda^2 + 1 = 0$ que no tiene solución en los reales. Podríamos extender la aplicación a $F : \mathbb{C}^2 \longrightarrow \mathbb{C}^2$ con la misma fórmula y tendríamos los autovalores $\lambda_1 = i$, $\lambda_2 = -i$ (con $i = \sqrt{-1}$) y sí sería diagonalizable en la base de vectores propios complejos $\left\{\left(\begin{smallmatrix} i \\ 1 \end{smallmatrix}\right) \text{ y } \left(\begin{smallmatrix} -i \\ 1 \end{smallmatrix}\right)\right\}$. Incluso con este tipo de extensiones a los complejos no todas las aplicaciones lineales de $\mathbb{R}^n$ en $\mathbb{R}^n$ son diagonalizables. Por ejemplo, $f : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$ dada por $f\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right)$ tiene sólo el autovalor $\lambda_1 = 2$ y el

subespacio formado por los autovectores (y el $\vec{0}$) tiene dimensión uno, por tanto no hay suficientes autovectores para formar una base.

En el caso diagonalizable, la fórmula de cambio de base relaciona la matriz original con la matriz diagonal a través de las coordenadas de los vectores de la base en que estamos diagonalizando. En general la fórmula responde al esquema $D = C^{-1}AC$.

En el ejemplo que hemos venido manejando, se tendría

$$\begin{pmatrix} 9 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 2 & 6 \\ 1 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 4 & -1 & 6 \\ 2 & 1 & 6 \\ 2 & -1 & 8 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 \\ 1 & 2 & 6 \\ 1 & 0 & 1 \end{pmatrix}.$$

Es importante el orden relativo en que se ordenan los autovalores y autovectores, así el primer elemento de la matriz diagonal, $d_{11} = 9$, es un autovalor que corresponde al autovector indicado por la primera columna en la matriz de cambio de base.

Aplicaciones Hay varias aplicaciones de la diagonalización a temas fuera de la propia álgebra lineal. Revisamos dos de ellas a través de sendos ejemplos:

1) Resolución de sistemas de ecuaciones diferenciales lineales con coeficientes constantes.

Deseamos hallar las funciones regulares $x = x(t)$ e $y = y(t)$ que satisfacen la relación

$\begin{cases} x' = x - 2y \\ y' = x + 4y \end{cases}$ que en forma matricial es $\vec{f}\,' = A\vec{f}$ con $\vec{f} = \begin{pmatrix} x \\ y \end{pmatrix}$ y $A = \begin{pmatrix} 1 & -2 \\ 1 & 4 \end{pmatrix}$.

Diagonalizando se tiene

$$D = C^{-1}AC \qquad \text{donde } D = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} \text{ y } C = \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}.$$

Así pues $\vec{f}\,' = CDC^{-1}\vec{f}$. Escribiendo $\vec{g} = C^{-1}\vec{f}$ se transforma en $\vec{g}\,' = D\vec{g}$ que es muy fácil de resolver como $\vec{g} = \begin{pmatrix} K_1 e^{2t} \\ K_2 e^{3t} \end{pmatrix}$ con K_1, K_2 constantes arbitrarias. La solución del sistema inicial será $\vec{f} = C\vec{g}$.

2) Resolución de ecuaciones de recurrencia.

Supongamos que queremos hallar todas las sucesiones $\{a_n\}_{n=0}^{\infty}$, $\{b_n\}_{n=0}^{\infty}$ que satisfacen

$\begin{cases} a_{n+1} = a_n - 2b_n \\ b_{n+1} = a_n + 4b_n \end{cases}$ o equivalentemente $\vec{s}_{n+1} = A\vec{s}_n$ con $\vec{s}_n = \begin{pmatrix} a_n \\ b_n \end{pmatrix}$, $A = \begin{pmatrix} 1 & -2 \\ 1 & 4 \end{pmatrix}$

que en cierta manera es la variante discreta del problema anterior (las derivadas vienen de incrementos infinitesimales). Iterando obtenemos que la solución es $\vec{s}_n = A^n \vec{s}_0$ y el problema consiste en hallar una fórmula para la potencia n-ésima de una matriz. Diagonalizando como antes $A = CDC^{-1}$ implica $A^n = CD^nC^{-1}$ y se obtiene

$$\vec{s}_n = \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 2^n & 0 \\ 0 & 3^n \end{pmatrix} \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}^{-1} \vec{s}_0.$$

