Proyecto #5: Salidas gráficas en Matlab. Interpolación

Víctor Domínguez María Luisa Rapún
http://www.unavarra.es/personal/victor_dominguez/

12 de diciembre de 2005

Capítulo 1

Introducción

Trataremos en este proyecto uno de los aspectos más potentes de Matlab: las salidas gráficas. Aunque es difícil, o al menos extenso, exponer con cierto detalle todos los comandos, su funcionamiento y los diferentes parámetros y opciones que se encuentran a disposición del usuario, sí que es asumible el conocimiento de los aspectos más básicos y que sea el propio usuario, con la ayuda de Matlab o con guías mucho más detalladas, quien profundice en los detalles que necesite.

En la segunda parte estudiaremos el problema de la interpolación polinómica como una buena piedra de toque para testar las salidas gráficas.

Como aspectos relacionados hablaremos someramente de la interpolación por *splines* y las curvas Bezier, que nos dan un ejemplo muy sencillo de Matemáticas aplicadas al diseño gráfico.

Parte I

Matlab: Salidas gráficas en Matlab

Capítulo 2

Dibujos bidimensionales

2.1. El comando `plot`

Ya hemos observado que los comandos de Matlab definen de una forma natural varios niveles de manipulación. Los niveles básicos son fáciles de utilizar pero acceder a detalles más finos exige argumentos opcionales y detalles mucho más finos.

Esta característica se destaca más, si cabe, en los comandos relacionados con las salidas gráficas. Es por ello que, empezando por la instrucción `plot`, recorreremos los diferentes niveles de forma gradual. El nivel superior, que conlleva un control absoluto del dibujo, no lo trataremos porque requeriría un exposición demasiada larga. Lo que aquí expondremos es suficiente en un 99 % de los casos, y en última medida se pueden acceder a todas las características de un dibujo desde la ventana gráfica *a golpe* de ratón.

2.1.1. Primer nivel

El primer comando que trataremos es `plot`[1]. Es una instrucción muy versátil y la más indicada para dibujar gráficas de funciones y curvas en el plano. Su sintaxis básica es

```
>> plot(x,y)
```

dibuja el vector `x` versus el vector `y`. Más en concreto une los puntos `(x(i), y(i))` mediante segmentos. Tomando un número suficiente elevado de puntos trazamos con ello una gráfica *suave*, sin esquinas.

Al ejecutar este comando se abre una ventana, *figure* en el vocabulario de Matlab, con la correspondiente salida gráfica.

Si se teclea

```
>> plot(y)
```

e `y` es real entonces la salida gráfica toma `x=1:n` donde `n` es la longitud de `y`. Si, por otro lado, `z` es un vector de números complejos, dibuja la parte real versus la parte imaginaria. Es decir, es equivalente a `plot(real(z),imag(z))`.

Ejercicio 1 ¿Qué hacen las siguientes líneas de código?

[1] que ha hemos tratado en el tema anterior.

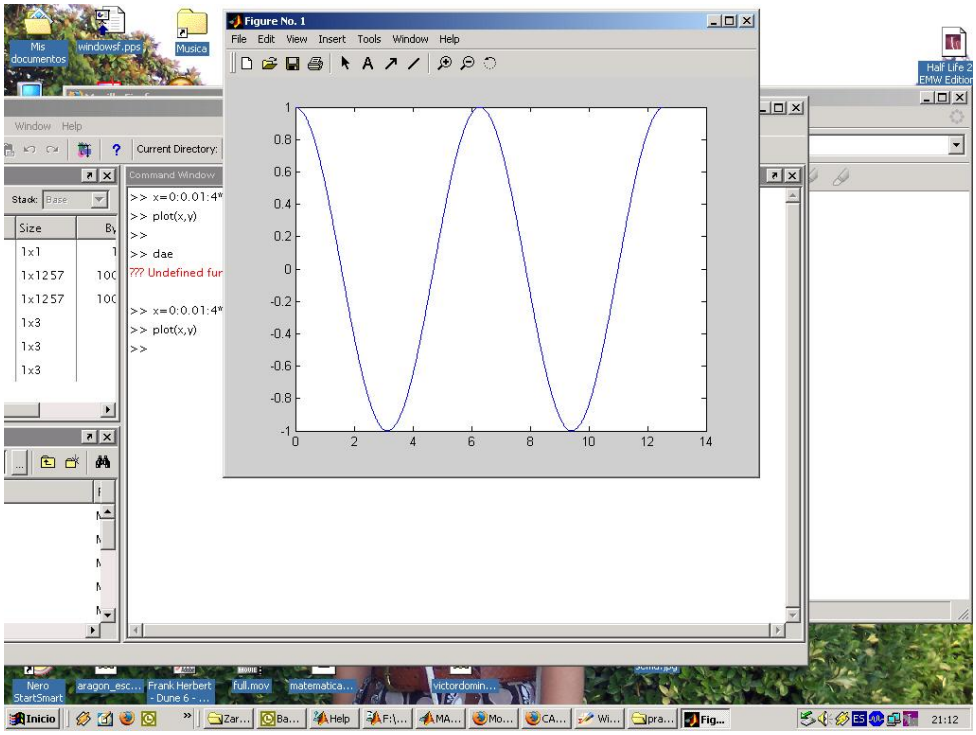


Figura 2.1: Ventana gráfica

```
>> clear i
>> t=linspace(0,2*pi,9);
>> plot(exp(i*t));
```

¿Te parece natural? ¿Qué observas con la escala? ¿Qué sucede si se ejecuta `axis equal`?. ¿Como dibujarías una circunferencia?.

□

2.1.2. Segundo nivel

El comando además acepta una serie de argumentos opcionales que entre otras cosas permiten controlar el color, colocar **marcas** sobre los puntos `(x(i), y(i))` y que tipo de marcas y cómo se unen estos puntos. Así, en su aspecto más general,

```
plot(x,y,S)
```

dibuja `y` vs. `x`, con S una cadena de caracteres que se construye con[2]

`b`	`azul`	`.`	`punto`	`-`	`linea 'solida'`
`g`	`verde`	`o`	`circulo`	`:`	`punteado`
`r`	`rojo`	`x`	`equis`	`-.`	`punto-linea`

[2]Traducido directamente de la ayuda de Matlab

c	cian	+	cruz	--	linea-linea
m	magenta	*	estrella		
y	amarillo	s	cuadrado		
k	negro	d	diamante		
		v	triangulo (down)		
		^	triangulo (up)		
		<	triangulo (left)		
		>	triangulo (right)		
		p	pentagono		
		h	hexagono		

La primera columna especifica el color utilizado, la segunda la marca sobre cada punto y la tercera que patrón siguen las líneas utilizadas para unir los puntos.

Por ejemplo el siguiente fichero `script`

```
x=linspace(0,4,100);
y=exp(-x).*cos(2*pi*x);
figure(1); plot(x,y,'.')
figure(2); plot(x,y,'r-.')
figure(3); plot(x,y,'sm--')
figure(4); plot(x,y,'hg')
figure(5); plot(x,y,'kv:')
```

genera (tras el reordenamiento manual de las ventanas adecuado, obviamente), la pantalla mostrada en la figura 2.2.

El comando `figure` abre una ventana gráfica asignándole un número. Si la ventana ya está abierta, la coloca como la ventana de salida por defecto. En particular, `figure` permite manejarse con varias ventanas de forma simultánea.

Para superponer varios dibujos sobre una misma ventana podemos

- Utilizar `plot` de la siguiente forma[3]

  ```
  >> plot(x,y,x2,y2,x3,y3,x4,y4);
  ```

  Con

  ```
  >> plot(x,y,'r-',x2,y2,'b:',x3,y3,'m-.',x4,y4,'k--');
  ```

  especificamos detalles individuales para cada dibujo. De ambas formas se dibujan simultáneamente las curvas definidas por los vectores $\{\{\texttt{x},\texttt{y}\},\{\texttt{x2},\texttt{y2}\},\{\texttt{x3},\texttt{y3}\},\{\texttt{x4},\texttt{y4}\}\}$.

- Utilizar la orden `hold on` que activa la superposición en pantalla. Por ejemplo,

  ```
  >> hold on  %activamos superposicion
  >> plot(x,y,'r-')
  >> plot(x2,y2,'r:')
  >> plot(x3,y3,'m.-')
  >> plot(x4,y4,'k--')
  ```

[3] Nótese que se asignan distintos colores a cada una de las gráficas de forma automática.

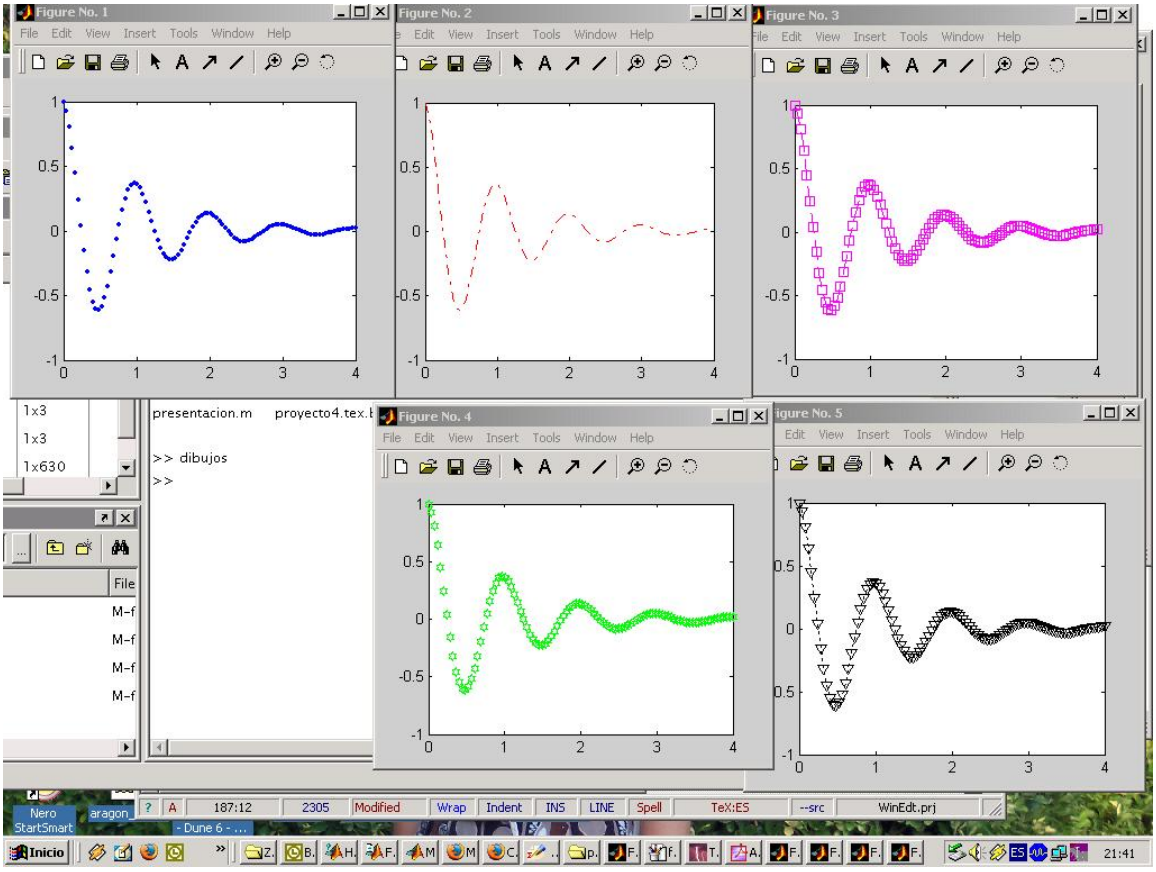


Figura 2.2: Ventana gráfica

Esta opción es más versátil dado que permite superponer gráficas construidas con diferentes comandos. La superposición se desconecta con `hold off`, de forma que un nuevo dibujo borrará los anteriores.

2.1.3. Tercer nivel

Ya en un tercer nivel, se pueden acceder a detalles concretos del dibujo, como el tamaño y color de las *marcas*, la anchura de la línea,... Las diferentes opciones llevan nombre nemotécnicos que ayudan a recordarlo[4]. Destacamos entre las más importantes a

[4]nemotécnicos en inglés, *of course*.

`color`: color de la línea.

`LineWidth`: anchura de la línea.

`MarkerEdgeColor`: color del borde de las marcas

`MarkerFaceColor`: color de la marca

`MarkerSize`: tamaño de la marca

Para especificar un color, se puede utilizar uno de estos caracteres $\{\texttt{b}, \texttt{g}, \texttt{r}, \texttt{c}, \texttt{m}, \texttt{y}, \texttt{k}\}$ o bien un vector con tres componentes con valores entre 0 y 1 que especifica un color según el estándar RGB[5].

Por ejemplo

```
>>x=0.01:0.2:2; y=sin(x)./x;
>>plot(x,y,'o-.','color',[0.2 0.4 0.6],'linewidth',2,...
  'markeredgecolor','k','markerfacecolor',[0.9 0.6 0.4],...
  'markersize',9)
```

Ejercicio 2 Se trata de que dibujes las funciones seno y coseno en $[-2\pi, 2\pi]$, la primera en rojo y la segunda en azul. El ancho de línea debe ser de dos puntos y deben seguir dos estilos diferentes, a tu elección.

Puedes empaquetar las instrucciones en un fichero *script*. Te será más cómodo para editar y cambiar los que desees.

□

Nota: la ventana gráfica

Es posible editar directamente estas propiedades en la *figura*. Para ello, seleccionar **edit plot** (véase Figura 2.3) y pulsar sobre la gráfica con un doble click. Se desplegará una ventana adicional con varias pestañas que informan y permiten modificar diferentes características del objeto. Si se pulsa sobre el fondo de la pantalla controlaremos más aspectos del propio entorno, como la escala, el ratio entre el eje OX y OY, las marcas sobre los ejes,..., aspectos que trataremos seguidamente.

No entraremos a explicar estos detalles en profundidad. Éste un buen ejemplo donde la prueba, la experimentación y el ensayo–error permiten aprender mejor y más rápidamente que cualquier manual que podamos redactar.

La ventaja de editar a través de comandos es que podemos dotar de un aspecto determinado nuestros dibujos sin necesidad de retocarlos en cada paso y para cada ejemplo. El precio que se paga es un mayor trabajo. Por contra la manipulación a través del ratón es mucho más simple, aunque rehacer un dibujo exige volver a dar el aspecto final deseado del dibujo a mano. Queda por tanto a elección del usuario qué método utilizar en cada caso.

[5] **R**ed, **G**reen, **B**lue. Crea un color añadiendo partes de rojo, verde y azul según los valores de un vector de tres componentes

Ejercicio 3 Ejecuta

```
>> t=linspace(-2*pi, 2*pi,200);
>> y1=cos(t);
>> y2=sin(t);
>> plot(t,y1,t,y2)
```

Desde la ventana gráfica dale el aspecto que habías dado en el ejercicio 2.

□

2.1.4. Comandos asociados

Paralelamente a `plot`, existe una serie de comandos qué controlan el entorno donde se despliega la gráfica. Entre los más elementales (o al menos, más fáciles de utilizar), podemos destacar:

`figure`	`clf`	`cla`	`axis`	`xlim`	`ylim`
`grid`	`legend`	`ylabel`	`xlabel`	`ytitle`	`title`

`figure`: `figure(n)` despliega una ventana gráfica asignándole el número n. Si ésta existe, la transforma en la ventana gráfica de salida.

`clf`: borra la ventana actual (*figure*) de gráficos.

`cla`: borra el *axes* actual. El *axes* es la parte de la ventana utilizada para dibujar. Una ventana puede contener varios *axes* (véase el comando `subplot` tratado más adelante).

`hold`: `hold on` permite que sucesivas gráficas se vayan solapando. `hold off` desconecta esta opción (es la que está por defecto).

`axis`: un comando algo complejo. Controla el ratio entre los ejes OX y OY, la parte del dibujo que se muestra por pantalla, forma de mostrar las coordenadas....

`xlim, ylim`: especifica los límites del dibujo. Se puede utilizar para *centrar* el dibujo

`grid`: `grid on` muestra una malla en pantalla; `grid off` la desconecta

`legend`: despliega una *leyenda*, esto es un cuadro explicativo sobre las gráficas presentes.

