Práctica_3

November 24, 2022

1 Práctica 3

Alonso Oma Alonso

[ ]:
```
clear all
addpath('./Biblioteca')
graphics_toolkit ("gnuplot"); %% Comando solo para jupyter notebooks
```

Para realizar esta práctica necesitas descargar algunos script de Octave que he subido al Aula Virtual. Si no lo has hecho ya, descarga el archivo *Practica3_AnadirBiblioteca.zip*, situado en la carpeta CodigoPracticas de los Recursos del AV. Descomprímelo y guarda su contenido en la carpeta Biblioteca que debe estar contenida en tu carpeta de trabajo con Octave de tu ordenador personal. Este archivo contiene el código de todas las funciones descritas a continuación.

Nota. He modificado algunas funciones desde la última vez que las subí al Aula Virtual. Las modificaciones son menores (tendentes a «vectorializar» más intensamente el código). Puedes sustituir los archivos existentes por los nuevos o dejar los antiguos, ambos funcionan, aunque los nuevos deberían ser más eficientes.

El objetivo general de esta práctica es experimentar con la interpolación polinomial, tanto de Lagrange como de Hermite, con diversas funciones. Para ello, en Biblioteca dispones de las funciones siguientes

- **dif_divNewton.m**, devuelve la tabla de diferencias divididas y los coeficientes del polinomio interpolador en la forma de Newton. Espera como variables los vectores x e y de nodos y ordenadas, respectivamente.
- **dif_divHermite.m**, devuelve la tabla de diferencias divididas y los coeficientes del polinomio interpolador de Hermite en la forma de Newton. Espera como variables el vector x de los nodos y una celda y que contiene los datos para la ordenada y las derivadas en cada nodo. Para comprender cómo funcionan las celdas consulta los comentarios sobre Octave al final del ejercicio 1.

  Las dos funciones anteriores no son realmente necesarias: las he incluido para que tengáis una herramienta que muestre la tabla de diferencias divididas, que puede ser práctico para chequear algunos cálculos.
- **interpolNewton.m**, devuelve los coeficientes del polinomio interpolador en la forma de Newton. Espera las mismas variables que *dif_divNewton*.
- **interpolHermite.m**, devuelve los coeficientes del polinomio interpolador de Hermite en la forma de Newton y una lista de los nodos repetidos, que será necesaria para evaluar el

polinomio. Espera las mismas variables que *dif_divHermite*.

- **polyinterpolador_eval.m**, espera como variables el vector de coeficientes del polinomio en la forma de Newton, el vector de nodos (repetidos en el caso de que el polinomio a evaluar sea de Hermite) y el vector de abscisas donde se desea evaluar el polinomio. Devuelve el vector de evaluaciones.
- **polyinterpolador.m**, devuelve los coeficientes de un polinomio interpolador escrito en la forma estándar $a_n x^n + \cdots + a_1 x + a_0$. Espera como variables los vectores de coeficientes de la forma de Newton y el vector de nodos (repetidos si se trata de un polinomio de Hermite).
- **nodosCheby.m**, espera como variables los extremos del intervalo [a, b] (en el orden natural) y un natural n. Devuelve los n + 1 nodos de Chebyshev en [a, b], es decir los ceros del polinomio de grado n + 1.

Antes de utilizar cualquiera de estas funciones, asegurate de entender qué argumentos espera, su tipo y significado, así como qué salidas produce, su significado, orden y formato.

1.1 Ejercicio 1: Dos ejemplos sencillos

1.1.1 1.1

Crea el sript **Ejercicio1.m** para construir los polinomios interpoladores correspondientes a los datos

x	1.5	2.7	3.1	-2.1	-6.6	11.0
y	0.0	1.0	-0.5	1.0	0.5	0.0

y los datos

x	y	y'	y''
0	1	2	
1	0	1	1
2	3		
3	1	1	

1.1.2 1.2

En cada caso, representa en una gráfica el polinomio interpolador y los puntos de la tabla interpolada.

Solución 1.1 y 1.2

- Primera tabla:

[2]:
```
x = [1.5, 2.7, 3.1, -2.1, -6.6, 11.0];
y = [0.0, 1.0, -0.5, 1.0, 0.5, 0.0];

[tabla,coefNewton] = dif_divNewton(x, y);
```

```
coefNewtonOrdenados = polyinterpolador(coefNewton, x)
```