5.9. Espacios vectoriales euclídeos

Producto escalar En $\mathbb{R}^n$ el producto escalar usual se define como $\vec{x} \cdot \vec{y} = x_1y_1 + \cdots + x_ny_n$ donde x_i e y_i son las coordenadas de $\vec{x}$ e $\vec{y}$ (en la base canónica).

En general se dice que una función $f : V \times V \longrightarrow \mathbb{R}$ define un *producto escalar* en V si satisface las propiedades siguientes

1. $f(\vec{x}, \vec{x}) \geq 0$ y $f(\vec{x}, \vec{x}) = 0 \Leftrightarrow \vec{x} = \vec{0}$
2. $f(\vec{x}, \vec{y}) = f(\vec{y}, \vec{x})$
3. f es lineal en cada variable, esto es,

$$f(\lambda_1\vec{x}_1 + \lambda_2\vec{x}_2, \vec{y}) = \lambda_1 f(\vec{x}_1, \vec{y}) + \lambda_2 f(\vec{x}_2, \vec{y})$$
$$f(\vec{x}, \lambda_1\vec{y}_1 + \lambda_2\vec{y}_2) = \lambda_1 f(\vec{x}, \vec{y}_1) + \lambda_2 f(\vec{x}, \vec{y}_2)$$

Los espacios vectoriales en los que se ha definido un producto escalar se dice que son *espacios vectoriales euclídeos*. Las notaciones más empleadas para indicar productos escalares son $f(\vec{x}, \vec{y}) = \vec{x} \cdot \vec{y}$ y $f(\vec{x}, \vec{y}) = \langle \vec{x}, \vec{y} \rangle$. También a veces se emplea $(\vec{x}, \vec{y})$.

Por ejemplo en $\mathbb{R}_n[x]$ (los polinomios reales de grado $\leq n$) $\langle P, Q \rangle = \int_{-1}^{1} PQ$ define un producto escalar. En $\mathbb{R}^n$ existen productos escalares diferentes del usual. Por ejemplo

$$\vec{x} \cdot \vec{y} = x_1y_1 + 2x_2y_2 + x_3y_3 + x_1y_2 + x_2y_1$$

define un producto escalar en $\mathbb{R}^3$. Para comprobar la primera propiedad es conveniente escribir $\vec{x} \cdot \vec{x} = (x_1 + x_2)^2 + x_2^2 + x_3^2$. Más adelante veremos un algoritmo para decidir esta la primera propiedad.

Dado un vector $\vec{x}$ de un espacio vectorial euclídeo, se define su *norma* como $\|\vec{x}\| = \sqrt{\vec{x}\cdot\vec{x}}$ y se dice que el vector es *unitario* si $\|\vec{x}\| = 1$. Dos vectores son *ortogonales* si su producto escalar es nulo. Al dividir un vector no nulo por su norma se obtiene siempre un vector unitario. A este proceso se le llama *normalización*.

Cualquier producto escalar satisface siempre las desigualdades de Cauchy-Schwarz y de Minkowski, dadas respectivamente, por

$$\vec{x}\cdot\vec{y} \leq \|\vec{x}\|\|\vec{y}\| \qquad \text{y} \qquad \|\vec{x}+\vec{y}\| \leq \|\vec{x}\| + \|\vec{y}\|.$$

Con un producto escalar se pueden definir distancias y ángulos (que pueden no coincidir con los medidos con el producto escalar usual): $d(\vec{x},\vec{y}) = \|\vec{x}-\vec{y}\|$, $\cos\alpha = \vec{x}\cdot\vec{y}/(\|\vec{x}\|\|\vec{y}\|)$.