`xlabel, ylabel`: añade títulos (*etiquetas*) a los ejes OX y OY

`title`: Coloca un título en la cabecera del dibujo.

`whitebg`: Asigna un color al fondo del dibujo

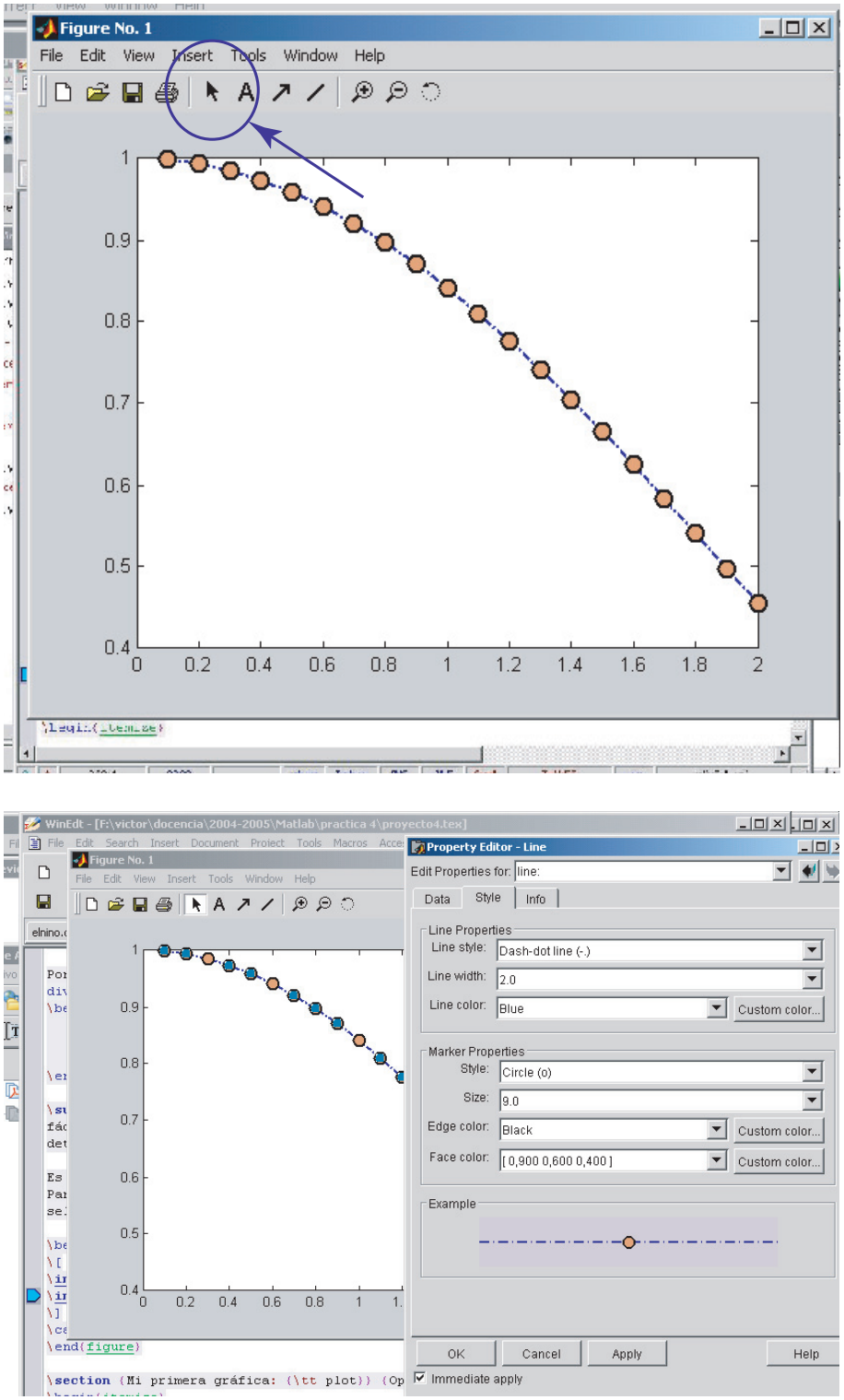


Figura 2.3: Edición de un dibujo

A modo de ejemplo, el siguiente conjunto de instrucciones (empaquetado en un fichero script[6]) despliega las gráficas mostradas en la Figura 2.4

```
figure(1)                          % desplegamos ventana 1
clf                                % borramos todo
x=linspace(0,5,100);
f=inline('exp(-n*x).*cos(x)'); % definimos funciones (vectorizadas)
hold on                            % solapamiento de graficas
y=f(1/3,x);
plot(x,y,'k--','linewidth',2)
y=f(1,x);
plot(x,y,'r-.','linewidth',2)
y=f(3,x);
plot(x,y,':','color',[0.0,0.0,0.5],'linewidth',2)
y=f(9,x);
plot(x,y,'-','color',[0.0,0.3,0.0],'linewidth',2)
grid on                            % desplegamos la red
axis([-0.5,6,-0.25,0.5])           % rango de los graficos
xlabel('Eje OX','fontname', 'Comic Sans Ms', 'Fontsize',12)
ylabel('Eje OY','fontname', 'Comic Sans Ms', 'Fontsize',12)
whitebg([.85 1 1]);                % color de fondo
title('Algunas graficas de funciones',...
        'Fontsize',16,'Fontname','Times new roman')
legend('exp(-x/3).*cos(x/3)', 'exp(-x).*cos(x)',...
       'exp(-3x)*cos(3x)', 'exp(-9x).*cos(9x)');
```

Nota. En el comando `title` hemos utilizado los atributos `fontname` y `fontsize` para especificar la fuente y su tamaño utilizada para "escribir" el título. Otros atributos son

`fontweight`: los valores posibles son `light, normal, demi, bold`. Especifican el trazo de los carácteres, desde fino (`light`) hasta negrita (`bold`).

`fontangle`: con `normal, italic, oblique` fijamos la "inclinación" de la fuente.

`rotate`: especifica el ángulo con el que se escribe el texto. El valor por defecto, 0, es la escritura horizontal, mientras que con 90 se escribe el texto en vertical.

Este tipo de atributos están disponibles para cualquier comando que se ocupe de desplegar textos en la pantalla gráfica. Por ejemplo, `xlabel`, `ylabel`, `title`, `text`,...

Ejercicio 4 Se trata de visualizar el efecto del comando `axis` con diferentes opciones sobre el aspecto final de un dibujo.

Teclea

[6] Observa la utilización de ... para cortar líneas demasiado largas.

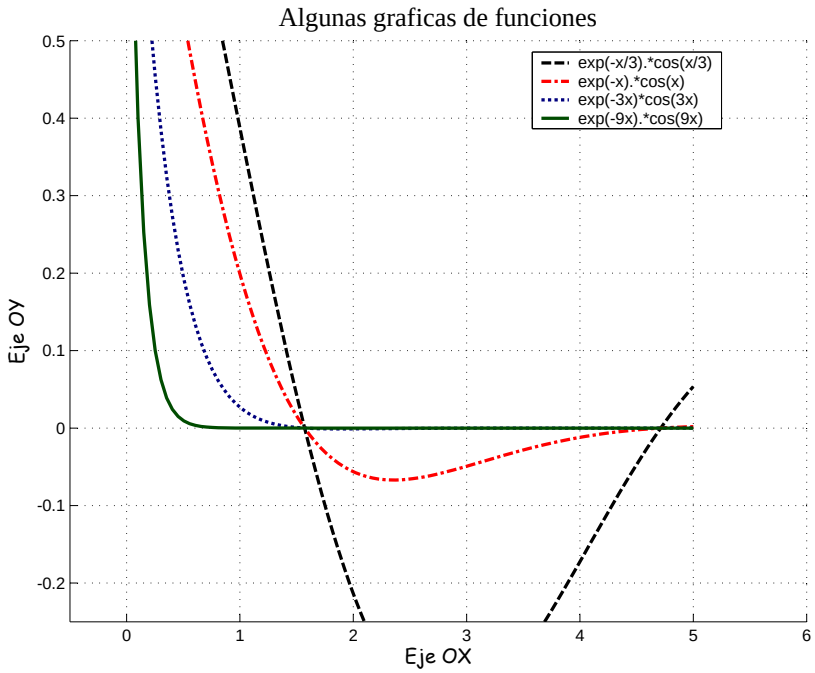


Figura 2.4: Un ejemplo

```
>> clf
>> x = 0:.025:pi/2;  plot(x,tan(x),'-ro')
```

La función tangente presenta una asíntota vertical[7] en $\pi/2$. Teclea los siguientes comandos y observa el aspecto final de la figura

```
>> axis equal
>> axis image
>> axis normal          % vuelta al formato original
>> axis([0 pi/2 0 5])   % especificamos el rango de salida
>> axis tight
```

¿Intuyes qué hace cada comando? Utilizando la ayuda deduce exactamente qué hace cada comando.

□

2.1.5. Cuarto nivel: `get` y `set`

Los comandos `get` y `set` permiten acceder y cambiar los atributos de cualquier objeto gráfico. No son, por tanto, en medida alguna exclusivos de `plot`. Más bien todo objeto gráfico, desde una simple curva hasta la propia ventana donde se despliega el dibujo

[7] Tiende a infinito

tiene asociada un puntero, un *handle* al que se encuentra enlazado y los diferentes valores opcionales se pueden visualizar (`get`) y editar (`set`) desde la línea de comandos.

Nos limitaremos de momento a dar unas ideas a grandes trazos para el comando `plot`.

```
>> x=0:0.01:pi;
>> h=plot(x,x.*cos(4*x));
```

En estos momentos `h` es un puntero que apunta al dibujo desplegado sobre la ventana. Ahora

```
>> get(h,'linestyle')

ans =

 -

>> get(h,'marker')

ans =

none

>> get(h,'linewidth')

ans =

0.5000
```

nos informa de que el dibujo se ha trazado con línea continua, sin ninguna marca y con anchura de línea 0.5.

Con `set` podemos cambiar cualquiera de estos atributos

```
>> set(h,'linewidth',2);
>> set(h,'color',[0.5 0.6 0.2])
>> set(h,'linestyle','-.')
```

de forma que la gráfica pasa a tener una anchura de 2 puntos, cambia el color y el estilo ahora es *punto-raya*.

Si se ejecuta `get(h)` podemos visualizar todos los atributos del objeto gráfico:

```
>> get(h)
   Color = [0.5 0.6 0.2]
   EraseMode = normal
   LineStyle = -.
   LineWidth = [2]
   Marker = none
   MarkerSize = [6]
   MarkerEdgeColor = auto
```

```
MarkerFaceColor = none
XData = [ (1 by 315) double array]
YData = [ (1 by 315) double array]
ZData = []

BeingDeleted = off
ButtonDownFcn =
Children = []
Clipping = on
CreateFcn =
DeleteFcn =
BusyAction = queue
HandleVisibility = on
HitTest = on
Interruptible = on
Parent = [101.001]
Selected = off
SelectionHighlight = on
Tag =
Type = line
UIContextMenu = []
UserData = []
Visible = on
```

De forma similar

```
h= legend('exp(-x/3)*cos(x/3)', 'exp(-x)*cos(x)',...
       'exp(-3x)*cos(3x)', 'exp(-9x)*cos(9x)');
```

devuelve en `h` un variable que permite a continuación la manipulación de muchas de sus propiedades. Para ello se utiliza la instrucción `set`

```
set(h,'fontsize',11,'fontname','arial','fontangle',..
   'oblique','color',[0.8 0.8 0.8])
```

cambia alguno de los atributos de la *leyenda*, como la fuente y su tamaño, la inclinación y el color de fondo.

Otro ejemplo lo da el siguiente código:

```
01    x=linspace(0,4,1000);
02    y=x.*log(x);
03    figure(1);
04    clf       % borramos
05    plot(x,y,'--','linewidth',3);
06    h2=gca;   % accedemos al handle de la grafica
07    set(h2,'xtick',[0 0.25 0.5 1 2 4],'fontsize',16,'ygrid',...
08         'off','xgrid','on','linewidth',2,'gridlinestyle','-.')
09    title('x*log(x)','fontangle','oblique','fontname',...
10         'Comic Sans ms ','fontweight','bold','fontsize',20)
```

accede a la estructura de la gráfica, que esencialmente es el marco donde desplegamos los dibujos. La líneas `07-08` edita los atributos `xtick` que especifica donde colocar las marcas en el eje OX, la fuente utilizada en el dibujo, la opción de `grid` en el eje OY, el tamaño de la línea para la "malla" y cómo dibujar ésta. El resultado se puede ver en la figura 2.5

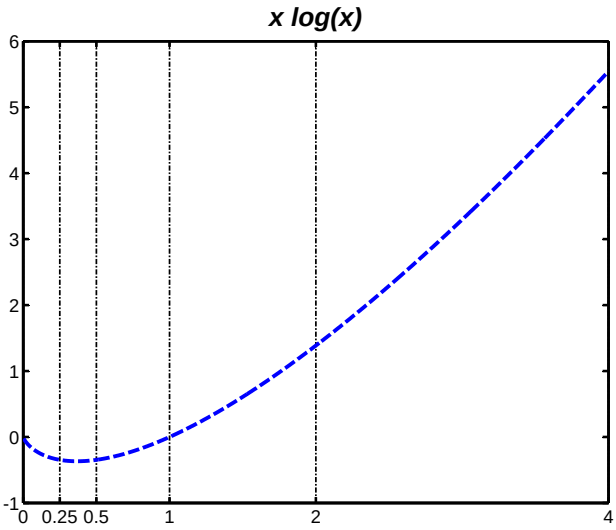


Figura 2.5: Salida correspondientes

Nota

Todas las propiedades anteriores se pueden modificar de una forma más fácil editándolo directamente en la ventana gráfica. Su manejo es fácil intuitivo (seleccionar, doble click, botón derecho del ratón....).

Con las órdenes `helpwin line` y `helpwin axes` obtenemos la información de las diferentes opciones para la instrucción `plot` (y similares) y para la figura.

Con `get` accedemos a los diferentes parámetros de un objeto gráfico y sus valores. Por ejemplo

```
>> get(gca) , get(gcf)
```

muestran los valores para el *axis* y *figure* utilizado en ese momento (*get current axis* y *get current figure*).

2.2. Otras salidas gráficas

Reseñaremos a continuación otros comandos relacionados con dibujos y gráficas bidimensionales de manejo similar a `plot`

`polar`: curvas en polares;

`semilogx, semilogy`: similar a `plot` pero utilizando, respectivamente, una escala logarítmica en el eje OX y el eje OY;

`loglog`: escala logarítmica en ambos ejes;

`stem`: dibuja puntos uniéndolos con una linea vertical al eje OX;

`stairs`: traza una gráfica en forma de escalera;

`bar, barh, bar3`: despliega gráficas en forma de barras. Muy apropiada para representar datos y estadísticas (estilo excel);

`area`: Muestra los datos en una gráfica de forma "acumulada". El color utilizado se controla mediante el comando `colormap`, cuyo funcionamiento veremos más adelante;

`line`: Une puntos mediante líneas. Es una instrucción de bajo nivel cuyo funcionamiento es similar a `plot`;

`patch`: una instrucción también de bajo nivel[8], construye polígonos, o caras en tres dimensiones y asigna un color a la cara definida;

2.3. El comando `subplot`

Este comando permite la visualización de diferentes subventanas gráficas (`axes`) en una misma ventana. A modo de ejemplo,

```
subplot(231)
```

define una ventana con seis zonas para salidas gráficas dispuestas en 2×3 (dos filas, tres columnas) y accede a la primera de ellas. La numeración es la desplegada en la figura 2.6.