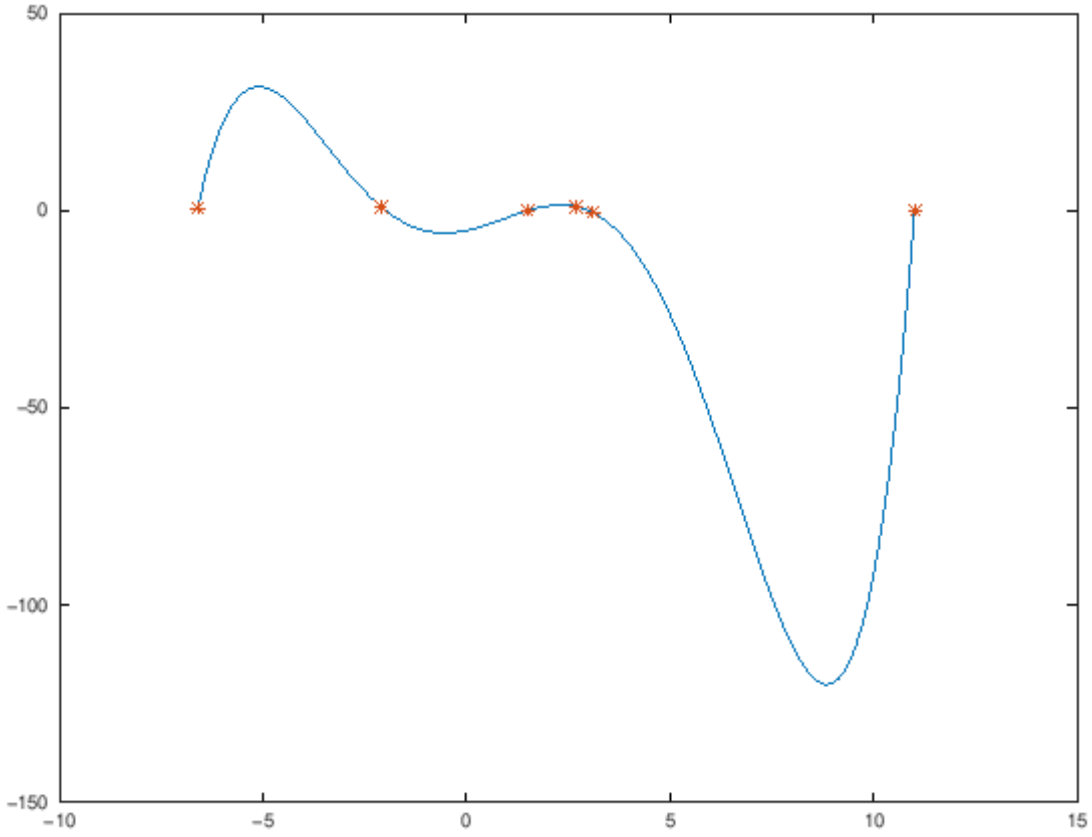
```
Número de nodos de interpolación= 6
Nodos:
    1.5000    2.7000    3.1000   -2.1000   -6.6000   11.0000
Ordenadas:
        0   1.0000  -0.5000   1.0000   0.5000        0

La tabla de diferencias divididas es:
    1.5000         0         0         0         0         0         0
    2.7000    1.0000    0.8333         0         0         0         0
    3.1000   -0.5000   -3.7500   -2.8646         0         0         0
   -2.1000    1.0000   -0.2885   -0.7212   -0.5954         0         0
   -6.6000    0.5000    0.1111   -0.0412   -0.0731   -0.0645         0
   11.0000         0   -0.0284   -0.0107    0.0039    0.0093    0.0078

Los coeficientes del polinomio interpolador de Newton son:
        0   0.8333  -2.8646  -0.5954  -0.0645   0.0078
coefNewtonOrdenados =

   7.7636e-03  -5.3610e-02  -5.1304e-01   1.6396e+00   2.1881e+00  -5.0273e+00
```

El polinomio escrito en terminos de x será:

$$0.0077636 * x^5 - 0.05361 * x^4 - 0.51304 * x^3 + 1.6396 * x^2 + 2.1881 * x^1 - 5.0273$$

Representamos en una gráfica:

[3]:
```
interval = linspace(min(x), max(x),200);
plot(interval, polyinterpolador_eval(coefNewton, x, interval), x, y, '*')
```



- Tabla 2:

[4]:
```
x = [0, 1, 2, 3];
y = {[1,2], [0,1,1], [3], [1,1]};

[coefNewton, x_rep] = interpolHermite(x, y)

coefNewtonOrdenados = polyinterpolador(coefNewton, x_rep)
```

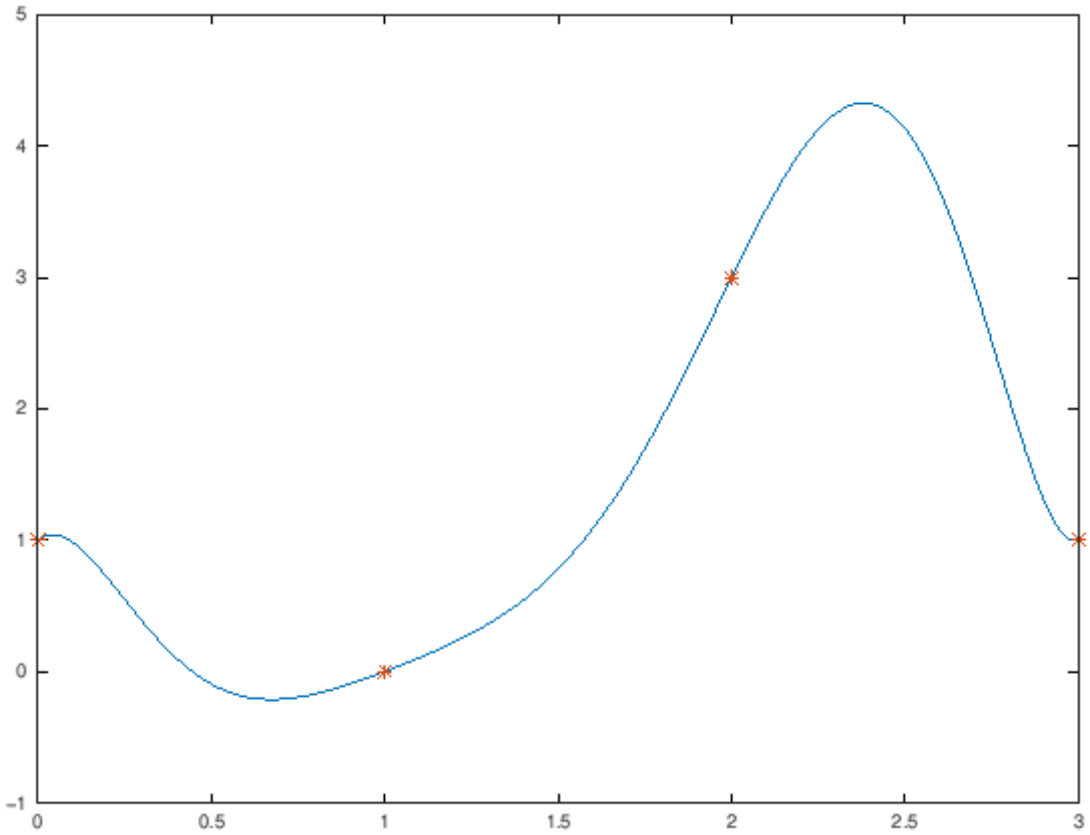
```
coefNewton =

   1.0000    2.0000   -3.0000    5.0000   -6.5000    4.0000   -1.7083    0.9375

x_rep =

   0   0   1   1   1   2   3   3

coefNewtonOrdenados =

    0.9375   -9.2083   35.0417  -65.7500   63.5208  -27.5417    2.0000    1.0000
```