Formas bilineales Una *forma bilineal* es una función $f : V \times V \longrightarrow K$ que es lineal en cada variable. En este curso $K = \mathbb{R}$ y una forma bilineal será simplemente una función que cumple la tercera propiedad del producto escalar.

Se dice que una forma bilineal es *simétrica* si $f(\vec{x},\vec{y}) = f(\vec{y},\vec{x})$ y que es *definida positiva* si $f(\vec{x},\vec{x}) > 0$ para todo $\vec{x} \neq \vec{0}$. Con estas definiciones se tiene que un producto escalar es una forma bilineal simétrica y definida positiva.

No todas las formas bilineales son productos escalares. Por ejemplo, en $\mathbb{R}^2$ la función f que asigna a cada par de vectores el determinante de sus coordenadas puestas en columna, es bilineal pero no un producto escalar porque $f(\vec{x},\vec{x}) = 0$ para todo $\vec{x}$.

Fijada una base $B = \{\vec{e}_1,\ldots,\vec{e}_n\}$ a cada forma bilineal f se le asigna la matriz A que tiene como elemento a_{ij} a $f(\vec{e}_i,\vec{e}_j)$.

Por ejemplo, al producto escalar definido antes en el espacio de polinomios en el caso $n = 2$ con la base canónica $\{1, x, x^2\}$ le correspondería la matriz:

$$A = \begin{pmatrix} \int_{-1}^{1} 1\cdot 1\,dx & \int_{-1}^{1} 1\cdot x\,dx & \int_{-1}^{1} 1\cdot x^2\,dx \\ \int_{-1}^{1} x\cdot 1\,dx & \int_{-1}^{1} x\cdot x\,dx & \int_{-1}^{1} x\cdot x^2\,dx \\ \int_{-1}^{1} x^2\cdot 1\,dx & \int_{-1}^{1} x^2\cdot x\,dx & \int_{-1}^{1} x^2\cdot x^2\,dx \end{pmatrix} = \begin{pmatrix} 2 & 0 & 2/3 \\ 0 & 2/3 & 0 \\ 2/3 & 0 & 2/5 \end{pmatrix}.$$

Al emplear otra base $B' = \{\vec{e}\,'_1,\ldots,\vec{e}\,'_n\}$, la matriz A de una forma bilineal f cambiará a una nueva matriz A' que se puede calcular o bien directamente con la definición $a'_{ij} = f(\vec{e}\,'_i,\vec{e}\,'_j)$ o bien con la fórmula de cambio de base

para formas bilineales $A' = C^t AC$ donde C es la matriz de cambio de base de B' a B.

Por ejemplo, empleando la base $B' = \{1, x, 3x^2 - 1\}$ la matriz correspondiente al producto escalar $\langle P, Q\rangle = \int_{-1}^{1} PQ$ sería ahora rehaciendo los cálculos

$$A' = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2/3 & 0 \\ 0 & 0 & 8/5 \end{pmatrix},$$

Reconocemos que los vectores de B' son ortogonales porque la matriz es diagonal ($f(\vec{e}\,'_i, \vec{e}\,'_j) = 0$ si $i \neq j$). Según la fórmula de cambio de base anterior, A' coincide con

$$C^t AC = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 3 \end{pmatrix}^t \begin{pmatrix} 2 & 0 & 2/3 \\ 0 & 2/3 & 0 \\ 2/3 & 0 & 2/5 \end{pmatrix} \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

donde C se ha construido con las coordenadas de los elementos de B' (respecto de la base B).

5.10. Vectores y proyecciones ortogonales

Ortogonalización Dada una base ortogonal de un espacio vectorial euclídeo es fácil ortonormalizarla sin más que normalizar cada uno de sus vectores (es decir, dividiéndolos por su norma).