Por ejemplo, las instrucciones

```
x=linspace(0,2*pi,150);

subplot(321)
plot(x,sin(x),'linewidth',2);
title('sin(x)','fontsize',14)
axis tight
subplot(322)
plot(x,cos(x),'linewidth',2);
title('cos(x)','fontsize',14)
axis tight
subplot(323)
plot(x,cos(2*x),'linewidth',2);
```

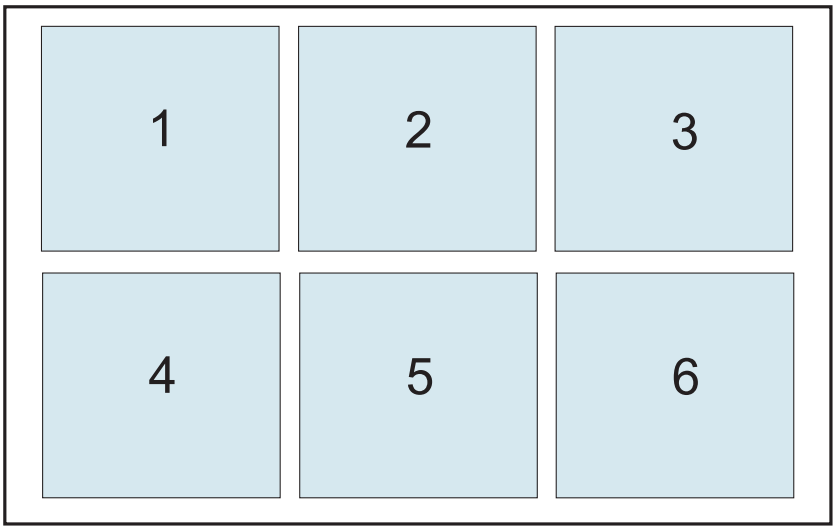


Figura 2.6: Numeración con `subplot`

```
title('sin(2x)','fontsize',14)
axis tight
subplot(324)
plot(x,sin(2*x),'linewidth',2);
title('cos(2x)','fontsize',14)
axis tight
subplot(325)
plot(x,sin(4*x),'linewidth',2);
title('sin(4x)','fontsize',14)
axis tight
subplot(326)
plot(x,cos(4*x),'linewidth',2);
title('cos(4x)','fontsize',14)
axis tight
```

crean la figura 2.7.

2.4. Comandos *fáciles de usar*

Matlab tiene implementada una serie de comandos que permiten dibujar de forma sencilla gráficas de funciones. No se tiene un control tan completo como con `plot` pero se compensa con su fácil uso. En el apartado que nos ocupa (gráficas bidimensionales) son resaltables

```
ezplot     ezpolar
```

La ayuda de Matlab es, en nuestra opinión, suficiente para aprender su manejo

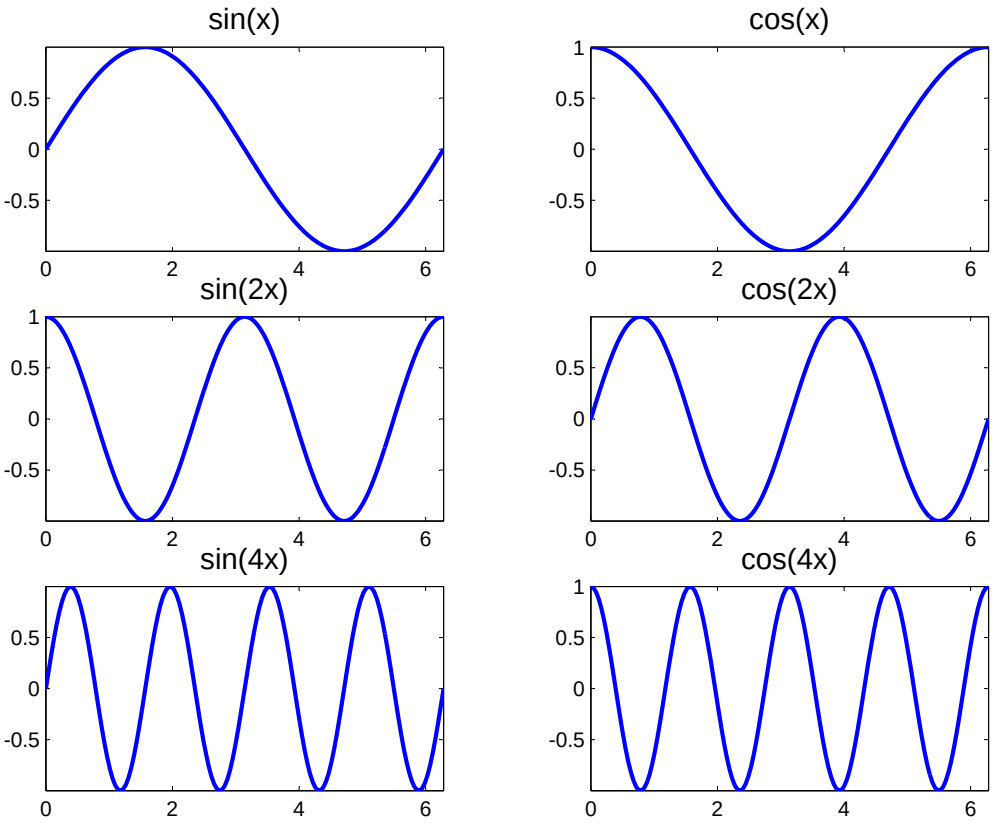


Figura 2.7: Disposición simultánea de gráficas con `subplot`

Capítulo 3

Gráficas en 3D

Cualquier dibujo en 3D, ya sea curvas o superficies, y su trazado a "mano" conllevan una serie de dificultades que hacen que los programas de dibujo y similares, como Matlab, se revelen especialmente útiles. Matlab incluye la posibilidad de *rotar* objetos y hacer animaciones en el espacio de forma que se consiga una visualización.

Hemos decidido incluir en este apartado las instrucciones relativas a curvas de nivel aunque hablando propiamente son dibujos bidimensionales. Su origen y su posterior interpretación nos conducen de nuevo al entorno espacial.

3.1. El comando `plot3`

Este comando sirve, a grosso modo, para dibujar curvas en el espacio. Su manejo es muy similar a `plot`, por lo que no nos detendremos demasiado en su explicación. Ésta es su sintaxis

`plot3(x,y,z,`*opciones*`)`

con *opciones* muy similares a `plot`. He aquí un ejemplo sencillo

```
clf
t=linspace(0,8*pi,200);
plot3(t.*cos(t),t.*sin(t),t,'r-','linewidth',2)
grid on        % dibujamos la 'malla'
title('Una curva en espiral...','fontsize',18,'fontname',...
     'Comic Sans MS','color',[0.675 0.000 0.000])
xlabel('eje OX','fontsize',14)
ylabel('eje OY','fontsize',14)
zlabel('eje OZ','fontsize',14)
```

cuyo resultado se muestra en la figura 3.1. Obsérvese que los comandos relacionados con los aspectos accesorios del dibujo (`axes`, `title`, `xlabel`, `grid`...) funcionan exactamente igual y que aparecen algunos nuevos cuya utilidad y manejo no debería causar sorpresa:

```
zlabel     zlim
```

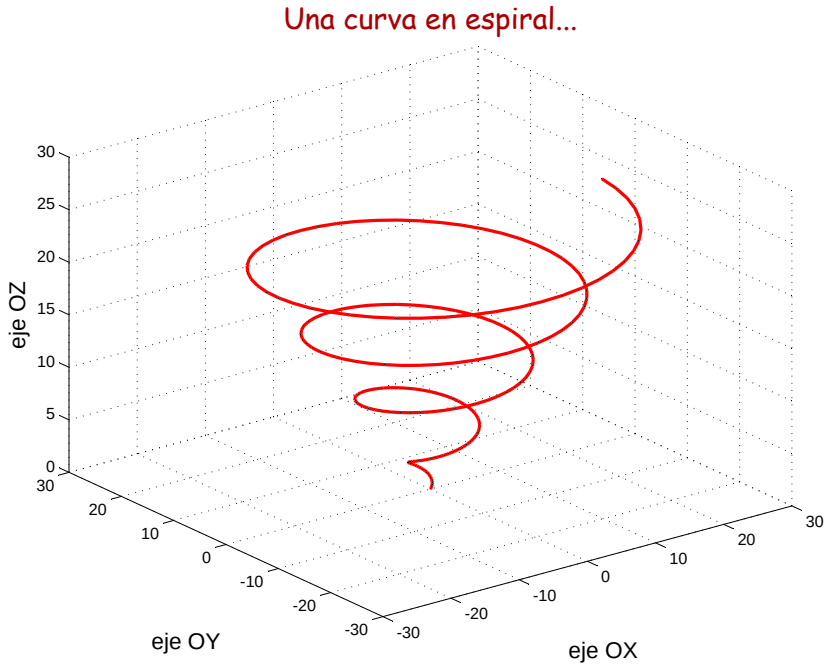


Figura 3.1: Un dibujo en 3D con `plot3`

Lo mismo se puede decir de la manipulación posterior del objeto geométrico, bien desde la línea de comandos o desde la ventana gráfica. Para poder manipular, rotar en 3D, el objeto gráfico basta presionar en la barra de herramientas en el botón señalado en la figura 3.2

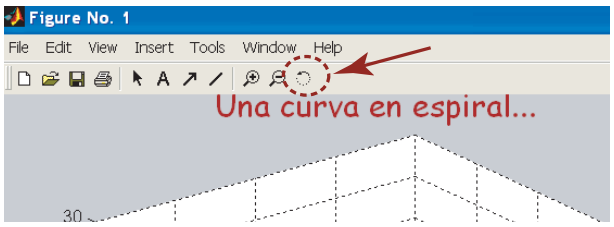


Figura 3.2: Botón para rotar los dibujos

3.2. El comando `surf`

3.2.1. Primer nivel

El comando `surf` dibuja superficies en el espacio. El formato es

```
surf(x,y,z)
```

donde `x`, `y` especifican una *malla* en el plano y `z` la altura correspondiente. El color que se da a cada punto es por defecto igual a la altura.

Queda pendiente, no obstante, como construir una malla. Aunque se puede hacer a mano (Matlab ofrece en su ayuda información detallada de cómo hacerlo), es mejor echar mano de comandos propios de Matlab, en este caso, `meshgrid`. Así, si `x` e `y` son dos vectores, este comando devuelve dos vectores que contienen la información de la malla. La construcción de dicha malla sigue el esquema marcado por la figura 3.3.

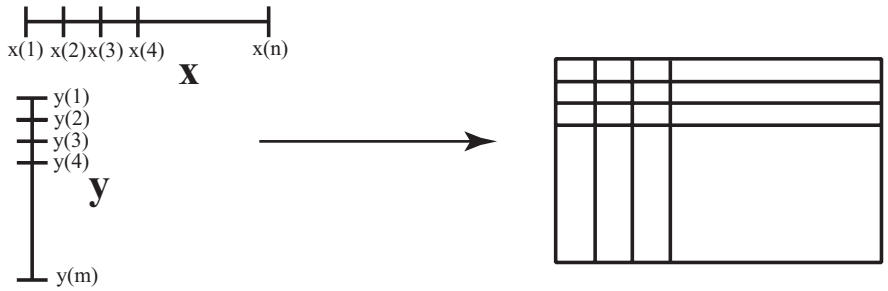


Figura 3.3:

Por ejemplo

```
>> x=[0 0.33 0.67 1]; y=[-1 0 1];
>> [X,Y]=meshgrid(x,y)

X =

    0    0.3300    0.6700    1.0000
    0    0.3300    0.6700    1.0000
    0    0.3300    0.6700    1.0000


Y =

   -1    -1    -1    -1
    0     0     0     0
    1     1     1     1
```

Las variables `X` e `Y` forman una malla según la sintaxis de Matlab.

El dibujo de la figura 3.4 se ha construido con el siguiente conjunto de instrucciones

```
>> x=linspace(-2,2,40);
>> y=linspace(-1,1,20);
>> [X,Y]=meshgrid(x,y);
>> size(X)
```

```
ans=

    20   40
>> size(Y)

ans=

    20   40
>> surf(X,Y,X.^2-Y.^2)
```

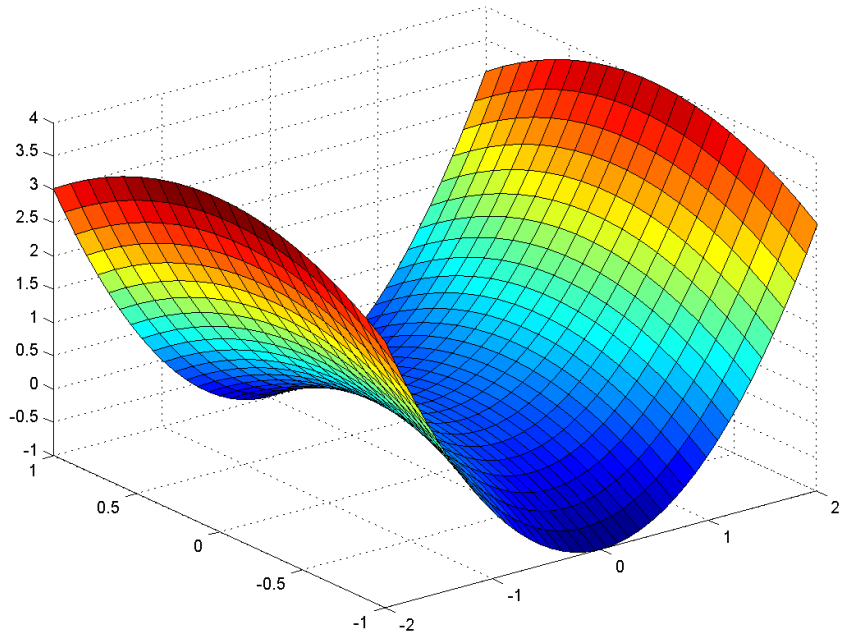


Figura 3.4: Superficie creada con `surf`

Comandos asociados

Relacionados con `surf` se dispone de una serie de instrucciones que controlan tanto el aspecto gráfico como la visualización final de la superficie. Nos limitaremos a hablar de estos cuatro comandos

```
colorbar        colormap        daspect        pbaspect
```

`colorbar` despliega una barra de colores que informa sobre la correspondencia entre valor numérico y color utilizado. Por defecto se despliega verticalmente a la derecha del dibujo, aunque puede mostrarse horizontalmente si así se desea[1].

[1] Consulta la ayuda...

`colormap` especifica que colores se van a utilizar en el dibujo. Existe un conjunto de formatos predefinidos que listamos a continuación

```
autumn    bone    colorcube    cool     copper    flag    gray
hot       hsv     jet          lines    pink      prism   spring
summer    white   winter       default
```

Para cambiar a un formato basta ejecutar

```
>> colormap('bone')
```

y cambiará el color utilizado en la gráfica actual y en las siguientes. Se pueden también definir formatos personalizados, bien mediante la línea de comando o desde la propia ventana gráfica.

`daspect` controlamos la relación entre los ejes del dibujo. Baste decir que

```
daspect([1 1 1])
```

fija que las proporciones de los ejes OX, OY y OZ sean iguales. Es decir con

```
>> sphere(40);      % dibuja una esfera
>> daspect([1 1 1]) % relaciones 1:1:1 en los ejes
```

la esfera se muestra como una esfera y no como un elipsoide.

`pbaspect` similar al anterior pero relacionado con la *caja* que enmarca el dibujo.

3.2.2. El comando `surf`: segundo y tercer nivel

En un segundo nivel de utilización del comando anterior podemos acceder a algunas opciones adicionales entre las que podríamos destacar[2]

[2] siempre a nuestro juicio...

`'edgealpha'`: se especifica un valor entre 0 y 1 que define la *transparencia* que tiene la rejilla o red en el dibujo (con 0 la red es transparente y por tanto no se ve)

`'facealpha'`: como la opción anterior pero relativa a las caras.

`'edgecolor'`: cuatro valores posibles: un color (en el formato habitual), `'none'`, `'flat'`, `'interp'`. Especifica qué color utilizar en la rejilla. Por defecto es negro, pero se puede eliminar la rejilla (`'none'`),se puede utilizar el color marcado por el primer vértice (`'flat'`) o bien utilizar un color definido por los colores de los dos vértices de cada eje

`'facecolor'`: igual que la opción anterior, con las mismas opciones. En este caso especifica el color de cada cara de la malla utilizada en el dibujo. Si no se especifica, el color se calcula utilizando la altura de los puntos, es decir, la coordenada `z`.

`'LineWidth'`: anchura de las líneas utilizadas en el dibujo de la red

`'marker'`: qué marca colocar en cada punto del dibujo. Sigue el mismo formato que `plot` (por defecto no coloca ninguna marca).

`'MarkerEdgeColor'`
`'MarkerFaceColor'` igual que en `plot`
`'MarkerSize'`

`'Meshstyle'` Tres opciones `both` (defecto), `row` y `column`. Especifica cómo despliega la red, entera, sólo las filas o sólo las columnas, respectivamente.