El polinomio escrito en terminos de x será:

$$0.9375 * x^7 - 9.2083 * x^6 + 35.042 * x^5 - 65.75 * x^4 + 63.521 * x^3 - 27.542 * x^2 + 2 * x^1 + 1$$

Representamos en una gráfica:

[5]:
```
interval = linspace(0, 3, 200);

for i= 1:length(y)
    ord(i) = y{i}(1);
endfor

plot(interval, polyinterpolador_eval(coefNewton, x_rep, interval), x, ord, "*");
```



1.2 Comentarios sobre Octave

- Para la interpolación de Hermite debemos utilizar variables de tipo celda (**cell**). Las celdas se pueden definir directamente como los vectores o matrices, pero utilizando llaves ({ }) en lugar de corchetes ([ ]). Así, por ejemplo, si escribimos

y={[1,2],[0,1,1],[3],[1,1]}

estaríamos definiendo una celda con una fila y cuatro «columnas» (denominadas y{1,1}, y{1,2}, etc.), cada una de ellas contiene los vectores indicados entre las llaves (que son vectores fila en realidad, por ejemplo y{1,2} es de tamaño 1×3). Como ves podemos de esta forma definir objetos estructurados de tipo matriz, pero donde la longitud de las columnas (o filas) es distinto para cada fila (o columna); incluso podemos definir una componente de una celda como una matriz o una celda.

En Octave no es posible indexar sobre las componentes de una celda. Por ejemplo, con la definición anterior de la celda y, no podemos hacer y{1:4}(1), que sería lo natural para recuperar la primera componente de cada una de las cuatro filas de y. Sí podemos indexar como y{2:4}, que devuelve las componentes 2 a 4 (que sería más lógico y válido denominar y{:,2:4}). También podemos recuperar una fila, por ejemplo, con z=y{2} obtenemos un vector z cuyo valor es [0,1,1]. Para recuperar por ejemplo la primera componente de cada componente haremos un bucle:

[6]:
```
%{
for i=1:length(y)
    z(i) = y{i}(1)
endfor
%}
```

- A menudo las ventanas gráficas quedan escondidas tras otras ventanas (si no sueles trabajar a la vez con varias ventanas, esto no es un problema) o tienen un tamaño o proporciones inadecuadas. Un truco para hacer aparecer las figuras previstas en un script puede ser incluir todas las órdenes que generan las figuras al final del script, pero esto obviamente puede ser molesto.

Una forma de hacerlas visibles es decidir dónde se van a colocar en pantalla y qué tamaño tendrán. Para esto disponemos de la orden siguiente:

set(1,"position",[x0,y0,x1,y1])

Con esta orden especificamos valores para la ventana gráfica correspondiente a la figure 1; los valores x0, y0 son las coordenadas de pantalla de la ventana de la figura y x1, y1 son las unidades de medida del ancho y alto. Para orientarte sobre los valores de estas dimensiones puedes obtener los valores de cualquier ventana gráfica que esté abierta (suponiendo que es la figura 1) con

get(1,"position")

1.3 Ejercicio 2: Fenómeno de Runge

Considera la función de Runge, $f(x) = \frac{1}{1+x^2}$, sobre e intervalo [-5, 5]. Vamos a representar gráficamente distintos polinomios interpoladores para esta función, así como los errores en la aproximación de los polinomios a la propia función. Los tres primeros apartados os realizamos en un script llamado **Ejercicio2_1.m**, el apartado d) en **Ejercicio2_2.m** y el apartado f) en **Ejercicio2_3.m**.