Si $\{\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_n\}$ es una base de un espacio vectorial euclídeo V el *proceso de Gram-Schmidt* produce una base ortogonal $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\}$ de V. Este proceso es un algoritmo definido por

$$\begin{cases} \vec{x}_1 = \vec{b}_1 \\ \vec{x}_i = \vec{b}_i - \displaystyle\sum_{j=1}^{i-1} \frac{\vec{b}_i \cdot \vec{x}_j}{\|\vec{x}_j\|^2}\, \vec{x}_j, \qquad i = 2, 3, \ldots, n. \end{cases}$$

Se puede interpretar el algoritmo diciendo que a cada $\vec{b}_i$ se le resta su proyección ortogonal (véase el siguiente apartado) sobre $\langle\{\vec{x}_1, \ldots, \vec{x}_{i-1}\}\rangle$.

Por ejemplo, la base $\{1, x, x^2\}$ de $\mathbb{R}_2[x]$ no es ortonormal con el producto escalar $\langle P, Q\rangle = \int_{-1}^{1} PQ$. Con el proceso de Gram-Schmidt conseguimos una base ortogonal $\{P_1, P_2, P_3\}$ tomando $P_1 = 1$ y

$$P_2 = x - \frac{\int_{-1}^{1} 1x\, dx}{\int_{-1}^{1} 1^2\, dx} 1 = x, \quad P_3 = x^2 - \frac{\int_{-1}^{1} 1x^2\, dx}{\int_{-1}^{1} 1^2\, dx} 1 - \frac{\int_{-1}^{1} x^3\, dx}{\int_{-1}^{1} x^2\, dx} x = x^2 - \frac{1}{3}.$$

Para ortonormalizarla se divide por la norma obteniéndose

$$\{\frac{1}{\sqrt{2}}, \sqrt{\frac{3}{2}}\, x, \sqrt{\frac{45}{8}}(x^2 - \frac{1}{3})\}.$$

De esta forma se concluye que todo espacio vectorial euclídeo (de dimensión finita) tiene bases ortonormales.

Proyección ortogonal Dado un subespacio W de un espacio vectorial euclídeo V se define el *complemento ortogonal* de W (en V) como el subespacio

$$W^\perp = \{\vec{v} \in V \ : \ \vec{v} \cdot \vec{w} = 0, \quad \forall \vec{w} \in W\}.$$

Se prueba que $V = W + W^\perp$ y es fácil ver que $W \cap W^\perp = \{\vec{0}\}$ de modo que V es suma directa de W y $W^\perp$. En consecuencia cada vector $\vec{v} \in V$ se descompone de manera única como $\vec{v} = \vec{w} + \vec{u}$ con $\vec{w} \in W$ y $\vec{u} \in W^\perp$. Se dice que $\vec{w}$ es la *proyección ortogonal* de $\vec{v}$ en W y se escribe $\vec{w} = \mathrm{Pr}_W(\vec{v})$. De la propiedad $(W^\perp)^\perp = W$ se sigue que, con la notación anterior, $\vec{u} = \mathrm{Pr}_{W^\perp}(\vec{v})$. En consecuencia

$$\vec{v} = \mathrm{Pr}_W(\vec{v}) + \mathrm{Pr}_{W^\perp}(\vec{v}).$$

Si $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\}$ es una base ortogonal de W entonces la proyección ortogonal responde a la fórmula:

$$\mathrm{Pr}_W(\vec{v}) = \frac{\vec{v} \cdot \vec{x}_1}{\|\vec{x}_1\|^2}\, \vec{x}_1 + \frac{\vec{v} \cdot \vec{x}_2}{\|\vec{x}_2\|^2}\, \vec{x}_2 + \cdots + \frac{\vec{v} \cdot \vec{x}_n}{\|\vec{x}_n\|^2}\, \vec{x}_n.$$

Cuando $W^\perp$ es un espacio particularmente sencillo puede ser más fácil hallar primero $\mathrm{Pr}_{W^\perp}(\vec{v})$ y de ahí despejar $\mathrm{Pr}_W(\vec{v})$.