Por ejemplo, la figura 3.5 se obtiene con

```
f=inline('x^2-y^2'); f=vectorize(f);
x0=linspace(-2,2,6);
y0=linspace(-2,2,4);
[X0,Y0]=meshgrid(x0,y0);
x1=linspace(-2,2,40);
y1=linspace(-2,2,40);
[X1,Y1]=meshgrid(x1,y1);
hold on    % solapamos dos dibujos
surf(X0,Y0,f(X0,Y0),'facecolor','none','edgecolor','k',...
     'marker','o', 'markersize',6,'MarkerFaceColor','k', 'linewidth',2);
surf(X1,Y1,f(X1,Y1),'facecolor','interp', 'facealpha',0.5,...
    'edgecolor','none');
colorbar
colormap('spring')
```

genera la figura 3.5

En un tercer nivel de dificultad (y de precisión), el comando devuelve un *handle* que permite modificar todas estas opciones a posteriori. Así, por ejemplo,

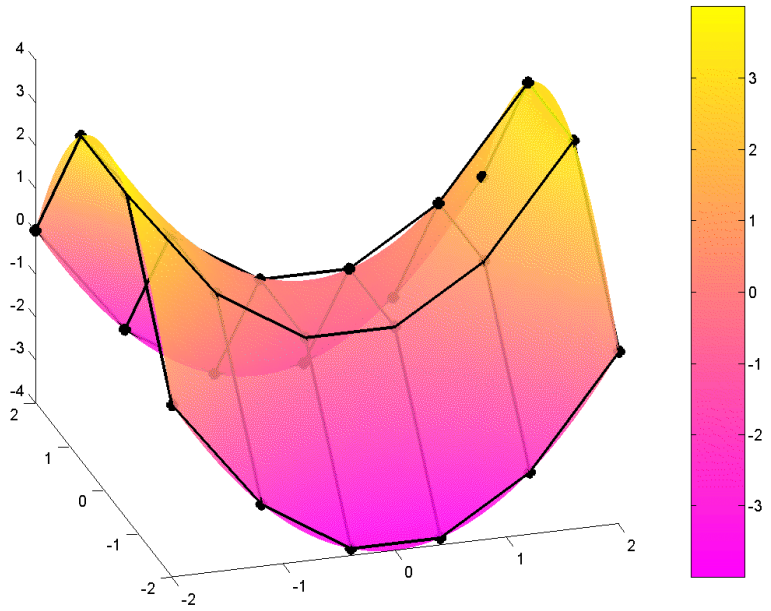


Figura 3.5: Algunas opciones con `surf`

```
>> x1=linspace(-2,2,60); y1=linspace(-2,2,60);
>> [X1,Y1]=meshgrid(x1,y1);
>> h=surf(X1,Y1,exp(-X1.^2-Y1.^2).*cos(pi*(X1.^2-Y1.^2)/2),...
   'facecolor','interp','edgecolor','none')
```

devuelve en h, después de dibujar, un puntero al dibujo.

Como antes, podemos leer una propiedad determinada mediante `get`:

```
>> get(h,'facecolor')

ans =

interp
```

o bien ver todas con

```
  >> get(h)
AlphaData = [1]
AlphaDataMapping = scaled
CData = [ (60 by 60) double array]
CDataMapping = scaled
EdgeAlpha = [1]
EdgeColor = none
EraseMode = normal
FaceAlpha = [1]
```

```
FaceColor = interp
LineStyle = -
LineWidth = [0.5]
Marker = none
MarkerEdgeColor = auto
MarkerFaceColor = none
MarkerSize = [6]
MeshStyle = both
XData = [ (60 by 60) double array]
YData = [ (60 by 60) double array]
ZData = [ (60 by 60) double array]
FaceLighting = flat
EdgeLighting = none
BackFaceLighting = reverselit
AmbientStrength = [0.3]
DiffuseStrength = [0.6]
SpecularStrength = [0.9]
SpecularExponent = [10]
SpecularColorReflectance = [1]
VertexNormals = [ (60 by 60 by 3) double array]
NormalMode = auto

BeingDeleted = off
ButtonDownFcn =
Children = []
Clipping = on
CreateFcn =
DeleteFcn =
BusyAction = queue
HandleVisibility = on
HitTest = on
Interruptible = on
Parent = [100.003]
Selected = off
SelectionHighlight = on
Tag =
Type = surface
UIContextMenu = []
UserData = []
Visible = on
```

Observamos así alguna de las opciones que hemos comentado y otras muchas más que no hemos estudiado.

Para cambiar una propiedad utilizamos de nuevo el comando `set`. Por ejemplo con

```
>> set(h, 'edgecolor',[0.2 0 0],'linewidth',1)
```

hacemos aparecer la rejilla en el dibujo con las líneas trazadas con anchura 1.

Nota

Las diferentes opciones para `surf`, que son compartidas por muchas de los comandos relacionados con el despliego de superficies y curvas de nivel, se pueden consultar en la **ayuda completa de** Matlab.

Mallados especiales

Con `sphere` podemos obtener el mallado de una esfera. Puede utilizarse para dibujar bien una esfera o incluso funciones definidas sobre una superficie. Por ejemplo

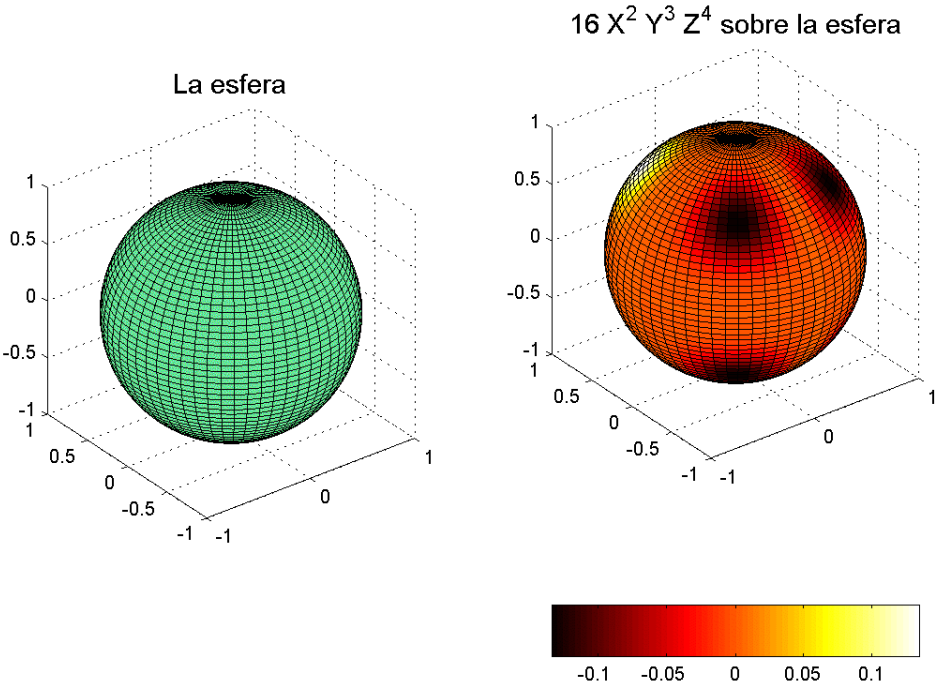


Figura 3.6: Esferas en 3D

```
[X,Y,Z]=sphere(60);
subplot(121)
surf(X,Y,Z,'facecolor',[0.4 0.9 0.6])
daspect([1 1 1])  % aspecto [1 1 1]
title('La esfera','fontsize',14)
subplot(122);
surf(X,Y,Z,16*X.^2.*Y.^3.*Z.^4)
title('16 X^2 Y^3 Z^4 sobre la esfera','fontsize',14)
colormap('hot')
colorbar('hor')   % barra de colores horizontal
daspect([1 1 1])  % aspecto [1 1 1]
```

véase la figura 3.6). Comandos similares son `ellipsoid` (elipsoides) y `cylinder` (cilindros).

3.3. Otros comandos relacionados

Comentaremos a continuación otras instrucciones relacionadas con gráficas de objetos tridimensionales.

3.3.1. `contour` y `contourf`

Despliegan las líneas de nivel del dibujo. Propiamente es una gráfica en 2D. La diferencia entre ellas es que la primera sólo traza las curvas de nivel mientras que la segunda colorea el espacio entre ellas.

Es posible añadir un texto sobre cada línea de nivel. Para ello basta hacer como en el ejemplo siguiente

```
>> x1=linspace(-2,2,60); y1=linspace(-2,2,60);
>> [X1,Y1]=meshgrid(x1,y1);
>> f=vectorize(inline('cos(x^2-y)'));
>> subplot(211)  % dos dibujos
>> [c,h]=contour(X1,Y1,f(X1,Y1)); colorbar;
>> clabel(c,h)   % Inserta el texto sobre las curvas de nivel
>> subplot(212)
>> [c,h]=contourf(X1,Y1,f(X1,Y1),5); colorbar % 5 curvas de nivel;
>> clabel(c,h,'fontsize',12);  % cambiamos tama\~{n}o letra
```

El resultado es el de la figura 3.7

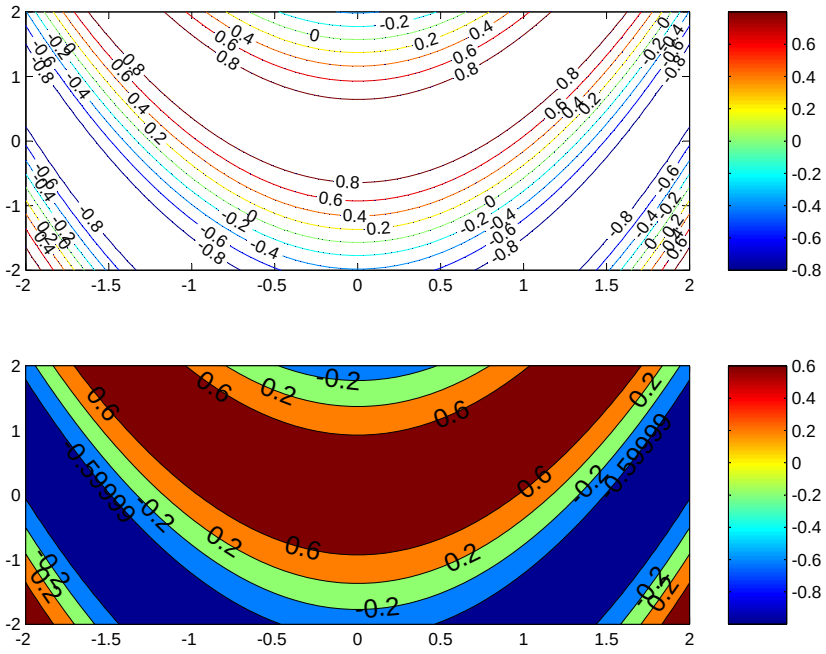


Figura 3.7: Líneas de nivel

3.3.2. `surfc` y `surl`

Ambas son variantes de `surf`. La primera despliega la superficie y dibuja en el plano OXY (plano inferior) las curvas de nivel.

La segunda es como `surf` pero el color que asigna a cada punto viene determinado por una *punto de iluminación exterior*. El resultado es el de una superficie de un color determinado iluminado desde un punto

3.3.3. `mesh, meshc, meshz`

Todos ellos dibujan únicamente la malla (rejilla) del dibujo. La segunda además añade las líneas de nivel, de forma semejante a como procedía `surfc`.

3.4. Comandos *fáciles de usar*

Los comandos

```
ezplot3      ezmesh      ezmeshc      ezsurf      ezsurfc
```

permiten trazar, de forma muy sencilla, curvas y superficies en el espacio. Su sencillez de uso (consulta la ayuda) queda lastrada con el inconveniente de que apenas se tiene control sobre el aspecto final.

3.5. Dibujos sobre dominios mallados en triángulos

Los comandos anteriores están pensados para dibujar superficies definidas esencialmente sobre una cuadrícula. Aunque en muchos casos esto es suficiente en ocasiones se trabaja con funciones definidas sobre conjuntos, o **dominios** en la terminología habitual, mucho más generales.

Una forma muy simple de trabajar con estos dominios es dividirlo en triángulos y construir la superficie solapando planos definidos sobre cada triángulo[3]. Los resultados que se obtienen son bastante satisfactorios, puesto que los triángulos son más flexibles que los rectángulos o paralelogramos a la hora de adaptarse a los dominios.

La división en triángulos de un conjunto así se denomina **triangulación** o **mallado** del dominio. Se dice que un mallado se hace más fino si el tamaño de los triángulos disminuye.

Una triangulación se dice **conforme** si la intersección de dos lados cualesquiera del mallado es o bien vacío (los triángulos no se tocan), o un vértice o un lado entero. Es decir no se admiten que un lado de un triángulo pueda ser parte de dos lados de dos triángulos diferentes (véase figura 3.8). Por otro lado, una familia de triangulaciones se dice **regular** si los triángulos *no se aplanan*, es decir, los ángulos de los triángulos no se hacen muy pequeños.

La siguiente cuestión es cómo almacenar la información de una triangulación. En primer lugar se considera una numeración de los nodos (vértices) de los triángulos. Las

[3]se utiliza el conocido resultado de que tres puntos no alineados definen un único plano. Es fácil ver también que la superficie así construida es continua

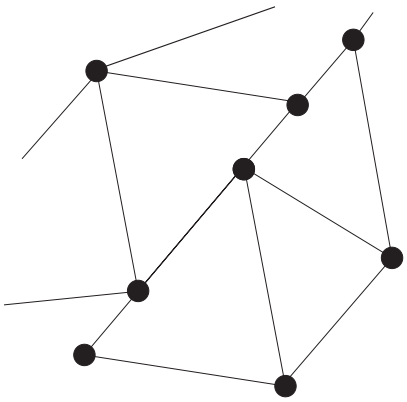


Figura 3.8: Triangulación no conforme

coordenadas de cada vértice se guardan en dos vectores, $\mathbf{x}$ e $\mathbf{y}$ de forma que las coordenadas del vértice `j` vengan dados por

$$(\mathbf{x}(\mathtt{j}), \mathbf{y}(\mathtt{j})).$$

En segundo lugar almacenamos los nodos que componen cada triángulo, mediante una matriz $\mathbf{t}$ de tres columnas y n filas con n el número total de triángulos. Así para saber que vértices están en el triángulo `i`, basta mirar

$$\mathbf{t}(\mathtt{i},1) \quad \mathbf{t}(\mathtt{i},2) \quad \mathbf{t}(\mathtt{i},3)$$

esto es, la fila `i` de $\mathbf{t}$.

A modo de ejemplo, el mallado expuesto en la figura 3.9 se guarda en las siguientes variables

- Triangulos

```
 8     3    16
 7     2    15
 9     5    21
11     6    20
10     1    14
12     4    17
16     9    21
 4     8    17
 1     7    14
15    11    20
 5    10    19
 6    12    20
14     7    15
 2    11    15
17    13    20
18    15    20
```

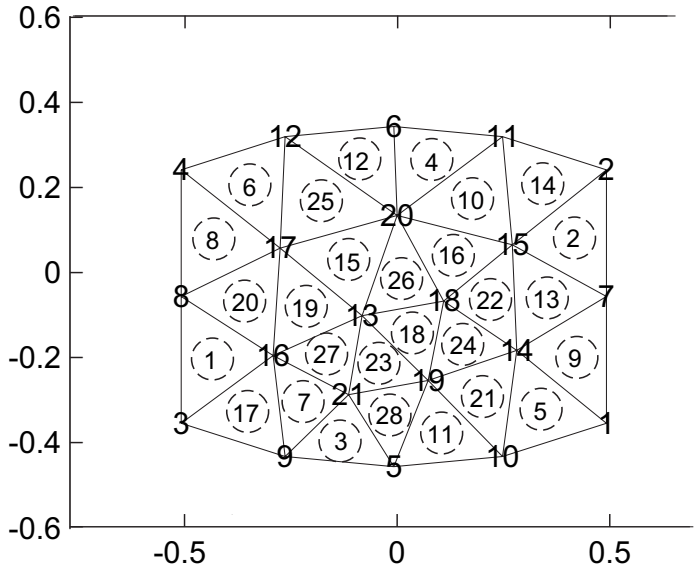


Figura 3.9: Ejemplo de triangulación. En el gráfico la numeración de los triángulos se rodea con un círculo

```
 3     9    16
18    13    19
16    13    17
 8    16    17
10    14    19
14    15    18
19    13    21
14    18    19
12    17    20
13    18    20
13    16    21
 5    19    21
```

- coordenadas:

```
     x          y

 0.5033    -0.3584
 0.5033     0.2378
-0.4967    -0.3584
-0.4967     0.2378
 0.0033    -0.4603
 0.0033     0.3397
 0.5033    -0.0603
```

```
-0.4967    -0.0603
-0.2523    -0.4363
 0.2590    -0.4363
 0.2590     0.3157
-0.2523     0.3157
-0.0716    -0.1050
 0.2924    -0.1866
 0.2820     0.0614
-0.2806    -0.1991
-0.2641     0.0537
 0.1204    -0.0707
 0.0838    -0.2577
 0.0113     0.1306
-0.1031    -0.2912
```

La información anterior es suficiente para construir la malla, la triangulación del dominio. Si además se desea construir la superficie en cuestión hace falta un vector adicional **z** de forma que

$$\mathbf{z}(\mathtt{j})$$

sea el valor de la función en el nodo `j`.

