Actividades 9

Prácticas de Cálculo Numérico I (doble grado)

3 de mayo de 2022

1. Interpolación de Lagrange y baricéntrica

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

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

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

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

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

Descomponer el producto que define L_j en dos, es algo más eficiente que comprobar $n+1$ veces la condición $k \neq j$, por eso se ha escrito así. Por otro lado, conociendo el comando `prod`, cuyo uso se aclarará en una actividad posterior, es posible ganar en rapidez de ejecución. Como sugiere uno de los comentarios, los puntos antes de `*` en las líneas 9 y 12 son superfluos para calcular valores individuales, pero si los dejamos, servirá también para `t` vectorial, evitando llamadas múltiples.

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

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

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

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

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

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

y w_j el denominador de L_j, es decir, $w_j = \prod_{k \neq j} (x_j - x_k)^{-1}$. Hay dos ventajas principales en esta formulación. La primera es que al ser los w_j independientes de t, cada nueva evaluación solo requiere una cantidad de operaciones comparable a n. La segunda ventaja es que añadir nuevos nodos, es decir, variar n, no implica rehacer todos los cálculos desde cero, basta añadir un sumando a N y a D por cada nuevo nodo. Esta propiedad la comparte con el *método de diferencias divididas de Newton*, que también se verá en la teoría.

Una pequeña desventaja, meramente técnica, es que hay que tener cuidado con no aplicar la fórmula para $t = x_j$. Además de las ventajas anteriores, se tiene que para ciertas elecciones naturales de los nodos los w_j admiten fórmulas explícitas [BT04] que permiten una precomputación de sus valores.

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

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

De nuevo `t` puede ser vectorial.

No es difícil probar que para nodos equiespaciados en el intervalo $[a, b]$ se puede tomar [BT04, (5.1)]:

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

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

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

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

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

Si usamos el código tal como está, veremos que la línea continua se aleja un poco de los nodos inicial y final. Esto se debe a las singularidades en x_0 y x_n. Una manera, un poco chapucera, pero efectiva, de evitarlo, es

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

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

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

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

2. Polinomios e interpolación con `matlab/octave`

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

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

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

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

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

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

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

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

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

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

Seguro que te estás preguntando acerca de qué sentido tiene el argumento redundante `n` en `polyfit(x,y,n)`. La respuesta es que `polyfit` tiene una utilidad más general que la vista aquí en la que `n` indica el grado del

polinomio que ajusta los datos `x` e `y`. Si `n` es al menos la longitud de estos vectores menos uno, existe un polinomio que ajusta perfectamente, el de interpolación. Si `n` fuera menor, entonces lo que devuelve `polyfit` son los coeficientes del ajuste por mínimos cuadrados.

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

3. Fenómeno de Runge

Tomando muchos nodos que cubran bien un intervalo, uno esperaría una buena aproximación de cualquier función regular por medio del polinomio interpolador. Sin embargo, no es así. El ejemplo clásico es la función $f(x) = (1 + x^2)^{-1}$. A pesar de pertenecer a C^∞, su polinomio interpolador P_n correspondiente a $n + 1$ nodos equiespaciados en $[-5, 5]$ no cumple $\|f - P_n\|_\infty \to 0$, de hecho $\|f - P_n\|_\infty \to \infty$. Esto es lo que se llama *fenómeno de Runge*. Si tienes curiosidad en ver pruebas matemáticas del inesperado comportamiento del polinomio interpolador en este ejemplo, las puedes encontrar en [IK66, §3.4] y [Epp87].

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

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

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

En [Epp87] se muestra que la convergencia se recuperaría con otra elección de los nodos.

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

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

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

estos nodos vienen dados por la fórmula

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

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

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

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

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

El fenómeno de Runge para nodos equiespaciados no es universal y tiene que ver con cuestiones de variable compleja [Epp87].

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

En la actividad anterior se limita n a un valor no muy elevado para no entrar en problemas con el épsilon máquina. Desde el punto de vista teórico, no hay ninguna limitación. La explicación tiene que ver con que las derivadas sucesivas de f están acotadas.