Por ejemplo, para hallar la proyección ortogonal de $\vec{v} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$ sobre el subespacio de $\mathbb{R}^4$ (con el producto escalar usual) $W = \{\vec{x} \in \mathbb{R}^4 \ : \ x_2 - x_3 + 2x_4 = 0\}$, podemos proceder rápidamente notando que $W = \{\vec{x} \in \mathbb{R}^4 \ : \ \vec{n} \cdot \vec{x} = 0\}$ con $\vec{n} = \begin{pmatrix} 0 \\ 1 \\ -1 \\ 2 \end{pmatrix}$ y por ello $\vec{n}$ genera $W^\perp$. Entonces

$$\mathrm{Pr}_W(\vec{v}) = \vec{v} - \mathrm{Pr}_{W^\perp}(\vec{v}) = \vec{v} - \frac{\vec{v} \cdot \vec{n}}{\|\vec{n}\|^2}\, \vec{n} = \begin{pmatrix} 1 \\ 2/3 \\ 4/3 \\ 1/3 \end{pmatrix}.$$

Por otro lado, la forma general de proceder sería extraer primero una base ortogonal de W. En este caso el algoritmo de Gauss aplicado de la forma habitual produce una base B que se ortogonaliza utilizando el proceso de Gram-Schmidt

$$B = \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -2 \\ 0 \\ 1 \end{pmatrix} \right\} \overset{\text{Gram-Schmidt}}{\longrightarrow} B' = \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \\ 1 \\ 1 \end{pmatrix} \right\}.$$

Con esta nueva base ya se puede aplicar la fórmula de la proyección ortogonal

$$\mathrm{Pr}_W(\vec{v}) = \frac{1}{1} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \frac{2}{2} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} + \frac{1}{3} \begin{pmatrix} 0 \\ -1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2/3 \\ 4/3 \\ 1/3 \end{pmatrix}$$

que coincide con el resultado obtenido antes.

5.11. Aplicaciones ortogonales y aplicaciones autoadjuntas

Aplicaciones ortogonales y autoadjuntas Consideremos un espacio vectorial euclídeo V. Se dice que una aplicación lineal $f : V \longrightarrow V$ es una *aplicación ortogonal* si preserva el producto escalar, es decir, si

$$\langle f(\vec{x}), f(\vec{y}) \rangle = \langle \vec{x}, \vec{y} \rangle \qquad \text{para todo } \vec{x}, \vec{y} \in V.$$

Una matriz cuadrada $A \in \mathcal{M}_{n\times n}(\mathbb{R})$ se dice que es una *matriz ortogonal* si verifica $A^t A = I$, donde I es la matriz identidad. De esta definición se sigue fácilmente $|A| = 1$, $A^{-1} = A^t$ y que las columnas de A son ortonormales.

La relación entre ambas definiciones es que si consideramos $\mathbb{R}^n$ con el producto escalar usual y $f : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ dada por $f(\vec{x}) = A\vec{x}$ entonces f es una aplicación ortogonal si y sólo si A es una matriz ortogonal.

Por ejemplo, la aplicación $f\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) = \left(\begin{smallmatrix} (3x-4y)/5 \\ (4x+3y)/5 \end{smallmatrix}\right)$ de $\mathbb{R}^2$ en $\mathbb{R}^2$ (con el producto escalar usual) es ortogonal porque

$$f\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) = \left(\begin{smallmatrix} 3/5 & -4/5 \\ 4/5 & 3/5 \end{smallmatrix}\right)\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) \qquad \text{y se verifica} \qquad \left(\begin{smallmatrix} 3/5 & 4/5 \\ -4/5 & 3/5 \end{smallmatrix}\right)\left(\begin{smallmatrix} 3/5 & -4/5 \\ 4/5 & 3/5 \end{smallmatrix}\right) = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right).$$

Dada una aplicación lineal $f : V \longrightarrow V$ con V un espacio vectorial euclídeo, se dice que $f' : V \longrightarrow V$ es la *aplicación adjunta* de f si

$$\langle f(\vec{x}), \vec{y} \rangle = \langle \vec{x}, f'(\vec{y}) \rangle \qquad \text{para todo } \vec{x}, \vec{y} \in V.$$

Consecuentemente se dice que f es una *aplicación autoadjunta* si $f = f'$.