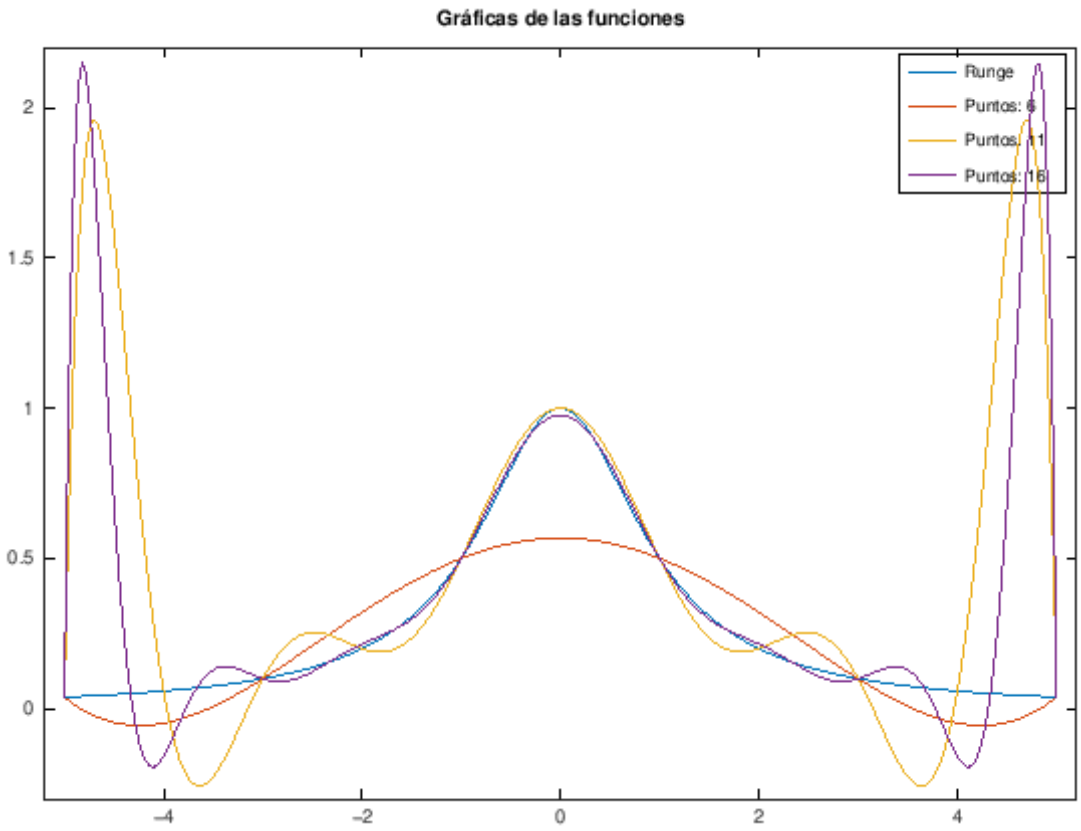
1.3.1 2.1 (apartado 'a')

Escribe un programa que, dado un vector de enteros positivos N , dibuje en una ventana la gráfica de la función de Runge y la gráfica de los polinomios interpoladores p_{N_i} de f en los $N_i + 1$ puntos equiespaciados que dividen al intervalo $[-5, 5]$ en N_i partes iguales, para cada componente N_i de N . Ejecútalo con el vector N=[5,10,15]. Puedes intentar hacerlo con valores mayores de N , por

ejemplo N=[20,30,40]. Juega un poco con la orden *ylim*, para poder apreciar bien la gráfica de la función de Runge y los polinomios a su alrededor.

Solución 2.1

[7]:
```
clear all

% Ejercicio2_1.m%

function ret = Runge(x)
    ret = 1./(1+x.^2);
end

N = [5, 10, 15];
interval = linspace(-5, 5, 501);

plot(interval, Runge(interval), sprintf(';Runge ;'))

for i=1:length(N)
    hold on
    nodos = linspace(-5, 5, N(i)+1);
    ordenadas = Runge(nodos);
    coef = interpolNewton(nodos, ordenadas);
    poly_eval = polyinterpolador_eval(coef, nodos, interval);

    plot(interval, poly_eval, sprintf(';Puntos: %u;', N(i) +1)) %sprintf para␣
 ↪la leyenda, hacer que cambie con el valor de cada i

endfor

title('Gráficas de las funciones')

ylim([-0.3, 2.2])
xlim([-5.2,5.2])
```



1.3.2 2.2 (apartado b)

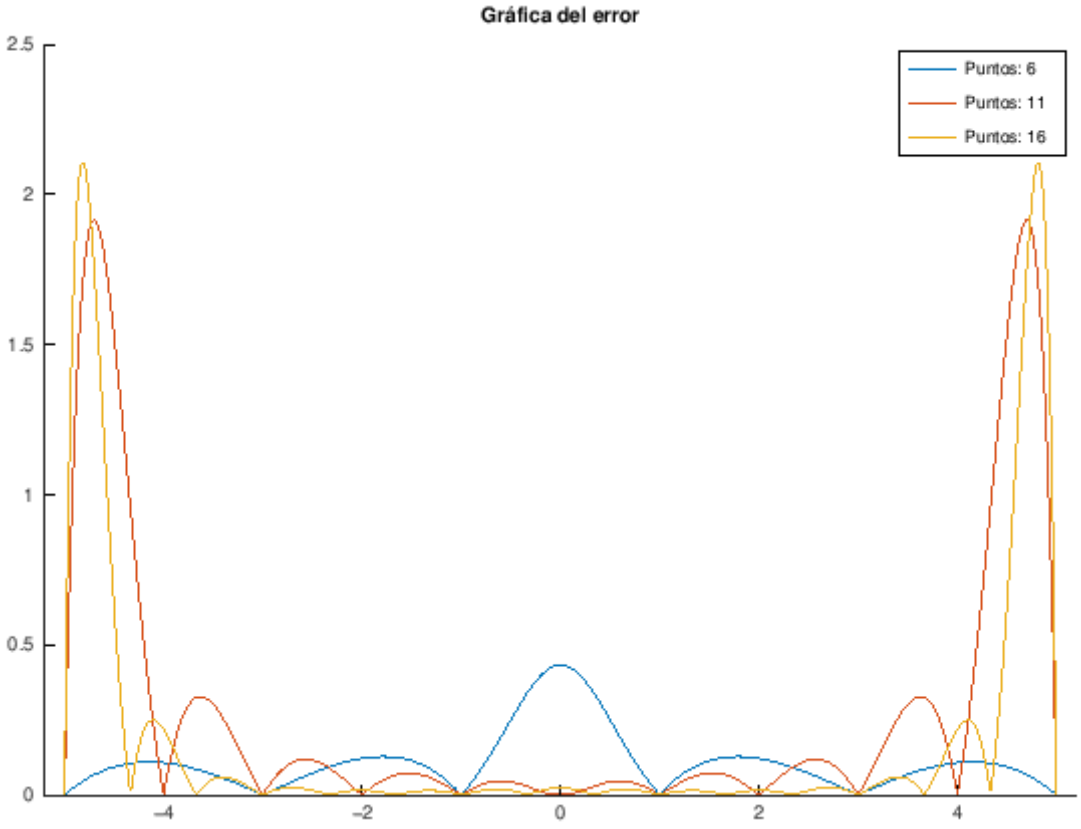
En el mismo programa anterior, introduce el código necesario para que dibuje en una segunda ventana las gráficas de las distintas funciones de error $|f(x) - p_N(x)|$. Utilizando la orden de Octave *max(x)*, que devuelve la mayor de las componentes del vector x, aproxima el error máximo, aplicando max a una muestra de 1000 puntos en el intervalo.