4. Splines

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

El comando en `matlab/octave` para definir un polinomio a trozos es `mkpp(x,A)` donde `x` son los nodos en los que se cambia de polinomio y las

filas de `A` son los coeficientes de los polinomios suponiendo el origen en el nodo, esto significa que en vez de los coeficientes de $p(x)$ debemos usar los de $p(x+x_j)$. Para evaluar un polinomio definido a trozos `pt` en una lista de valores `t` se usa `ppval(pt,t)`. Por ejemplo, consideremos

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

Para dibujar su gráfica, podríamos usar

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

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

Dados $\big\{(x_j, y_j)\big\}_{j=0}^{n}$, en su versión más simple y empleada, la interpolación con *splines* (*cúbicos naturales*) es la que se consigue mediante una función f tal que $f\big|_{[x_j,x_{j+1}]}$ es un polinomio de grado menor o igual que tres, f, f', f'' son continuas y $f''(x_0) = f''(x_n) = 0$. Se puede probar que solo hay una función con tales propiedades. Además, las propiedades sobre f'' equivalen a que minimiza la norma 2 de la derivada segunda [Cha20, §2.1.3]. Esto es muy interesante en términos prácticos porque la derivada segunda tienen que ver con la curvatura y entonces los splines tienden a curvarse lo menos posible, en algún sentido, lo cual, como examinaremos a continuación, nos libra del comportamiento salvaje de la interpolación que hemos observado en $f(x) = 1/(1+x^2)$.

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

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

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

Para favorecer la comparación se incluye también lo obtenido con el polinomio de interpolación y los (x_j, y_j). Si añades la gráfica de $f(x) = 1/(1+x^2)$ verás que es indistinguible a simple vista de lo obtenido con la interpolación con splines. En [QS07, §3.3] puedes encontrar un programa que no usa la función nativa de `spline` y es un poco más flexible.

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

El cálculo de los coeficientes de los polinomios cúbicos que conforman los splines se reduce a resolver un sistema $n \times n$ [SB93, §2.4], pero es un sistema muy especial, con matriz tridiagonal, que requiere mucho menos esfuerzo computacional que uno genérico, lo cual añade la rapidez a las ventajas de los splines.

5. Curvas de Bézier y B-splines

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

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

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

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

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

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

Si ahora concatenamos esta curva con otra correspondiente a los puntos P_3, P_4, P_5 y P_6 de forma que P_4 esté alineado con P_2P_3, el resultado será derivable, no tendrá picos. La generalización a $P_0, \ldots, P_n$ con $n = 3k$ conforma lo que se llama una *curva de Bézier*. Estas se usan a menudo en programas de diseño gráfico interactivos en los que se permite mover los

puntos de control conservando la alineación para modificar la forma de la curva. Las propiedades anteriores hacen que el procedimiento sea bastante intuitivo y, con un poco de práctica, se logre aproximar bien cualquier curva con unos cuantos puntos de interpolación y control.

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

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

figure(1)
n = (size(P,1)-4)/3;
for j = 0:n
        r = 3*j+1:3*j+4;
        plot( P(r,1), P(r,2), 'o', P(r,1), P(r,2), '--')
        hold on
        t = linspace(0,1, 300)';
        R = [(1-t).^3, 3*t.*(1-t).^2, 3*t.^2.*(1-t),
                ↪ t.^3]*P(r,:);
        plot( R(:,1), R(:,2) )
        hold on
end
hold off
```

El código funciona con cualquier lista de puntos `P`, siempre con un número de filas del tipo $3k+1$.

ACTIVIDAD 9.5.1. *Dibuja la curva de Bézier que interpola* $P_{3j} = (3j, 0)$, $0 \le j \le 2J$ *con puntos de control* $P_{3j+1} = \big(3j+1, (-1)^j\big)$, $P_{3j+2} = \big(3j+2, (-1)^j\big)$. *Con este propósito, quizá te sea útil utilizar* `repmat(v,1,r)` *que aplicado a un vector fila* `v` *lo repite* `r` *veces multiplicando su dimensión por esta cantidad.*

Las curvas de Bézier no interpolan todos los P_j y los llamados *B-splines* exageran esta situación pues, genéricamente no interpolan ninguno de los puntos de partida, solo los aproximan.

Para motivar e introducir los B-splines, fijémonos solo en un caso particular, pero importante, en que tenemos puntos de interpolación $\big\{(x_j, y_j)\big\}_{j=0}^{n}$ con $x_{j+1} - x_j = h$ constante. Supongamos que n es gigantesco y que no estamos dispuestos a hacer el esfuerzo computacional de resolver el sistema requerido por los *splines*. Resulta que la función definida en $[x_0, x_{n-1}]$ mediante

$$f(t) = \sum_{k=1}^{4} Q_k\big((t-x_k)/h\big) y_{j+3-k} \qquad \text{para} \quad t \in [x_j, x_{j+1}], \quad 0 < j < n-1,$$

es un buen sustituto sencillo para los splines tomando

$$Q_1(t) = \frac{1}{6}t^3,\ Q_2(t) = \frac{1}{6}(t+1)^3 - \frac{2}{3}t^3,\ Q_3(t) = Q_2(1-t),\ Q_4(t) = Q_1(1-t).$$

Un cálculo muestra

$$f(x_j) = \frac{1}{6}\big(y_{j-1} + 4y_j + y_{j+1}\big).$$

Si h es pequeño y los y_j provienen de una función regular, los y_j presentarán variaciones pequeñas y se cumplirá aproximadamente $f(x_j) = y_j$. Aunque te resulte chocante, operaciones de este tipo están funcionando todos los días en las entrañas de nuestros dispositivos digitales cuando ampliamos una foto [Cha20, §2.1.3].

El siguiente código, no muy elegante, muestra el resultado para $n = 10$ y la función $y = \operatorname{sen} x$ en $[-\pi, \pi]$.