De nuevo si consideramos $\mathbb{R}^n$ con el producto escalar usual y $f : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ dada por $f(\vec{x}) = A\vec{x}$ estos conceptos tienen una interpretación matricial sencilla: La aplicación adjunta de f es $f'(\vec{x}) = A^t\vec{x}$ y por tanto f es autoadjunta si A es una matriz simétrica ($A = A^t$).

Por ejemplo, la adjunta de la aplicación antes citada es $f\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) = \frac{1}{5}\left(\begin{smallmatrix} 3x+4y \\ -4x+3y \end{smallmatrix}\right)$.

Estos criterios para decidir si f es ortogonal o autoadjunta se extienden a cualquier espacio euclídeo siempre que se usen bases ortonormales. Es decir, dada una aplicación $f : V \longrightarrow V$ cuya matriz en una base ortonormal es A, entonces f es ortogonal si y sólo si $A^tA = I$ (la matriz es ortogonal) y es autoadjunta si y sólo si $A = A^t$ (la matriz es simétrica). En cualquier caso A^t será la matriz de la aplicación adjunta en dicha base.

Teorema espectral Sea V un espacio vectorial euclídeo sobre $\mathbb{R}$. El *teorema espectral* asegura que dada una aplicación autoadjunta $f : V \longrightarrow V$ existe una base ortonormal de V formada por autovectores de f. Dicho de otra forma, cualquier aplicación autoadjunta diagonaliza en una base ortonormal.

En particular, toda matriz simétrica $A \in \mathcal{M}_{n\times n}(\mathbb{R})$ es diagonalizable.

El teorema espectral un resultado teórico, para llevar a efecto la diagonalización hay que emplear los métodos de los temas anteriores.

Por ejemplo, diagonalicemos la matriz

$$A = \begin{pmatrix} 3 & 2 & \sqrt{2} \\ 2 & 3 & \sqrt{2} \\ \sqrt{2} & \sqrt{2} & 2 \end{pmatrix}$$

en una base ortonormal (con el producto escalar usual de $\mathbb{R}^3$).

Unos cálculos (que es mejor abreviar usando las propiedades de los determinantes) prueban que la ecuación característica $|A - \lambda I| = 0$ es $(\lambda - 1)^2(6 - \lambda) = 0$, entonces los valores propios son $\lambda_1 = 1$ y $\lambda_2 = 6$.

Al resolver $(A-I)\vec{x}=\vec{0}$ y $(A-6I)\vec{x}=\vec{0}$ por Gauss se obtiene que los autovectores correspondientes a $\lambda_1=1$ y a $\lambda_2=6$ son los vectores (salvo $\vec{0}$) respectivamente de los subespacios

$$E_1=\left\langle\left\{\begin{pmatrix}-1\\1\\0\end{pmatrix},\begin{pmatrix}-\sqrt{2}/2\\0\\1\end{pmatrix}\right\}\right\rangle \qquad \text{y} \qquad E_2=\left\langle\left\{\begin{pmatrix}\sqrt{2}\\\sqrt{2}\\1\end{pmatrix}\right\}\right\rangle.$$

El teorema espectral asegura que los vectores de E_1 y los de E_2 son ortogonales pero no que los generadores que nosotros elijamos formen una base ortonormal. Si ortogonalizamos los generadores de E_1 con el proceso de Gram-Schmidt, debemos sustituir el segundo por

$$\begin{pmatrix}-\sqrt{2}/2\\0\\1\end{pmatrix}-\frac{\sqrt{2}/2}{2}\begin{pmatrix}-1\\1\\0\end{pmatrix}=\begin{pmatrix}-\sqrt{2}/4\\-\sqrt{2}/4\\1\end{pmatrix}.$$