Solución 2.2

[8]:
```
figure(2)

interval = linspace(-5, 5, 1000);

for i=1:length(N)
    hold on
    nodos = linspace(-5, 5, N(i)+1);
    ordenadas = Runge(nodos);
    coef = interpolNewton(nodos, ordenadas);
    poly_eval = polyinterpolador_eval(coef, nodos, interval);

    plot (interval, abs(poly_eval -Runge(interval)), sprintf(';Puntos: %u;',␣
 ↪N(i) +1)) %sprintf para la leyenda, hacer que cambie con el valor de cada i
```

```
    error(i) = max(abs(poly_eval -Runge(interval)));
    printf('El error máximo en %i nodos es %f. \n', N(i), error(i))

endfor

title('Gráfica del error')

xlim([-5.2,5.2])
```

```
El error máximo en 5 nodos es 0.432669.
El error máximo en 10 nodos es 1.915633.
El error máximo en 15 nodos es 2.106860.
```



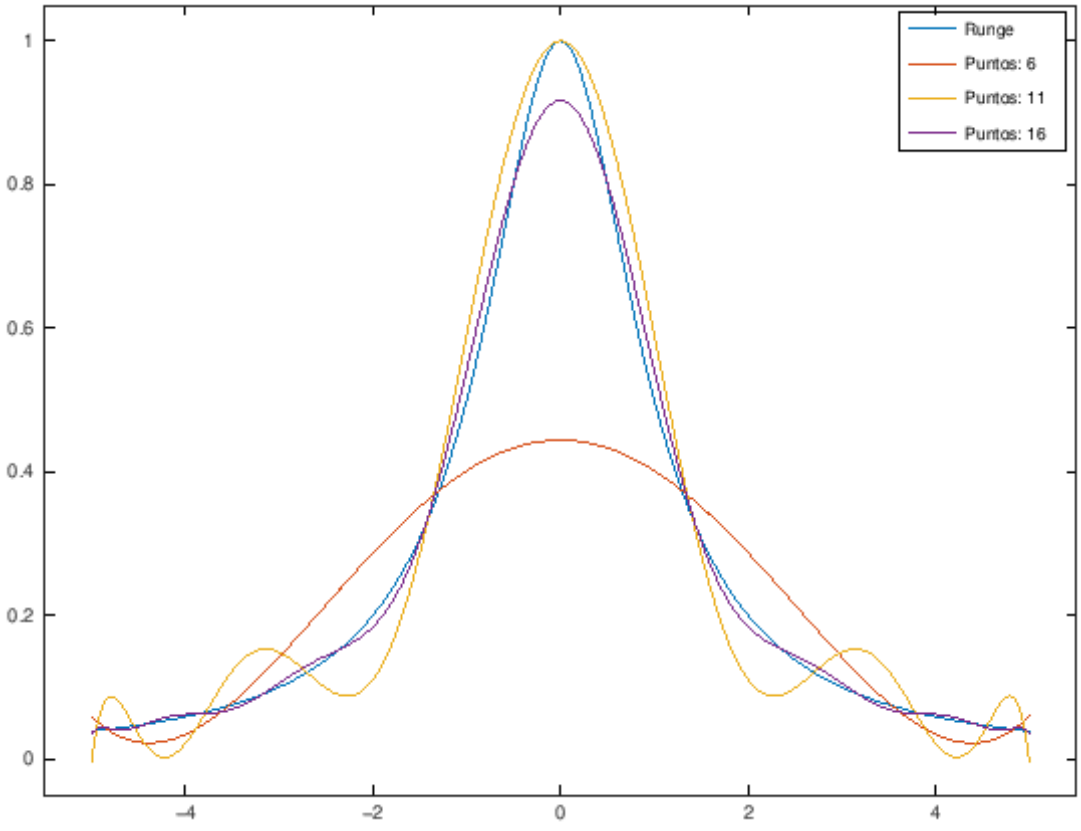
1.3.3 2.3 (apartado c)