```
q1 = @(x) x.^3/6;
q2 = @(x) (x+1).^3/6-2*x.^3/3;
q3 = @(x) q2(1-x);
q4 = @(x) q1(1-x);

n = 10;
x = linspace(-pi,pi, n+1);
y = sin(x);
h = x(2)-x(1);

figure(1)
plot(x,y,'bo')
hold on
t = linspace(-pi,pi,100);
plot(t,sin(t), 'r--')
hold on

for j=1:n-2
        t = linspace(x(j+1), x(j+2),10 );
        tn = (t-x(j+1))/h;
        plot(t,q1(tn)*y(j+3) + q2(tn)*y(j+2) +
            ↪ q3(tn)*y(j+1) + q4(tn)*y(j))
        hold on
end

hold off
```

La función $b(t)$ definida por $Q_1(t+2)$, $Q_2(t+1)$, $Q_3(t)$ y $Q_4(t-1)$, respectivamente en $[-2,-1]$, $[-1,0]$, $[0,1]$ y $[1,2]$, y cero en el resto, es lo que se llama *B-spline*. Es un spline en el sentido de que es un polinomio cúbico a trozos con derivada segunda que además se anula en los extremos.

ACTIVIDAD 9.5.2. *Dibuja la gráfica de $b(t)$ en $[-2,2]$.*

Para terminar, algunos comentarios para el que tenga curiosidad en la nomenclatura y la historia.

Las curvas de Bézier deben su nombre al ingeniero francés que las usó extensivamente en el diseño de automóviles en la década de 1960. En realidad, desde el punto de vista matemático, su origen teórico está en los *polinomios de Bernstein*, que son bastante anteriores.

La función f construida a partir de los Q_k se puede expresar como combinación lineal de versiones convenientemente escaladas y trasladadas de b y, en conexión con esto, la terminología B-spline abrevia *basis spline* en inglés, pues a partir de b se obtiene una base del espacio vectorial de splines. Por otro lado, *spline* era el nombre que recibían las plantillas para trazar curvas usadas en ingeniería industrial. Hace años era habitual que los escolares tuvieran splines en este sentido, materializados en unas placas plásticas con formas caprichosas que llamaban plantillas de curvas (aunque, al parecer, el nombre técnico es *plantillas Burmester*), las cuales, usadas con pericia, permitían dibujar elipses y formas más complicadas. Posiblemente los gráficos con ordenador han hecho que ahora sean menos comunes.

Referencias

[BT04] J.-P. Berrut and L. N. Trefethen. Barycentric Lagrange interpolation. *SIAM Rev.*, 46(3):501–517, 2004.

[Cha20] F. Chamizo. A course on signal processing. `http://matematicas.uam.es/~fernando.chamizo/libreria/libreria.html`, 2020.

[Epp87] J. F. Epperson. On the Runge example. *Amer. Math. Monthly*, 94(4):329–341, 1987.

[IK66] E. Isaacson and H. B. Keller. *Analysis of numerical methods*. John Wiley & Sons, Inc., New York-London-Sydney, 1966.

[QS07] A. Quarteroni and F. Saleri. *Cálculo Científico con MATLAB y Octave*. Springer, Milan, 2007.

[SB93] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*, volume 12 of *Texts in Applied Mathematics*. Springer-Verlag, New York, second edition, 1993. Translated from the German by R. Bartels, W. Gautschi and C. Witzgall.