Con estos tres vectores tenemos una base ortogonal de $\mathbb{R}^3$ formada por vectores propios de A, que ortonormalizada (dividiendo cada vector por su norma) es

$$B=\left\{\begin{pmatrix}-1/\sqrt{2}\\1/\sqrt{2}\\0\end{pmatrix},\begin{pmatrix}-\sqrt{2}/2\sqrt{5}\\-\sqrt{2}/2\sqrt{5}\\2/\sqrt{5}\end{pmatrix},\begin{pmatrix}\sqrt{2}/\sqrt{5}\\\sqrt{2}/\sqrt{5}\\1/\sqrt{5}\end{pmatrix}\right\}.$$

La matriz C de cambio de base de B a la canónica es la que tiene como columnas estos vectores y se cumple

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&6\end{pmatrix}=C^{-1}AC.$$

Nótese que C es una matriz ortogonal (porque sus columnas son vectores ortonormales) y por tanto $C^{-1}=C^t$.

5.12. Diagonalización de formas cuadráticas

Formas cuadráticas Una *forma cuadrática* es una función $Q:V\longrightarrow\mathbb{R}$ tal que $Q(\vec{x})=f(\vec{x},\vec{x})$ donde f es una forma bilineal.

Si $x_1,\ldots,x_n$ son las coordenadas de $\vec{x}$ en una base B, entonces $Q(\vec{x})=\sum_{i,j=1}^n f_{ij}x_ix_j$ donde $\left(f_{ij}\right)_{i,j=1}^n$ es la matriz de f en la base B. Tomando $a_{ij}=(f_{ij}+f_{ji})/2$ tenemos $Q(\vec{x})=\sum_{i,j=1}^n a_{ij}x_ix_j=\vec{v}^{\,t}A\vec{v}$ donde A es una matriz simétrica y $\vec{v}$ es un vector (columna) en $\mathbb{R}^n$ de coordenadas $x_1,\ldots,x_n$.

En definitiva, una forma cuadrática no es otra cosa que un polinomio cuadrático homogéneo en las coordenadas. Además se puede escribir como $\vec{v}^{\,t}A\vec{v}$ con $A=A^t$, la *matriz de la forma cuadrática* en la base elegida.

Se dice que una forma cuadrática Q es *definida positiva* si $Q(\vec{x}) > 0$ para $\vec{x} \neq \vec{0}$, se dice que es *definida negativa* si $Q(\vec{x}) < 0$ para $\vec{x} \neq \vec{0}$ y, suponiendo que su matriz tiene determinante no nulo, se dice que Q es *indefinida* si no es ni definida positiva ni definida negativa.

Si A con $|A| \neq 0$ es la matriz de una forma cuadrática Q, por el teorema espectral A diagonaliza en una base ortonormal. En dicha base Q se expresa como $\lambda_1 x_1^2 + \ldots \lambda_n x_n^2$ donde $\lambda_1, \ldots, \lambda_n$ son los autovalores y Q será definida positiva si todos ellos son positivos y definida negativa si todos son negativos. Un posterior cambio de base $x_i \mapsto x_i'/\sqrt{|\lambda_i|}$ combinado con un reordenamiento de las variables permiten llegar a que Q en cierta base se expresa en coordenadas como $y_1^2 + \cdots + y_k^2 - (y_{l+1}^2 + \cdots + y_{l+k}^2)$. A esta expresión se le llama *forma normal* de Q y al par de números (k, l) *signatura* de Q (si $l = 0$, Q es definida positiva).

Por ejemplo, $Q(\vec{x}) = -x^2 + 8xy - 7y^2$ con $\vec{x} = \left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right)$ tiene como matriz $A = \left(\begin{smallmatrix} -1 & 4 \\ 4 & -7 \end{smallmatrix}\right)$, $Q(\vec{x}) = \vec{x}^{\,t} A \vec{x}$. Los autovalores de A son $\lambda_1 = 1 > 0$ y $\lambda_2 = -9 < 0$ entonces su signatura es $(1, 1)$ y es indefinida. De la misma forma si consideramos el ejemplo en $\mathbb{R}^3$ dado por $Q = x^2 + 5y^2 + 2z^2 + 4yz$ los autovalores son $\lambda_1 = 1 > 0$ (doble) y $\lambda_2 = 6 > 0$ por tanto Q es definida positiva.