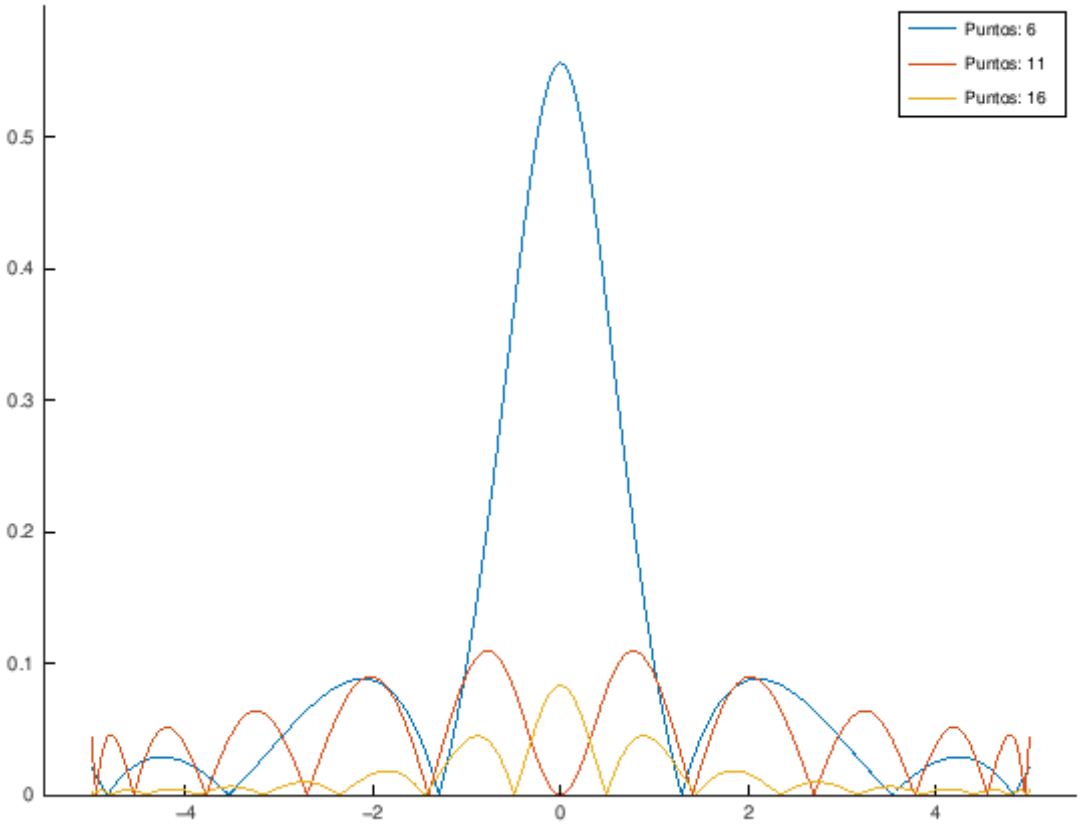
Realliza las mismas tareas anteriores pero considerando interpolación con nodos de Chebysev, para los mismos números de puntos.

Solución 2.3

[9]:
```
figure(3)

a = -5;
b = 5;
interval = linspace(a, b, 1000);

plot(interval, Runge(interval), sprintf(';Runge ;'))

for i=1:length(N)

    nodosCheb = nodosCheby(a, b, N(i));
    ordenadasCheb = Runge(nodosCheb);
    coef = interpolNewton(nodosCheb, ordenadasCheb);
    poly_eval = polyinterpolador_eval(coef, nodosCheb, interval);

    hold on
    plot(interval, poly_eval, sprintf(';Puntos: %u;', N(i) +1)) %sprintf para␣
 ↪la leyenda, hacer que cambie con el valor de cada i

endfor
ylim([-0.05, 1.05])
xlim([-5.5, 5.5])

figure(4)
for i = 1:length(N)
    nodosCheb = nodosCheby(a, b, N(i));
    ordenadasCheb = Runge(nodosCheb);
    coef = interpolNewton(nodosCheb, ordenadasCheb);
    poly_eval = polyinterpolador_eval(coef, nodosCheb, interval);

    error(i) = max(abs(poly_eval -Runge(interval)));
    printf('El error máximo en %i nodos es %f. \n', N(i), error(i))

    hold on
    plot (interval, abs(poly_eval -Runge(interval)), sprintf(';Puntos: %u;',␣
 ↪N(i) +1)) %sprintf para la leyenda, hacer que cambie con el valor de cada i

endfor
ylim([0.0, 0.6])
xlim([-5.5, 5.5])
```

```
El error máximo en 5 nodos es 0.555887.
El error máximo en 10 nodos es 0.109154.
El error máximo en 15 nodos es 0.083094.
```


Runge
Puntos: 6
Puntos: 11
Puntos: 16
1
0.8
0.6
0.4
0.2
0
−4
−2
0
2
4



1.3.4 2.4 (apartado d)

Ahora considera la interpolación de Hermite para valores N = [5, 10, 15], añadiendo las condiciones siguientes:

1. Valor de las derivadas primera y segunda en -5 y 5.
2. Valor de la derivada primera en -5, $-1/\sqrt{3}$, $1/\sqrt{3}$ y 5.

Añade la segunda ventana con los respectivos errores. (Puedes hacer también N = [20, 30, 40].)