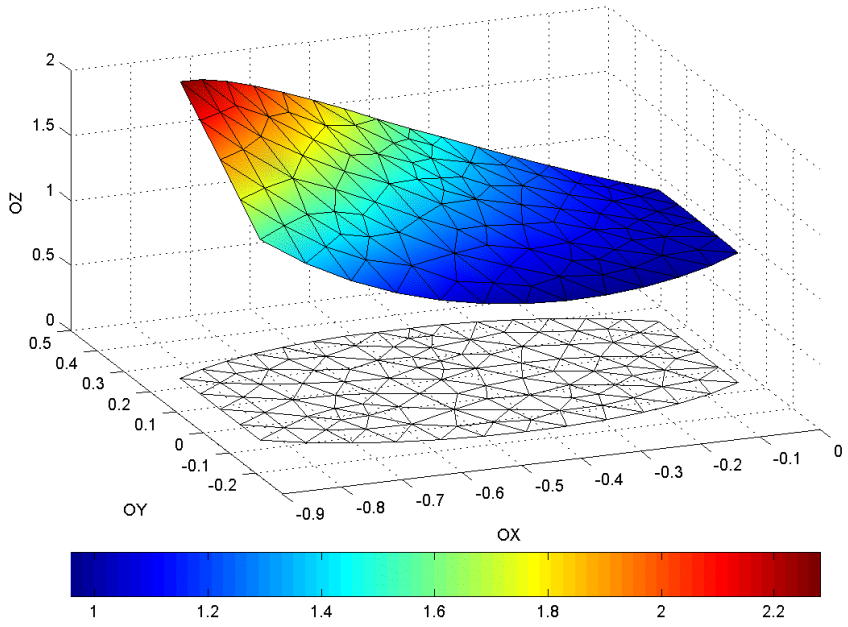


Figura 3.10: Gráfica sobre un dominio triangulado

Esta es una forma ya estándar de definir y trabajar con una triangulación que también sigue Matlab[4] con los comandos `trimesh` y `trisurf`

[4]La toolbox pdetool dedicada a la resolución de ecuaciones en derivadas parciales sigue una variante algo más complicada que la expuesta arriba.

La primera despliega la malla triangular especificada por `t`, la matriz conteniendo los triángulos, `x` y `y`, que dan las coordenadas de los vértices:

```
trimesh(t,x,y)
```

Se puede especificar la coordenada z de los vértices (la altura),

```
trimesh(t,x,y,z)
```

con lo que se dibuja la malla 3D correspondiente.

El comando `trisurf` es similar, pero *coloreando* las caras. Es propiamente hablando, una superficie.

Nota

Un tema nada trivial es la construcción de un mallado sobre un dominio (poligonal) dado. Existen multitud de algoritmos que tratan este problema. En principio se plantea la construcción de una malla gruesa, con pocos triángulos y de área considerable, con los triángulos lo más regulares posibles (sin deformar, *alargar*, en demasía los triángulos).

Posteriormente, se trata de *refinar* la malla, es decir, dividir los triángulos en triángulos más pequeños hasta que se alcance una precisión adecuada.

Esta idea se esconde detrás de aplicaciones como la interpolación (aproximación de una función) y especialmente el método de elementos finitos, probablemente el método[5] más utilizado en la resolución de problemas de contorno para ecuaciones en derivadas parciales.

Si se desea información de cómo se puede inicializar un malla en Matlab, así como el algoritmo utilizado se puede consultar el comando `initmesh` (incluido tema de triangulaciones de Delaunay). Para el refinamiento, véase `refinemesh`.

Por último, y volviendo de nuevo al comando `surf`, es posible dibujar superficies definidas sobre partes de un rectángulo. Para ello, cualquier punto cuya coordenada z sea un `NaN` (*not a number*), no se dibuja. Por ejemplo

```
>> x=linspace(-1,1,101); y=x;
>> [X,Y]=meshgrid(x,y);
>> Z=-X.^2-Y.^2;
>> Z(sqrt(X.^2+Y.^2)<0.5)=NaN;
>> surf(X,Y,Z)
```

despliega una superficie[6] con un *hueco* en la circunferencia centrada en el origen y de radio $1/\sqrt{2}$ (véase Figura 3.11). De todos modos se observa la deficiente aproximación de los cuadrados, propios de `surf`, del círculo interior.

[5] Propiamente hablando es una familia de métodos

[6] La tercera línea merece un comentario, aunque instrucciones parecidas han sido tratadas en el Tema 1. Al hacer la comparación `sqrt(X.^2+Y.^2)<0.5`, Matlab devuelve una matriz lógica, de 1s y 0s. El comando en `Z(sqrt(X.^2+Y.^2)<0.5)=NaN`, hace que los puntos situados dentro del círculo de radio $1/\sqrt{2}$, que es donde la expresión lógica es 1, tomen como valor `NaN`

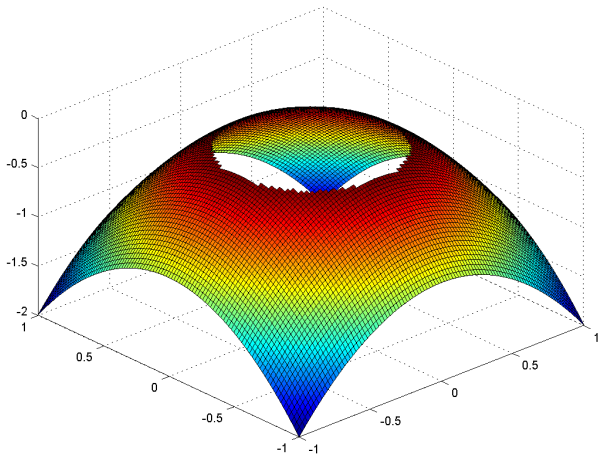


Figura 3.11: Utilización de `NaN` en un dibujo

Parte II

Interpolación

Capítulo 4

Interpolación polinómica de Lagrange

Un problema ya clásico es la construcción, o aproximación de una función conocidos unos pocos valores. Algunas referencias a este tipo de problemas se remontan al manejo de tablas trigonométricas y posteriormente logarítmicas y exponenciales donde unos pocos valores estaban tabulados y para valores intermedios era necesario un proceso de interpolación.

Ciertamente la llegada de los ordenadores ha contribuido a que parte de estas aplicaciones hayan perdido su sentido pero paralelamente han surgido nuevos problemas como aquellos relacionados con el diseño gráfico (CAD, de *computer aided design*). Un buen ejemplo lo encontramos en diversas aplicaciones muy prácticas que requerían la construcción curvas, o superficies en el espacio, que pasaran por unos puntos determinados y que tuvieran buenas propiedades geométricas.

En estos apuntes nos restringiremos a la interpolación polinómica (esto es, a la construcción de polinomios que pasen por unos puntos predeterminados) y comentaremos brevemente la interpolación polinómica a trozos con las funciones *spline*.

Por último trataremos de forma superficial las curvas Bezier. Propiamente hablando, no se trata de un problema de interpolación sino de trazar curvas suaves que sigan el camino, con cierta libertad, marcado por un polígono. Su fácil construcción anima a su exposición como cierre de este tema.

Capítulo 5

Interpolación polinómica

5.1. El problema

El problema que queremos resolver es el siguiente

> Dados un conjunto de $n+1$ puntos $\{(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)\}$, construir un polinomio p_n de grado n tal que
>
> $$p_n(x_j) = y_j, \qquad j = 0, \dots, n.$$

El polinomio p_n se le conoce como polinomio de interpolación de Lagrange. Habitualmente, y_j son valores de una función f que por ser o bien desconocida en el resto de puntos, o bien porque es cara computacionalmente de evaluar, sólo se dispone de su valor en un conjunto discreto de puntos.

Tenemos una serie de cuestiones pendientes:

- ¿Se puede construir siempre el polinomio de interpolación p_n?. Y relacionada con esta cuestión, ¿el polinomio de interpolación es único?
- ¿Existen formas óptimas para cálcular este polinomio?
- ¿Cómo aproxima p_n a esta función f?

y como cuestión asociada

- ¿Cuál es la mejor forma de evaluar un polinomio? Es decir, ¿Nos interesa el polinomio escrito de una forma especial o simplemente poder evaluar de forma fácil, rápida y estable?

5.2. Existencia del polinomio de interpolación

Probaremos la existencia del polinomio de interpolación mediante un razonamiento directo. Tomemos

$$\mathbb{P}_n \ni p_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

un polinomio de grado n. Obviamente, p_n cuenta con $n+1$ parámetros libres que son simplemente sus coeficientes $\{a_j\}$. Si exigimos que

$$p_n(x_j) = y_j, \qquad j = 0, \ldots, n$$

nos encontramos con que los coeficientes satisfacen el sistema lineal

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{5.1}$$

Por tanto el problema se reduce a la resolución de un sistema de ecuaciones lineales y la existencia y unicidad del polinomio de interpolación a que el sistema en cuestión sea compatible determinado.

El sistema anterior es una matriz de tipo Vandermonde que nos ha surgido repetidas veces en estos apuntes[1]. Dado que el sistema es cuadrado, tiene el mismo número de ecuaciones que de incógnitas, se tiene que

la existencia de solución para cualquier dato es equivalente a la unicidad.

Además, la unicidad de solución es equivalente a que la única solución posible para el término independiente nulo sea el polinomio cero. Pero esto es inmediato puesto que todo polinomio no nulo de grado n tiene a los sumo n raíces.

Otra forma de ver la existencia y unicidad es de tipo constructiva. Tomemos

$$L_j(x) := \frac{(x-x_0)\cdots(x-x_{j-1})(x-x_{j+1})\cdots(x-x_n)}{(x_j-x_0)\cdots(x_j-x_{j-1})(x_j-x_{j+1})\cdots(x_j-x_n)} = \prod_{i\neq j} \frac{x-x_i}{x_j-x_i}.$$

Es fácil ver que $L_j \in \mathbb{P}_n$ y que además

$$L_j(x_i) := \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

Por tanto,

$$p_n(x) = y_0 L_0(x) + y_1 L_1(x) + \ldots + y_n L_n(x).$$

satisface las condiciones (5.1). Una vez probada la existencia deducimos por los mismos argumentos la unicidad del polinomio de interpolación.

La fórmula anterior se conoce como **fórmula de Lagrange del polinomio de interpolación** y a la base $\{L_j\}_j$ **base de Lagrange del problema de interpolación**.

Ejercicio 5 Programar una función que evalúe el polinomio de interpolación en un conjunto de puntos según el siguiente prototipo

[1] Cada columna es el resultado de elevar a una potencia el vector $(1 \ x_0 \ x_1 \ \cdots \ x_n)^\top$

```
% LAGRANGEP
%
% LAGRANGEP(X0,Y0,X)  evalua el polinomio que interpola en (x0,y0)
%                     en x mediante la formula de Lagrange
%
% X0 y Y0 son dos vectores de la misma longitud, X puede ser un
% vector.
```

□

Solución. He aquí una posible implementación[2]

```
01    % LAGRANGEP
02    %
03    % LAGRANGEP(X0,Y0,X)  evalua el polinomio que interpola en (x0,y0)
04    %                     en x
05    %
06    % X0 y Y0 son dos vectores de la misma longitud, X puede ser un
07    % vector.
08
09    function y=lagrangep(x0,y0,x);
10
11    x0=x0(:).'; y0=y0(:).'; x=x(:).';  % todos vectores filas
12    n=length(x0);
13    if (length(y0)~=n)
14          disp('ERROR. Long de x0 debe ser igual a Long de y0')
15          return
16    end
17    y=zeros(size(x));  % y es un vector nulo de igual dimension que x
18
19    for j=1:n
20          p=ones(size(x));
21          for i=[1:j-1 j+1:n]
22                p=p.*(x-x0(i))./(x0(j)-x0(i));
23          end
24          y=y+p*y0(j);
25    end
26
27    return
```

Hemos probado el programa anterior para interpolar la función $\exp(\sin(6x))$ en $[0, \pi]$ en diversos puntos uniformemente distribuidos (a igual distancia). El resultado, junto con el error que comente el error de interpolación, se muestra en la figura 5.1.

Ejercicio 6 Implementa la construcción del polinomio de interpolación mediante la resolución directa del sistema dado en (5.1). ¿Qué observas cuando el grado del polinomio crece?.

□

[2] Observa el `for` de la línea 21

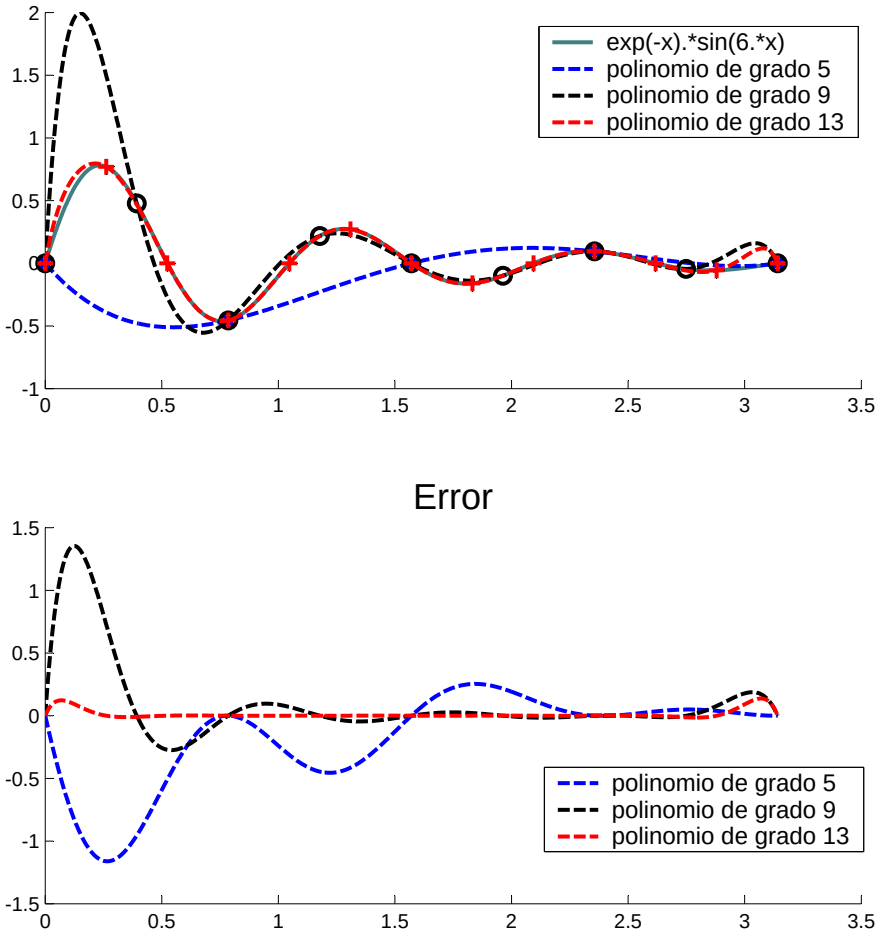


Figura 5.1: Polinomios de interpolación con diferentes grados

5.3. Fórmula de Newton

Es fácil comprobar que la fórmula de Lagrange tiene un costo computacional elevado. Por ello es necesario explorar formas alternativas de construir el polinomio de interpolación y de proceder a su evaluación.