El *criterio de Sylvester* es un criterio simple y eficiente en dimensiones bajas para decidir si una forma cuadrática es definida positiva. Lo que establece es:

$$\vec{x}^{\,t} A \vec{x} \text{ definida postiva } \Leftrightarrow \Delta_k = \begin{vmatrix} a_{11} & \ldots & a_{1k} \\ \vdots & & \vdots \\ a_{k1} & \ldots & a_{kk} \end{vmatrix} > 0 \text{ para todo } k.$$

Evidentemente, Q es definida negativa si y sólo si $-Q$ es definida positiva y el criterio se extiendo de la forma obvia.

Por ejemplo, la forma cuadrática en $\mathbb{R}^3$ dada por $x^2 + 11y^2 + 5z^2 - 4xy + 2xz - 2yz$ es definida positiva porque

$$\Delta_1 = |\,1\,| > 0, \qquad \Delta_2 = \begin{vmatrix} 1 & -2 \\ -2 & 11 \end{vmatrix} > 0 \quad \text{y} \quad \Delta_3 = \begin{vmatrix} 1 & -2 & 1 \\ -2 & 11 & -1 \\ 1 & -1 & 5 \end{vmatrix} > 0.$$

Usando el criterio de Sylvester, la forma cuadrática antes mencionada $Q(\vec{x}) = -x^2 + 8xy - 7y^2$, no es definida positiva porque $\Delta_1 < 0$, tampoco es defi-

nida negativa porque cambiando de signo la matriz se tiene $\Delta_2 < 0$, por consiguiente es indefinida, como ya habíamos visto de otro modo.

Un tercer método para comprobar si una forma cuadrática es definida positiva es el *algoritmo de Gauss* que se basa en completar cuadrados para eliminar todos los productos cruzados y conseguir expresar la forma cuadrática como una combinación de cuadrados.

Por ejemplo si $Q = x^2 - y^2 + 4z^2 - 2xy + 6xz - 2yz$, elegimos una variable, digamos x, y queremos eliminar los productos $-2xy+6xz$ para ello se completan cuadrados escribiendo $x^2-2xy+6xz = (x-y+3z)^2-y^2-9z^2+6yz$ (nótese que los coeficientes -1 y 3 son la mitad de -2 y 6). Sustituyendo obtenemos

$$Q = (x - y + 3z)^2 - 2y^2 - 5z^2 + 4yz.$$

Ahora ya no hay productos cruzados con x. Se repite el mismo procedimiento con la variable y para eliminar $4yz$. Lo más cómodo es sacar el el coeficiente -2 factor común antes de completar cuadrados: $-2(y^2 - 2yz) = -2\big((y - z)^2 - z^2\big)$. Sustituyendo se llega a

$$Q = (x - y + 3z)^2 - 2(y - z)^2 - 3z^2.$$

Con esto ya está claro que la forma cuadrática es indefinida. El cambio de base $X = x - y + 3z$, $Y = \sqrt{2}\ (y - z)$, $Z = \sqrt{3}\ z$ lleva a la forma normal $X^2 - Y^2 - Z^2$.

Un caso excepcional en el algoritmo de Gauss es cuando la forma cuadrática sólo tiene productos cruzados y entonces no hay ninguna variable en la que comenzar completando cuadrados. Por ejemplo, $Q = xy + 3xz + 5yz$. En ese caso, antes de comenzar se hace un cambio de base previo que provoque la aparición de algún cuadrado. Típicamente $x = X + Y$, $y = X - Y$ (o con otros nombres de variables) que llevaría la Q anterior a $X^2-Y^2+8Xz-2Yz$ sobre la cual ya se pueden completar cuadrados hasta llegar a $(X + 4z)^2 - (Y + z)^2 - 15z^2$ que prueba que la signatura es $(1, 2)$.