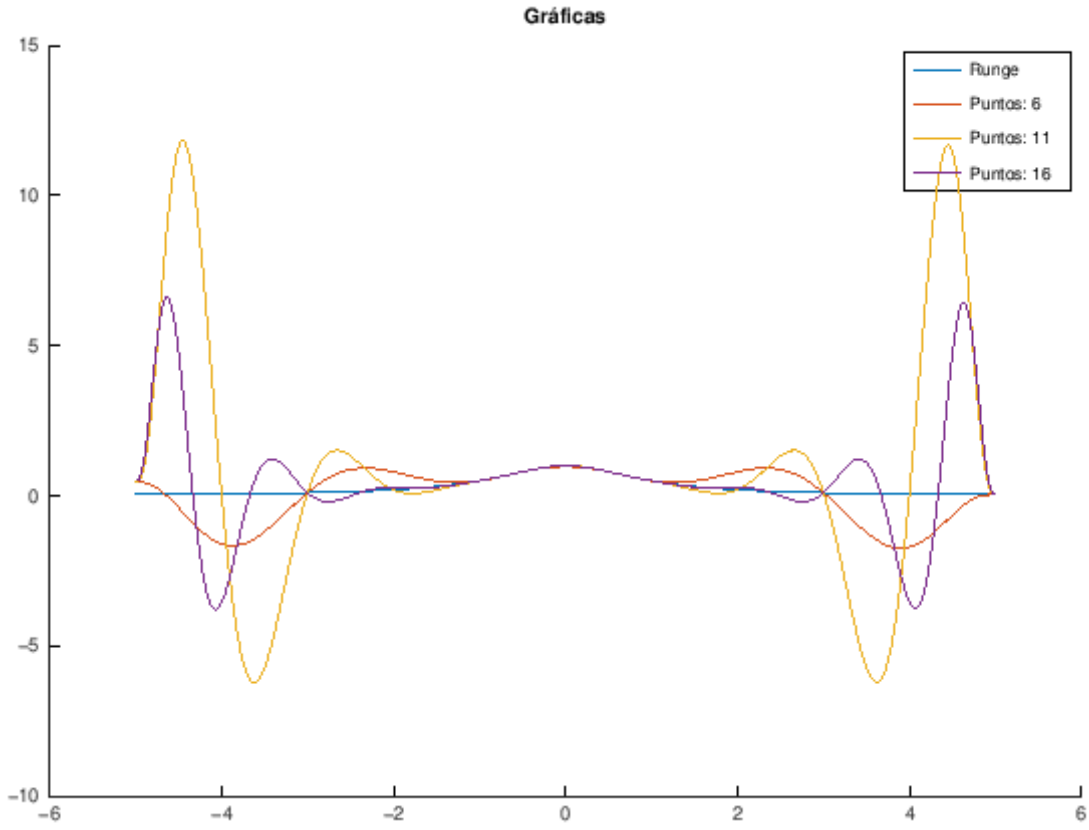
Solución 2.4

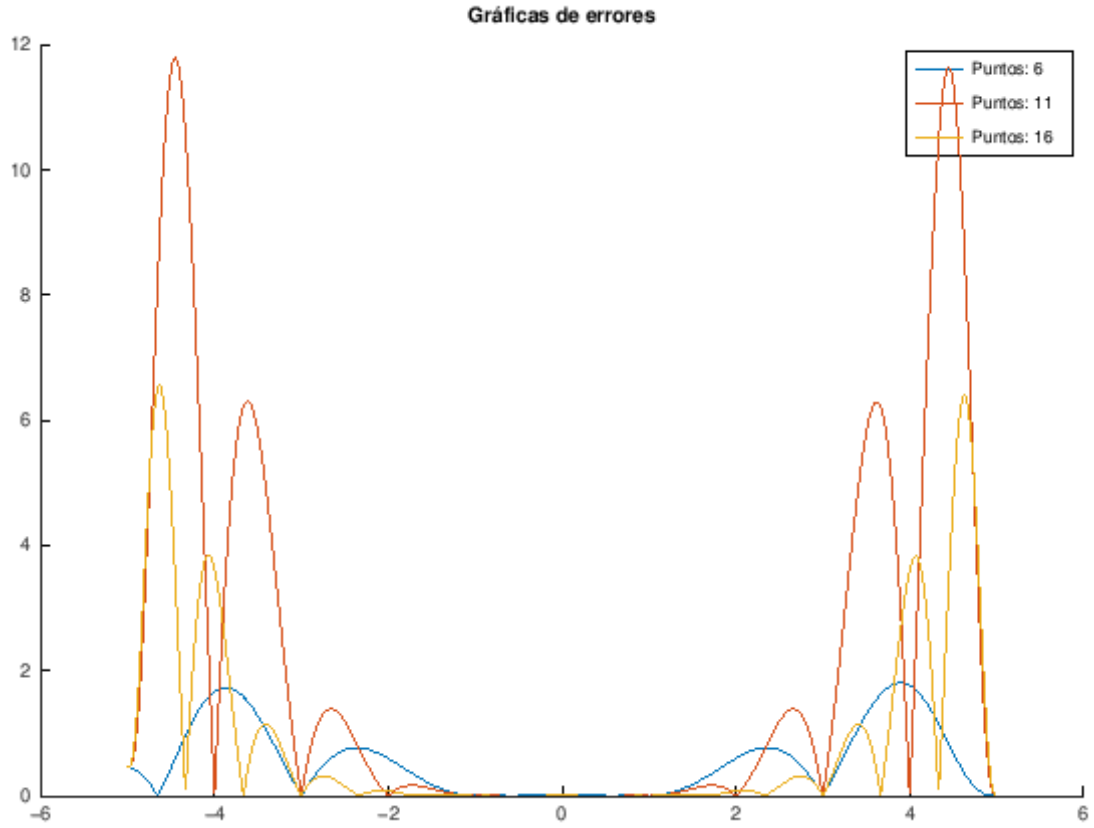
[10]:
```
clear all

% Ejercicio2_2.m%

function ret = Runge(x)
    ret = 1./(1+x.^2);
end

% También se pueden calcular a mano las derivadas en lugar de definirlas en␣
 ↪octave.
function ret = diff_1_Runge(x)
```

```
    ret = (-2.*x)./((1+x.^2).^2);
end

function ret = diff_2_Runge(x)
    ret = (-2.*((-3.*x.^2) + 1))./((1+x.^2).^3);
end

a = -5;
b = 5;
interval = linspace(a, b, 1000);
N = [5, 10, 15];

figure(1)
title('Gráficas')
hold on
plot(interval, Runge(interval), ';Runge;')

for i=1:length(N)
    hold on
    clear y

    nodos = linspace(a, b, N(i)+1);
    k = length(nodos);

    % Añadimos la primera condición.
    y{1} = [Runge(1), diff_1_Runge(1), diff_2_Runge(1)];
    y{k} =  [Runge(nodos(k)), diff_1_Runge(nodos(k)), diff_2_Runge(nodos(k))];

    for j=2:k-1
        y{j}(1) = Runge(nodos(j));
    endfor

    % Añadimos la segunda condición
    nodos(k+1) = (-1/sqrt(3));
    y{k+1} = [Runge(nodos(k+1)), diff_1_Runge(nodos(k+1))];

    nodos(k+2) = (1/sqrt(3));
    y{k+2} = [Runge(nodos(k+2)), diff_1_Runge(nodos(k+2))];

    % Dibujamos la gráfica.
    [coefNewton, x_rep] = interpolHermite(nodos, y);
    plot(interval, polyinterpolador_eval(coefNewton, x_rep, interval),␣
↪sprintf(';Puntos: %u;', N(i) +1)) %sprintf para la leyenda, hacer que cambie␣
↪con el valor de cada i);

    % Calculamos los errores.
    poly_eval = polyinterpolador_eval(coefNewton, x_rep, interval);
```

```
    error(i) = max(abs(poly_eval -Runge(interval)));
    errores{i} = abs(poly_eval - Runge(interval)); % Guardamos los errores para␣
 ↪no tener que recalcularlos a la hora de dibujar la gráfica de errores.

    printf('El error máximo en %i nodos es %f. \n', N(i)+1, error(i))

endfor

figure(2)
title('Gráficas de errores')
hold on

for i=1:length(N)
    hold on
    plot (interval, errores{i}, sprintf(';Puntos: %u;', N(i) +1)) %sprintf para␣
 ↪la leyenda, hacer que cambie con el valor de cada i
endfor
```

```
El error máximo en 6 nodos es 1.805543.
El error máximo en 11 nodos es 11.785275.
El error máximo en 16 nodos es 6.553418.
```