Si observamos detenidamente la fórmula de Lagrange, ésta se basa en tomar como base de $\mathbb{P}_n$, los polinomios de grado n, la dada por

$$\{L_0(x), L_1(x), \ldots, L_n(x_n)\}.$$

En esta base, las coordenadas del polinomio de interpolación son la solución de un sistema que es diagonal (de hecho es la identidad) lo que hace que la resolución del sistema sea inmediata. El precio que se paga como contrapartida es una evaluación más cara del polinomio de interpolación.

Podemos explorar otras bases que, aumentando el costo de la resolución del sistema, den fórmulas del polinomio de interpolación cuya evaluación sea más barata.

Planteamos así utilizar la base

$$\{1, (x-x_0), (x-x_1)(x-x_0), \ldots, (x-x_{n-1})(x-x_{n-2})\cdots(x-x_0)\}.$$

Es inmediato probar que es una base de los polinomios de grado n. Dicho de otra forma, que cualquier polinomio de grado n se puede escribir

$$\alpha_0 + \alpha_1(x-x_0) + \ldots + \alpha_n(x-x_{n-1})(x-x_{n-2})\cdots(x-x_0). \tag{5.2}$$

Además el sistema lineal que daría los coeficientes $(\alpha_j)_j$ es ahora triangular superior, por lo que su resolución es prácticamente directa en $\mathcal{O}(n^2)$ operaciones.

Nótese que si $p_n(x)$ interpola a f en $n+1$ puntos, añadir un punto más (x_{n+1}, y_{n+1}) (donde $y_{n+1} = f(x_{n+1})$) es simplemente corregir el polinomio anterior en la forma siguiente

$$p_{n+1}(x) = p_n(x) + \alpha_{n+1}(x-x_0)\cdots(x-x_n),$$

concretamente

$$\alpha_{n+1} = \frac{y_{n+1} - p_n(x_{n+1})}{(x_{n+1}-x_0)\cdots(x_{n+1}-x_n)} \tag{5.3}$$

Es decir, el trabajo hecho para calcular el polinomio de interpolación en $n+1$ puntos se puede utilizar si se desea añadir un punto más de interpolación.

5.3.1. Diferencias divididas

Siguiendo la notación clásica, consideraremos que los valores y_j que deseamos interpolar provienen de una función f a priori desconocida, es decir,

$$y_j = f(x_j), \quad j = 0, \ldots, n.$$

Los coeficientes del polinomio de interpolación dado en (5.4) se denotan mediante

$$f[x_0] + f[x_0, x_1](x-x_0) + \ldots + f[x_0, x_1, \ldots, x_n](x-x_{n-1})(x-x_{n-2})\cdots(x-x_0). \tag{5.4}$$

y se denominan **diferencias finitas**. Esta forma de escribir el polinomio se conoce como **fórmula de Newton del polinomio de interpolación**[3].

Queda aún pendiente como calcular las diferencias divididas de forma que evitemos trabajar con el sistema lineal o con (5.3). Es inmediato comprobar que si

$$\left|\begin{array}{l} p_n \text{ interpola en } \{(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n))\}, \\ \\ q_n \text{ interpola en } \{(x_1, f(x_1)), (x_2, f(x_2)), \ldots, (x_{n+1}, f(x_{n+1}))\} \end{array}\right.$$

entonces

$$p_{n+1}(x) = q_n(x) + \frac{x - x_{n+1}}{x_0 - x_{n+1}}(p_n(x) - q_n(x)) \tag{5.5}$$

es el polinomio que interpola en los $n+1$ puntos

$$\{(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_{n+1}, f(x_{n+1}))\}$$

[3] De nuevo los nombres utilizados dan idea de la antigüedad de estas técnicas.

Detengámonos un momento a examinar el coeficiente director de cada polinomio. El coeficiente en x^n de p_n y q_n y el de x^{n+1} de p_{n+1} son respectivamente

$$f[x_0,\ldots,x_n], \qquad f[x_1,\ldots,x_n] \qquad f[x_0,\ldots,x_{n+1}].$$

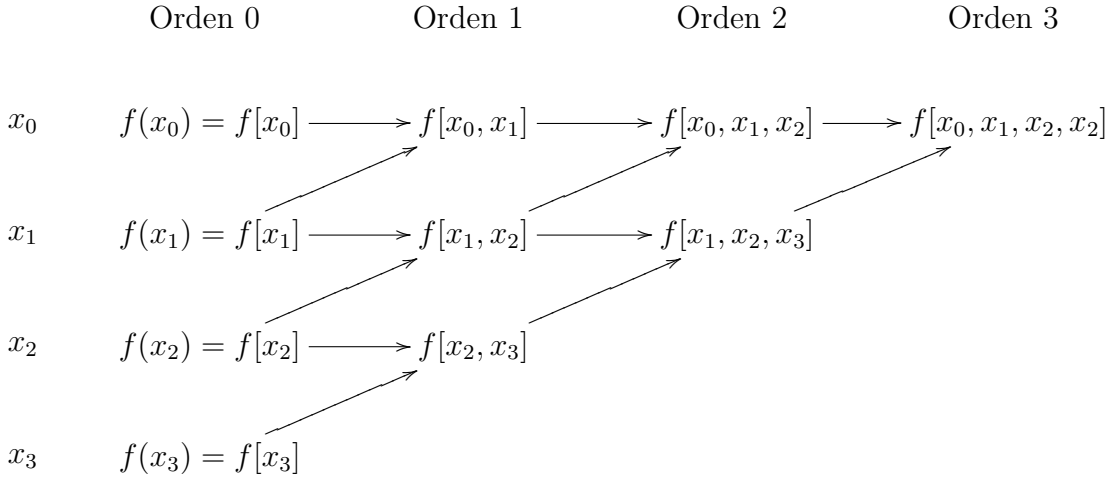
Utilizando (5.5) deducimos que

$$f[x_0,x_1\ldots,x_{n+1}] = \frac{f[x_0,\ldots,x_n]-f[x_1,\ldots,x_{n+1}]}{x_0-x_{n+1}}, \tag{5.6}$$

que permite expresar la diferencia dividida de $n+2$ puntos en término de dos de $n+1$ puntos. Finalmente, como[4]

$$f[x_j] = f(x_j)$$

obtenemos un algoritmo recursivo que nos permite calcular las diferencias divididas de cualquier orden. El algoritmo se puede dibujar en forma de árbol



donde

$$f[x_i,x_{i+1},\ldots,x_{i+j}] = \frac{f[x_i,\ldots,x_{i+j-1}]-f[x_{i+1},\ldots,x_{i+j}]}{x_i-x_{i+j}}$$

se calcula utilizando (5.6). En forma de pseudocódigo, el algoritmo sería el siguiente.

Diferencias divididas

```
01  x0, y0 % datos
02  n = length(x0) − 1
03  D(0, :) = y0;
04  for j=1:n
05      for i=0:n-j
06          D(i, j) = (D(i, j − 1) − D(i + 1, j − 1)) / (x0(i) − x0(i + j))
07      end
08  end
09
```

[4]Simplemente porque el polinomio de grado 0 que pasa por $(x_j, f(x_j))$ es el polinomio constante $f(x_j)$

```
10  % Evaluacion del polinomio
11  y=D(n,0)
12  for i=n-1:-1:0
13      y = (x − x0(i)). ∗ y + D(i, 0)
14  end
```

Con la notación anterior

$$\mathtt{D(i,j)} = f[x_{\mathtt{j}}, x_{\mathtt{j}+1}, \ldots, x_{\mathtt{j}+\mathtt{i}}]$$

de forma que `i` es el orden de la diferencia y `j` el punto desde *donde empieza*.

Ejercicio 7 Implementa el cálculo del polinomio de interpolación mediante diferencias divididas según el siguiente prototipo

```
01    % NEWTONP
02    %
03    % NEWTONP(X0,Y0) devuelve el polinomio que interpola en (X0,Y0)
04    %                 en su forma de Newton
05    %
06    % NEWTONP(X0,Y0,X) evalua el polinomio en x
```

□

Solución. Comentaremos antes algunos detalles. La orden

```
>> syms x
```

define en Matlab una variable simbólica. Esto es, `x` es un objeto matemático `x`, una entidad propia, de forma que,

```
>> x^2

ans =

x^2
```

Así se pueden realizar manipulaciones algebraicas con toda libertad. Matlab cuenta con una *toolbox* dedicada al cálculo simbólico[5]. Así,

```
>> int(x)  %integral de x

ans =
```

[5]Probablemente los mejores programas de cálculo simbólico sean Maple y Mathematica. Matlab se centra en el numérico por lo que el manejo de expresiones simbólica entra algo forzado, sobre todo en la sintaxis que es algo engorrosa. De todas formas, desde Matlab se puede llamar a cualquier comando de Maple por lo que en teoría todo lo que se puede hacer en Maple también se puede hacer con Matlab aunque no sea de una forma tan cómoda.

```
1/2*x^2

>> int(cos(x)*sin(x))  % integral de cos(x)*sin(x)

ans =

-1/2*cos(x)^2

>> diff(exp(cos(x)),x) % derivada de exp(cos(x)) respecto a x

ans =

-sin(x)*exp(cos(x))

>> int(x^2,0,2)        % integral definida

ans =

8/3

>> solve('x^2+2*x-2=0','x')   % ecuacion

ans =

[ -1+3^(1/2)]
[ -1-3^(1/2)]
```

Nuestra intención es que, si no se especifican valores donde evaluar el polinomio, devuelva el polinomio, escrito en forma de Newton utilizando una variable simbólica.

Por otro lado y entrando ya en el tema de su implementación, en el algoritmo anterior todas las entradas se numeran de `0` a `n`. Hay que tener en cuenta, como ya hemos hecho repetidas veces, que la numeración en Matlab de vectores y matrices comienza en 1 y por tanto los vectores y bucles irán de 1 a `n` donde `n` es ahora la **longitud del vector de datos** y de puntos de interpolación.

Dicho esto, una implementación, posible como tantas otras, es la siguiente

```
01    % NEWTONP
02    %
03    % NEWTONP(X0,Y0) devuelve el polinomio que interpola en (X0,Y0)
04    %                 en su forma de Newton
05    %
06    % NEWTONP(X0,Y0,X) evalua el polinomio en x
07
08    function y=newtonp(x0,y0,varargin)
09
10    x0=x0(:).'; y0=y0(:).'; n=length(x0);
```

```
11    if (length(y0)~=n)
12         disp('long de x0 debe ser igual long de y0')
13         return
14    end
15    if nargin>2
16         x=varargin{1};   % evaluamos x
17    else
18         syms x;           % x es una variable simbolica
19    end
20    % calculo de las diferencias divididas
21    D=zeros(n);   % matriz  n x n de ceros
22    D(:,1)=y0.';
23    for j=2:n
24         for i=1:n-j+1
25              D(i,j)=(D(i,j-1)-D(i+1,j-1))/(x0(i)-x0(i+j-1));
26         end
27    end
28
29    % evaluacion del polinomio
30    D=D(1,:);     % nos quedamos con las diferencias que vamos a usar
31    y=x.^0.*D(n); % asi y tiene la longitud de x
32    for i=n-1:-1:1
33          y=(x-x0(i)).*y+D(i);
34    end
35    return
```

Ejercicio 8 El bucle interno en las líneas 24–26 se puede vectorizar. Hacedlo.

□

Ejercicio 9 Un análisis más detenido muestra que no es necesario definir toda una matriz `D` sino que es posible realizar todo el proceso con un vector. Para ello observa que las diferencias $\mathtt{D}(i,j)$ con $\mathtt{j} > 0$ sólo se utilizan dos veces, para construir $\mathtt{D}(\mathtt{i}-1,\mathtt{j}+1)$ y $\mathtt{D}(\mathtt{i},\mathtt{j}+1)$. Retoca el algoritmo y programa el método resultante.

□

Ayuda. Basta modificar las líneas `04--08` del algoritmo como sigue

Modificación

```
03      D = y0;
04      for j=1:n
05          for i=n:-1:j
06              D(i) = (D(i) - D(i-1)) / (x0(i) - x0(i-j))
07          end
08      end
```

Nótese que las diferencias se sobrescriben en el vector pero de una forma tal que un valor no es borrado hasta que ha sido utilizado. De ahí que el bucle en `05` vaya *hacia atrás*. Con esta modificación

$$\texttt{D(j)} = f[x_0, x_1, \ldots, x_{\texttt{j}}].$$

Implementar la evaluación del polinomio es ahora inmediato.

Nota

Observa que el método funciona aún cuando los puntos $x_0, \ldots, x_n$ no estén ordenados, aunque para una evaluación del polinomio más estable es mejor que así sea. Nótese que la definición de las diferencias divididas no depende del orden de los puntos

$$f[x_0, \ldots, x_i, \ldots, x_j, \ldots x_n] = f[x_0, \ldots, x_j, \ldots, x_i, x_n].$$

Se puede construir el polinomio mediante

$$f[x_n] + f[x_n, x_{n-1}](x - x_n) + \ldots + f[x_n, x_{n-1}, \ldots, x_0](x - x_1)(x - x_2)\cdots(x - x_n). \quad (5.7)$$

En este caso se habla de **diferencias regresivas**, mientras que en el caso anterior, el que hemos tratado, recibe el nombre de **diferencias progresivas**.

Modificar los algoritmos anteriores para trabajar con las diferencias regresivas es un trabajo sin demasiada complicación.

Ejercicio 10 Hacedlo.

□

Interpolación en Matlab

La interpolación polinómica esta implementada en Matlab mediante dos comandos

```
polyfit        polyval
```

La primera calcula el polinomio que se ajusta (*fit*) un conjunto de datos. La sintaxis es

```
>> p=polyfit(x0,y0,n)
```

donde `x0` y `y0` son los puntos a interpolar y n el polinomio utilizado. Si `n` se toma igual a `length(x0)` − `1`, devuelve el polinomio de interpolación mientras que para `n` menor calcula el polinomio que aproxima a los puntos por mínimos cuadrados.

La orden anterior devuelve un vector (`p`) con los coeficientes del polinomio. Su evaluación se lleva a cabo ahora mediante

```
>> polyval(p,x)
```

donde `x` es un vector de puntos donde se desea evaluar el polinomio.

5.4. Análisis del error

El error entre función y polinomio de interpolación se puede estudiar utilizando algunas propiedades de las diferencias divididas que detallaremos más adelante. En cualquier caso, el resultado relevante es el siguiente: si f es la función regular que interpolamos en $\{x_0, x_1, \ldots, x_n\} \subset [a, b]$, entonces el error en cada punto se puede escribir

$$f(x) - p_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \underbrace{(x - x_0)(x - x_1) \cdots (x - x_n)}_{\omega_n(x)} \tag{5.8}$$

donde $\xi \in [a, b]$. Vemos que hay varias contribuciones al error final:

- El factor $1/n!$ que tiende rápidamente a 0.
- Un factor que depende del crecimiento de la derivada $n + 1$.
- Un término que depende de la colocación de los nodos de interpolación y que puede ser muy oscilante.

En vista de lo anterior es fácil concluir que hay convergencia del polinomio de interpolación a la función si ésta es suave. Dicho de otra manera, función y polinomio se vuelven indistinguibles si tomamos un número elevado de puntos.

Desafortunadamente[6], **la conclusión anterior es falsa**. Uno de los primeros (contra)ejemplos lo encontró Runge a finales del siglo XIX[7]. Se trata de interpolar la función

$$f(x) = \frac{1}{1 + 25x^2}$$

en el intervalo $[-1, 1]$ en un conjunto uniforme de puntos. El resultado lo vemos en la figura 5.2.

Una forma de reducir el error es controlar la función $\omega_n(x)$ y hacerlo lo más pequeño posible. Así, el problema se plantea en los siguientes términos[8]

> ¿Qué elección de los nodos $\{x_0, x_1, \ldots, x_n\}$ hace que $\max_x |\omega_n(x)|$ sea mínimo?

Éste es un problema clásico, fue resuelto por Chebyshev[9] en término de las raíces de una familia de polinomios conocidos como polinomios de Chebyshev de primer tipo.

[6] Y contraintuitivamente. ¿Cómo entender si no que dos funciones derivables que coincidan en un número cada vez mayor de puntos no acaben siendo muy similares?

[7] Éste fue el siglo de la fundamentación de las matemáticas. Multitud de resultados que se tenía como evidentes se mostraron falsos y fue necesario reconstruir las matemáticas desde su base. Por ejemplo, en el mismo siglo se mostró la existencia de funciones no continuas en ningún punto, o funciones continuas no derivables en ningún punto o con un número infinito de máximos y mínimos locales, se construyeron curvas que rellenaban el plano, o curvas cerradas que englobando una área finita tenían una longitud infinita (fractales)... Algunas de estas construcciones, originarias esencialmente en las matemáticas puras han encontrado en el siglo XX aplicaciones a problemas de índole esencialmente práctico.

[8] Un problema **min-max**: minimizar un máximo.

[9] Matemático ruso del siglo XIX. Se encontró con este problema cuando estudiaba los primeros motores de vapor. Concretamente, la transformación de un movimiento perpendicular (los pistones) en un movimiento rotatorio (la rueda).

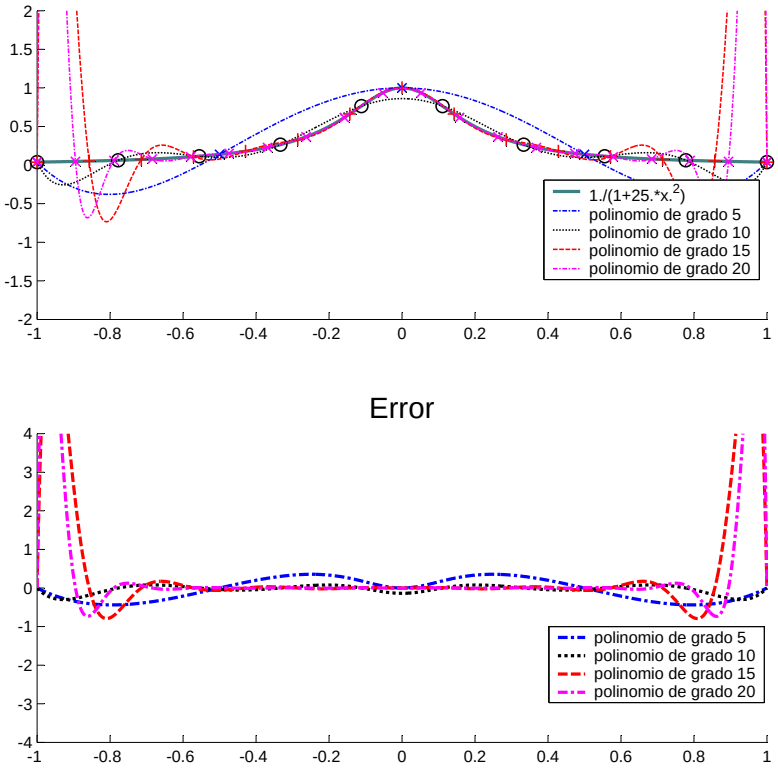


Figura 5.2: Ejemplo de Runge

Para un intervalo arbitrario la disposición de los $n+1$ puntos de interpolación es la dada

$$\left\{ \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{\pi}{2n+2} + \frac{k\pi}{n+1} \right) \right\}_{k=0}^{n} \tag{5.9}$$

Es fácil comprobar que la distribución de puntos no es en medida alguna uniforme sino que tiende a concentrar puntos en los extremos.

Tomar los puntos así, **no asegura** la convergencia del polinomio de interpolación pero sí permite controlar mejor uno de los términos que más rápido pueden crecer con n. De hecho con esta elección,

$$\max_{x\in[a,b]} \omega_n(x) = \frac{1}{2^n}$$

con lo que hacemos tender $\omega(x) \to 0$ cuando $n \to \infty$. En la figura 5.3 observamos como ahora el polinomio interpolante para el ejemplo de Runge converge a la función.

Ejercicio 11 Elegir un intervalo arbitrario $[a, b]$ y realizar un fichero script que muestre la disposición de los puntos dados por (5.9). Dibuja para diferentes valores de n. Dibuja también el polinomio $\omega_n(x)$. ¿Qué observas?

□

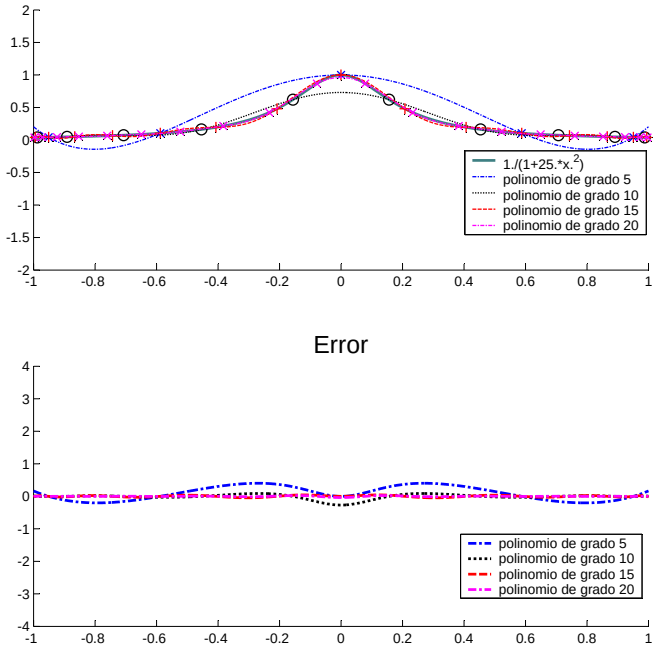


Figura 5.3: Ejemplo de Runge con otros nodos

Ejercicio 12 Se trata de probar una serie de propiedades de las diferencias divididas de las que finalmente se deduce la estimación del error del polinomio de interpolación dada en (5.8).

(1) Probar que $f^{(j)}(x) - p^{(j)}(x)$ se anula en $n+1-j$ puntos para $j = 0, \ldots, n+1$.

(2) Utilizando (1), probar que existe ξ tal que

$$f[x_0, \ldots, x_n] = \frac{f^{(n)}(\xi)}{n!}.$$

(3) Fijemos $y \in [a, b]$. Entonces

$$f(y) = p_n(y) + f[x_0, \ldots, x_n, y](x - x_0)(x - x_1) \cdots (x - x_n).$$

Utilizad ahora (2) para probar la estimación del error en (5.8)

Ayuda: El **Teorema de Rolle** establece que si g es derivable en $[a, b]$ con $g(a) = g(b)$ entonces existe $c \in (a, b)$ tal que

$$g'(c) = 0.$$

Aplicando este resultado, existe $\alpha_j \in (x_j, x_{j+1})$ $(j = 0, \ldots, n-1)$ tal que

$$f'(\alpha_j) - p'(\alpha_j) = 0.$$

Se puede aplicar el mismo resultado para probar que existe $n-2$ puntos tales que $f''(\beta_j) = p''(\beta_j)$ y así sucesivamente.

□

Capítulo 6

Extensiones adicionales

6.1. Interpolación de Hermite

La interpolación de Hermite toma como datos adicionales valores de la derivadas. Concretamente

$$\begin{cases} (x_0, f(x_0)), \ldots, (x_0, f^{(m_0)}(x_0)) \\ (x_1, f(x_1)), \ldots, (x_1, f^{(m_1)}(x_1)) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ (x_n, f(x_n)), \ldots, (x_n, f^{(m_n)}(x_n)) \end{cases}$$

Observemos que no puede haber *huecos* en los datos. Por ejemplo, si la derivada de orden 2 es dato, también lo es el valor de la función y su primera derivada. El número total de datos es

$$N := (m_0 + 1) + (m_1 + 1) + \cdots + (m_n + 1)$$

por lo que el polinomio debería tener grado $N - 1$.

El análisis de este problema no tiene mayor dificultad que el ya realizado para la interpolación polinómica de Lagrange. La clave está en la fuerte relación entre diferencias divididas y derivada, mostrada en el ejercicio 12. Así, una derivada se interpreta básicamente como un nodo repetido.

Entrando más en detalle, la idea es ordenar los nodos tomarlos repetidos según el número de datos que consideramos en ese punto. Así, consideramos

$$\{\underbrace{x_0, x_0, \ldots, x_0}_{m_0+1}, \underbrace{x_1, x_1, \ldots, x_1}_{m_1+1}, \ldots, \underbrace{x_n, x_n, \ldots, x_n}_{m_n+1}\}$$

y definir las diferencias divididas como sigue:

$$f[x_0, x_1, \ldots, x_m] := \begin{cases} \dfrac{f^{(m)}(x_0)}{m!}, & \text{si } x_0 = x_m \\ \dfrac{f[x_0, x_1, \ldots, x_{m-1}] - f[x_1, x_1, \ldots, x_m]}{x_0 - x_m}, & \text{en otro caso} \end{cases}$$

El polinomio de interpolación se construye ahora como antes:

$$f[x_0] + (x - x_0)\Big(f[x_0, x_1] + (x - x_1)\Big(f[x_0, x_1, x_2] + (x - x_2)\Big(\ldots \\ +(x - x_{n-1}) f[x_0, x_1, \ldots, x_n]\Big)\Big)\Big)$$

Diferencias divididas para la interpolación Hermite

```
01  x0, y0 datos
02  n = length(x0) − 1
03  D = y0;
04  for j=1:n
05      for i=n:-1:j
06          if x0(i) == x0(i − j)
07              D(i) = D(i)/j
08          else
09              D(i) = (D(i) − D(i − 1)) / (x0(i) − x0(i − j))
10          end
11      end
12  end
13
14  % Evaluacion del polinomio
15  y=D(n);
16  for i=n-1:-1:0
17      y = (x − x0(i)). ∗ y + D(i)
18  end
```

Nótese que en el algoritmo anterior la tabla de diferencias divididas se almacena de forma compacta, como se sugirió en el Ejercicio 9.

Ejercicio 13 Entiende bien el algoritmo anterior. Implementa la interpolación de Hermite según el siguiente prototipo

```
01    HERMITE
02
03    HERMITE(X0,Y0)   Calcula el polinomio de interpolacion de Hermite
04                     Los Nodos X0 deben estar ordenados. Nodos repetidos
05                     se interpretan como  derivadas sucesivas en ese
06                     punto
07
08    HERMITE(X0,Y0,X) Evalua el polinomio resultante en X
```

siguiendo las ideas expuestas en el ejercicio 7.

□

Observa que, una vez implementada esta función,

```
>> x0=[0 0 0 0 0 0]; y0=[1 1 1 1 1 1];
>> p=hermitep(x0,y0)
```

devuelve de hecho el polinomio de Taylor de orden 5 de $f(x) = \exp(x)$ en el cero. La orden

```
>> simple(hermite(x0,y0))
```

hace más evidente esta identidad.

6.2. Interpolación de funciones periódicas

Si una función es 2π-periódica es natural reemplazar en el problema de interpolación los polinomios por funciones periódicas. Concretamente, podemos utilizar

$$\sin(nx), \quad \cos(nx), \qquad n \in \mathbb{N} \cup \{0\}$$

o bien,

$$\exp(\mathrm{i}nx), \qquad n \in \mathbb{Z}.$$

Las combinaciones de las funciones anteriores se denominan polinomios trigonométricos. Ambas son equivalentes, pero la utilización de exponenciales complejas facilita tanto el análisis como la manipulación así como la extensión de la interpolación a funciones complejas

Supongamos que tomamos un número impar $2n+1$ de puntos de interpolación equidistantes:

$$x_k = \frac{2k\pi}{2n+1}, \qquad k = 0, \ldots, 2n.$$

El problema queda así definido: calcular

$$p_{2n+1}(x) := \sum_{j=-n}^{n} \alpha_j \exp(\mathrm{i}nx)$$

tal que

$$y_k = p_{2n+1}(x_k) = \sum_{j=-n}^{n} \alpha_j \exp\Big(\frac{2\pi \mathrm{i} jk}{2n+1}\Big).$$

De esta forma, el problema se reduce a utilizar la transformada de Fourier Discreta, y particularmente la FFT.

Es interesante señalar que el interpolante converge a la función bajo hipótesis mucho más débiles. Por ejemplo, basta con que la función sea derivable.

Ejercicio 14 Comprobar como la interpolación con exponenciales trigonométricas y con senos y cosenos son equivalentes. Para ello escribir el polinomio trigonométrico en senos y cosenos, en función de exponenciales complejas y viceversa.

□

Ejercicio 15 Comprueba que efectivamente se reduce todo a una transforma de Fourier discreta.

□

Ayuda Observa que

$$\exp\Big(-\frac{2\pi \mathrm{i} jk}{2n+1}\Big) = \exp\Big(\frac{2\pi \mathrm{i}(2n+1-j)k}{2n+1}\Big).$$

Por tanto el problema es equivalente a encontrar α_j tales que

$$y_k = p_{2n+1}(x_k) = \sum_{j=0}^{n} \alpha_j \exp\Big(\frac{2\pi \mathrm{i} jk}{2n+1}\Big) + \sum_{j=n+1}^{2n} \alpha_{2n+1-j} \exp\Big(\frac{2\pi \mathrm{i} jk}{2n+1}\Big).$$

Ejercicio 16 ¿Qué pasa si se toman un número par de puntos de interpolación?

□

6.3. Interpolación polinómica a trozos

Los polinomios tienen un importante inconveniente que desaconseja su utilización para un número elevado de puntos. Obsérvese la figura 6.1, donde el polinomio exhibe un comportamiento ciertamente caótico. Ello se debe a que se *obliga* a que el polinomio sea prácticamente constante en dos zonas con un salto en medio. El polinomio es excesivamente **rígido**[1] que en última medida provoca que éste se *rompa*.

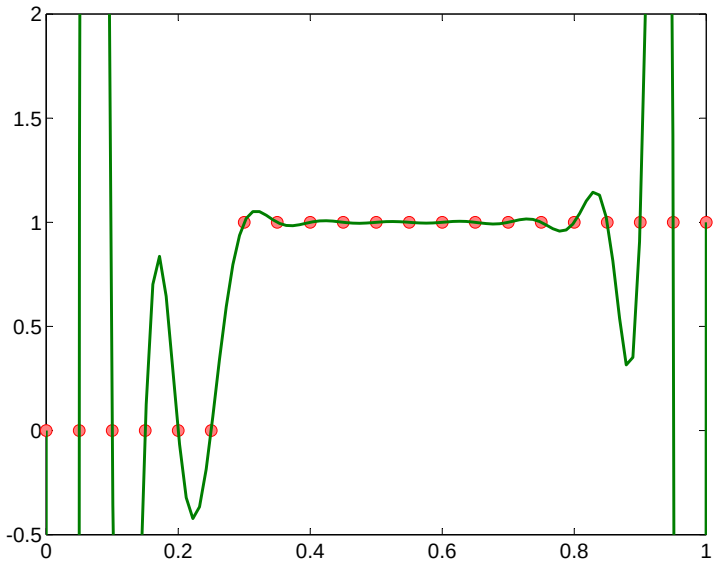


Figura 6.1: Rigidez polinómica

Como forma de solventar estos problemas se opta por utilizar elementos más complicados para interpolar pero que sean más *flexibles*.

Uno de los elementos más populares son los *splines*. Un spline[2] es una función polinómica a trozos que tiene cierta regularidad. Es decir, los diferentes polinomios se unen exigiendo además sean continuas y derivables hasta cierto orden. Los representantes más sencillos son las constantes a trozos y las poligonales. En este último caso, se trata sencillamente de unir los puntos por segmentos obteniéndose una curva poligonal. Sin embargo para las constantes a trozos no podemos exigir continuidad y para las poligonales no podemos exigir derivabilidad en los puntos de contacto (tiene picos).

Los splines cúbicos son probablemente los más utilizados. Recurriendo a polinomios de grados 3 se trata de construir una función s_n tal que

$$s_n(x)|_{[x_i,x_{i+1}]} \in \mathbb{P}_3, \quad i = 0, \ldots, n-1$$

[1] Es un símil con lo que sucede si se dobla una vara muy rígida. El problema en utilizar polinomios es que basta modificar un sólo dato para que el polinomio se cambie en todo el dominio, en muchas ocasiones sin ningún control. Sin embargo, en un interpolante uno debería esperar que la modificación de un punto afectara sólo al entorno de dicho punto.

[2] aceptaremos el anglicismo a partir de ahora.

y que tanto la función como sus derivadas primera y segunda sean continuas[3]

$$s_n(x_i) = y_i, \qquad \lim_{x \to x_j^+} s_n^{(j)}(x) = \lim_{x \to x_j^-} s_n^{(j)}(x), \quad j = 0, 1, 2.$$

Es decir, sobre cada intervalo $[x_j, x_{j+1}]$ la función es un polinomio de grado tres (cúbico). Al tener la primera derivada continua, se preserva la existencia de recta tangente en todo punto y por tanto la *velocidad* de recorrido de la curva, mientras que con la continuidad de la derivada segunda podemos definir la curvatura de la curva siendo ésta continua en todo el dominio de definición.

Dado que tenemos n intervalos, y cuatro parámetros libres sobre cada intervalo (s_n es un polinomio de grado 3 en cada intervalo), disponemos de $4n$ parámetros libres. Por contra, los datos de interpolación fijan $n+1$ restricciones y las condiciones de continuidad del spline y de sus dos primeras derivadas en las interfaces de los subintervalos dan otras $3n$ condiciones. En total hacen

$$n + 1 + 3(n - 1) = 4n - 2$$

por lo que el problema precisa fijar dos condiciones adicionales para poder hablar, siquiera plantear, la unicidad del spline.

Estas son algunas de las elecciones posibles

- $s_n''(x_0) = s_n''(x_n) = 0$, **spline natural**.
- Datos adicionales $s_n'(x_0) = y_{-1}$, $s_n'(x_n) = y_{n+1}$ **spline grapado**.
- Condiciones de periodicidad: si los datos son periódicos se puede imponer $s_n'(x_0) = s_n'(x_n)$, $s_n''(x_0) = s_n''(x_n)$ **spline periódico**.

No nos detendremos a detallar como se prueba la existencia y unicidad de tales splines, ni la forma de calcularlos. Baste decir que se recurre a técnicas más eficientes, y por tanto complejas, que la forma directa que se deduce de su definición.

Entre las buenas propiedades de los splines destacamos que convergen a la función si ésta es continua. Aún es más, si la función f que se interpola es regular (de hecho basta con que su derivada cuarta sea continua)

$$\max_x |s_n(x) - f(x)| \leq C \max_j (x_{j+1} - x_j)^4$$

con C una constante dependiente de f pero independiente de los puntos.

Nota: splines en Matlab

Matlab dispone de dos comandos encargados de esta tarea:

```
ppval        spline
```

[3] Si forzáramos la continuidad de la tercera derivada tendríamos globalmente un polinomio de grado 3

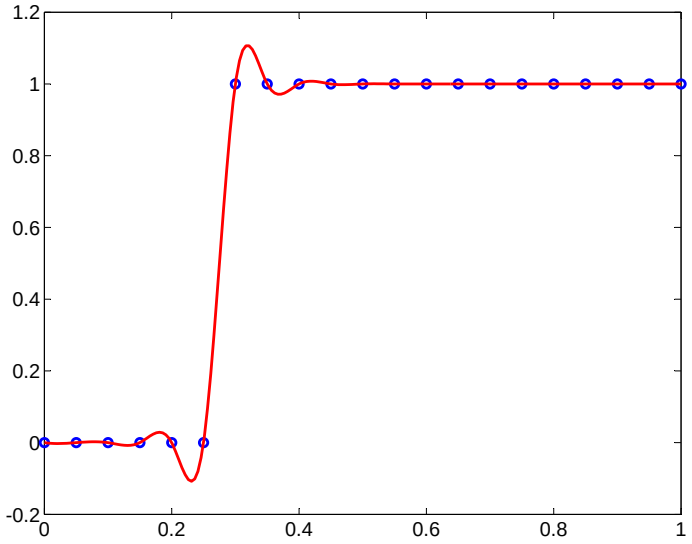


Figura 6.2: Interpolación con splines cúbicos naturales

El primero (*piecewise polynomial value*) evalúa una función polinómica a trozos. Uno de sus argumentos es la propia estructura de la función según una sintaxis propia de Matlab. La segunda instrucción devuelve una estructura de este tipo que contiene el spline que interpola a datos.

A modo de ejemplo, he aquí el spline natural que interpola a los datos desplegados en la figura 6.1[4]

```
>> x0=linspace(0,1,21);
>> y0=[zeros(1,6) ones(1,15)];
>> p1=spline(x0,y0);        % spline natural
>> x=linspace(0,1,200);
>> h=plot(x0,y0,'o',x,ppval(p1,x),'r','linewidth',2);
```

Por otro lado Matlab tiene implementado en una toolbox comandos que entran en detalles más finos para trabajar con splines. Ejecutando `splinetool` podemos acceder a un entorno gráfico donde testar las capacidades de Matlab en este campo.

Nota

Los splines tienen un origen muy natural. Existía una herramienta de dibujo, ya anticuada, que recibía el nombre de spline, o trazador en castellano. Consistía en una especie de regla hecha de un material flexible que se moldeaba para hacer coincidir un trazado por una serie de puntos. La curva así trazada era físicamente equivalente a resolver el problema de construir una curva que pase por una serie de puntos y que minimice la energía elástica. La solución matemática a este problema es un spline natural (polinomio cúbico a trozos con derivada segunda nula en los extremos) lo que justifica su denominación.

[4] Los splines siguen conservando cierta rigidez pero se adaptan mucho mejor a los datos.

Se pueden definir splines de mayor grado, siguiendo exactamente la idea anterior, esto es, pegando polinomios a trozos y dando la mayor regularidad posible.

Ejercicio 17 Se plantea el siguiente problema: dada una serie de puntos en el plano

$$\Big\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\Big\}$$

construir una curva en paramétricas

$$\mathbf{x}(t) := (x(t), y(t)), \qquad t \in [0, T] \tag{6.1}$$

tal que

$$\mathbf{x}(t_j) = (x_j, y_j). \tag{6.2}$$

Una buena elección es utilizar splines. Para ello se trata de calcular dos splines $\mathsf{cx}(t)$ y $\mathsf{cy}_n(t)$ que satisfagan, obviamente,

$$\mathsf{cx}_n(t_j) = x_j, \qquad \mathsf{cy}(t_j) = y_j.$$

Tenemos así dos problemas de interpolación desacoplados, esto es, cada uno se resuelve independiente del otro. Queda como cuestión pendiente la elección de T en (6.1) y de los puntos t_j en (6.2). Una buena elección es tomar

$$h_j := \sqrt{(x_{j+1} - x_j)^2 + (y_{j+1} - y_j)^2},$$

que coincide con la longitud del segmento que une (x_j, y_j) y (x_{j+1}, y_{j+1}), y hacer luego

$$t_0 = 0, \qquad t_k := h_0 + h_1 + \ldots + h_{k-1}, \qquad T = t_n.$$

Una vez ya calculados, la curva se dibuja evaluando

$$\mathbf{x}(t) := (\mathrm{cx}(t), \mathrm{cy}(t)), \qquad t \in [0, T].$$

Implementa un programa que siga este prototipo

```
01    SPLINE2D
02
03    SPLINE2D(X,Y) Traza una curva en el plano que pasa por
04                  (X(i),Y(i)) utilizando splines cubicos
05
```

□

Ejercicio 18 El comando `ginput` permite leer puntos a través del ratón. Lee bien la ayuda y modifica el programa anterior para que permita introducir los datos con el ratón en caso de que se llame a la función sin argumentos.

Entre las posibilidades que ofrece está la de controlar que tecla del teclado o botón del ratón se ha utilizado.

□

Ayuda. Deduce que realiza el siguiente código...

```
x0=[]; y0=[]; n=length(x0);
salir=0;
cla
axis([0 1 0 1])
while salir==0
    [xi,yi,b]=ginput(1);
    if b==3                % boton derecho del raton
        cla
        if n>0             % borramos el ultimo punto
            x0(n)=[]; y0(n)=[];
            plot(x0,y0,'ro')
            axis([0 1 0 1])
            n=n-1;
        end
    elseif  isempty(b)  % Se ha pulsado return: finalizamos lectura
        salir=1;
    elseif  b==2        % boton central: finalizamos la lectura
        salir=1
    else                % introducimos el punto
        x0=[x0 xi]; y0=[y0 yi];
        plot(x0,y0,'o')
        n=n+1;
        axis([0 1 0 1])
    end
end
```

6.4. Curvas Bezier

Las curvas Bezier son curvas en el plano o espacio que quedan determinadas por un conjunto de puntos. La curva en cuestión sólo pasa por el punto inicial y final, pero no por el resto de puntos. Sin embargo estos puntos forman un *polígono de control* en el siguiente sentido: la curva está contenida en el polígono formado por esos puntos e imita la forma de dicho polígono.

Para construir la curva consideramos los polinomios de Bernstein de orden n:

$$B_j^n(t) = \binom{n}{j} t^j (1-t)^{n-j}, \qquad j = 0, 1, \ldots, n$$

Es fácil comprobar que los polinomios verifican las siguientes propiedades:

- $\{B_j^n\}_{j=0}^n$ son una base de $\mathbb{P}_n$. Dicho de otra forma, cualquier polinomio se puede escribir como una combinación de estos polinomios
- $\displaystyle\sum_{j=0}^{n} B_j^n(t) = 1.$

Las curvas Bezier se definen ahora como sigue: para los puntos

$$\mathbf{x}_i = (x_i, y_i), \quad \text{con} \quad i = 0, \ldots, n.$$

y construimos la curva

$$S(t) := \sum_{j=0}^{n} B_j^n(t)\mathbf{x}_j = \Big(\sum_{j=0}^{n} B_j^n(t)x_j, \sum_{j=0}^{n} B_j^n(t)y_j \Big), \qquad t \in [0,1].$$

El parámetro t se mueve de forma que para $t = 0$ estamos en el punto inicial y para $t = 1$, en el final. Nótese que los puntos utilizados para dibujar la curva **están ordenados**: hay un punto inicial $\mathbf{x}_0$, uno final $\mathbf{x}_n$ y a $\mathbf{x}_j$ le sigue $\mathbf{x}_{j+1}$...

Es fácil ver que $S(0) = \mathbf{x}_0$ y $S(1) = \mathbf{x}_n$. Podemos comprobar, figura 6.3, que la curva se *adapta* a la forma que marcan los puntos.

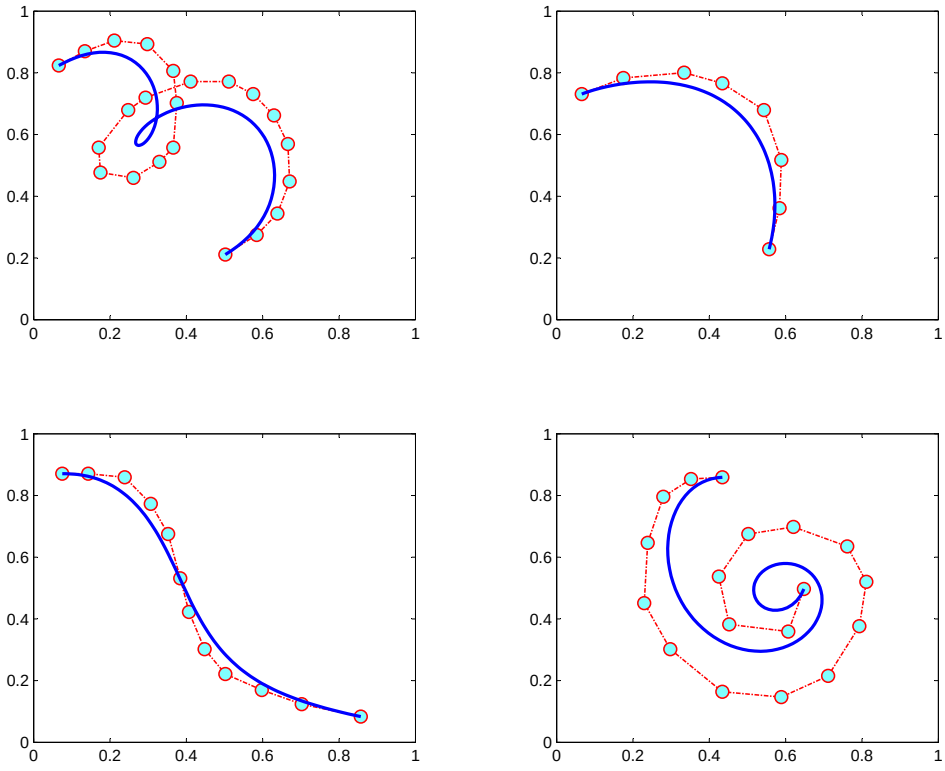


Figura 6.3: Curvas Bezier para algunas configuraciones de puntos

Ejercicio 19 Utilizando el código mostrado en el Ejercicio 18, programar las curvas Bezier. Comparar con las curvas que define la interpolación por splines y la polinómica.

□

Ejercicio 20 Implementar la siguiente extensión del programa anterior: permitir que el usuario seleccione puntos y pueda moverlos o borrarlos (por ejemplo, con el botón derecho del ratón) y así comprobar la sensibilidad de la curva al polígono de control.

Una sugerencia sería la seguiente: una vez leídos los puntos, y dibujada la correspondiente curva nos situamos de nuevo en la ventana gráfica con `ginput` y leemos la selección hecha por el ratón. Ahora hay tres opciones

- Si pulsamos *cerca* de un punto, entendemos que hemos seleccionado dicho punto.
  - Si se ha pulsado con el botón derecha lo eliminamos del dibujo, redibujamos la curva y esperamos de nuevo.
  - Si pulsamos *cerca* de un punto con el botón izquierdo entendemos que ese punto lo vamos a mover. Esperamos otra selección del ratón (otro `ginput`), reemplazamos el punto seleccionado por el nuevo y redibujamos
- Si pulsamos lejos de todos los puntos, entendemos que añadimos un nuevo punto. Llamémoslo $\mathbf{y}$. En este punto hay un tema no trivial: cómo ordenamos el punto $\mathbf{y}$ respecto a los puntes anteriores $\{\mathbf{x}_j\}$.

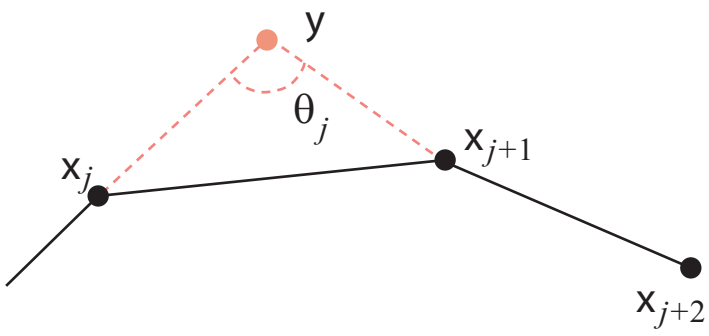


Figura 6.4: Ordenación de nodos para la curva Bezier

Una forma de determinar[5] este orden es medir el ángulo entre los vectores

$$\mathbf{u} = \overrightarrow{\mathbf{x}_j\mathbf{y}}, \qquad \mathbf{v} = \overrightarrow{\mathbf{x}_{j+1}\mathbf{y}}$$

y quedarse con aquel para el que ángulo θ_j sea mayor (véase la figura 6.4). Para ello, basta utilizar que

$$|\cos(\theta_j)| = \left|\frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}\right|.$$

tomar j para el que $|\cos(\theta_j)|$ es máximo, y construir la curva Bezier con puntos

$$\{\mathbf{x}_1, \ldots, \mathbf{x}_j, \mathbf{y}, \mathbf{x}_{j+1}, \ldots, \mathbf{x}_n\}$$

$\{\mathbf{x}_j, \mathbf{x}_{j+1}\}$

□

[5] Gracias a Jon García por sugerirnos esta solución.

Nota

Las curvas Bezier fueron introducidas por los ingenieros Bezier y Casteljou, que trabajaban entonces en Renault y Citröen. Concretamente, buscaban herramientas para el diseño de parachoques de forma que la curva que diera su forma fuera controlada por una serie de puntos y que tuviera buenas propiedades geométricas. Los algoritmos iniciales seguían ideas muy geométricas, que todavía se utilizan en la práctica, y se tardó algún tiempo en darle la forma que hemos mostrado, más matemática. Resultado de este estudio surgieron las curvas B-splines, que reemplazaban los polinomios de Bernstein por polinomios de grados adecuados. Las curvas B-splines son más flexibles que las curvas Bezier y en su caso extremos incluyen a éstas.

Por otro lado existen formas mucho más eficientes de evaluar y calcular dichas curvas que permiten en última media su implementación en una forma más interactiva, de forma que el usuario mueva los puntos (cambie el *polígono de control*) y que la curva se redibuje entonces.

Índice de figuras

Índice general