1.3.5 2.5 (apartado e)

Analiza la aproximación que proporcionan los distintos polinomios interpoladores con los que hemos experimentado.

Solución 2.5 Una conclusión rápida a la que podemos llegar es a que, de acuerdo a la teoría, al interpolar aproximando con los nodos de Chebyshev las aproximaciones a la función original son mucho más precisas.

1.3.6 2.6 (apartado f)

El propio Runge demostró que, para nodos equidistribuidos, los polinomios $p_n(x)$ que interpolan a la función $f(x)$ anterior convergen uniformemente a $f(x)$ en el intervalo [−c, c], con c 3.63 y divergen fuera de [−c, c]. Considera ahora funciones $f_a(x) = \frac{1}{1+ax^2}$, para $a > 0$.

Estudia gráficamente si el intervalo donde los polinomios convergen a $f_a(x)$ crece o decrece con el valor de a.

Solución 2.6

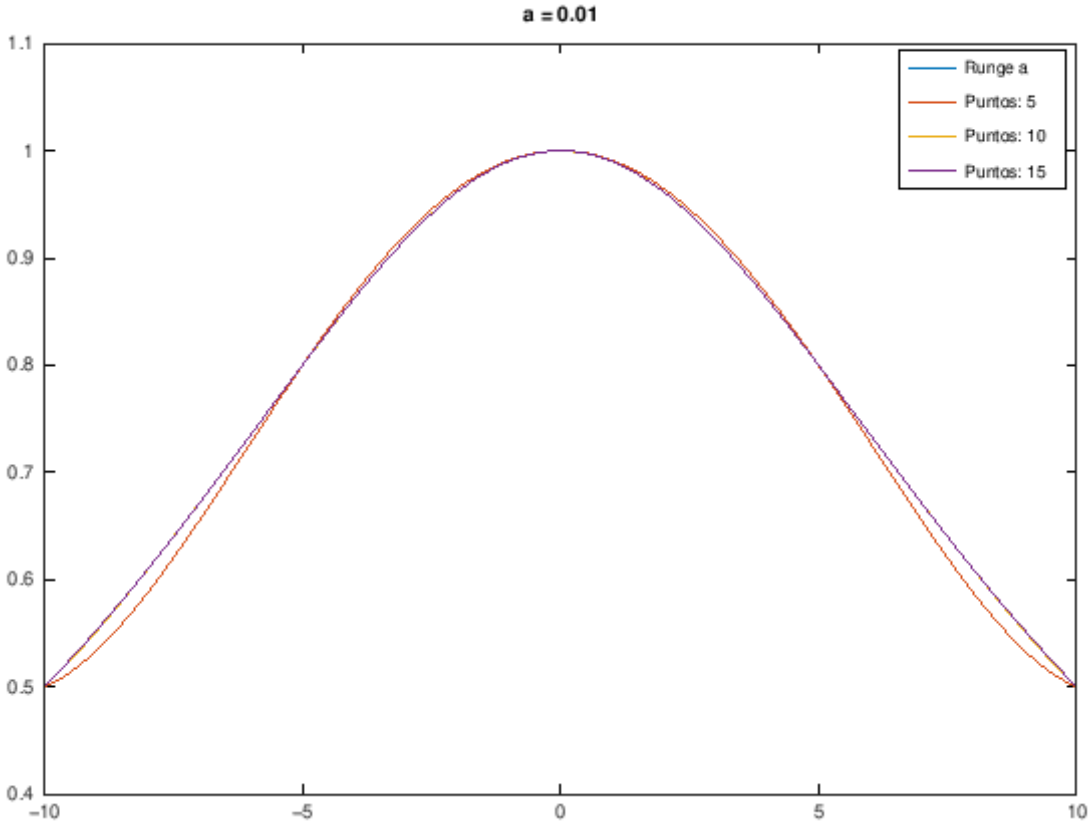
[11]:
```
clear all

% Ejercicio2_3.m%

function ret = runge_a(a, x)    % a será una constante. La función de Runge␣
 ↪original será equivalente a esta cuando a = 1.
    if a<0
        error("'a' debe ser mayor que cero.")
    endif

    ret = 1./(1+a.*x.^2);
end
```

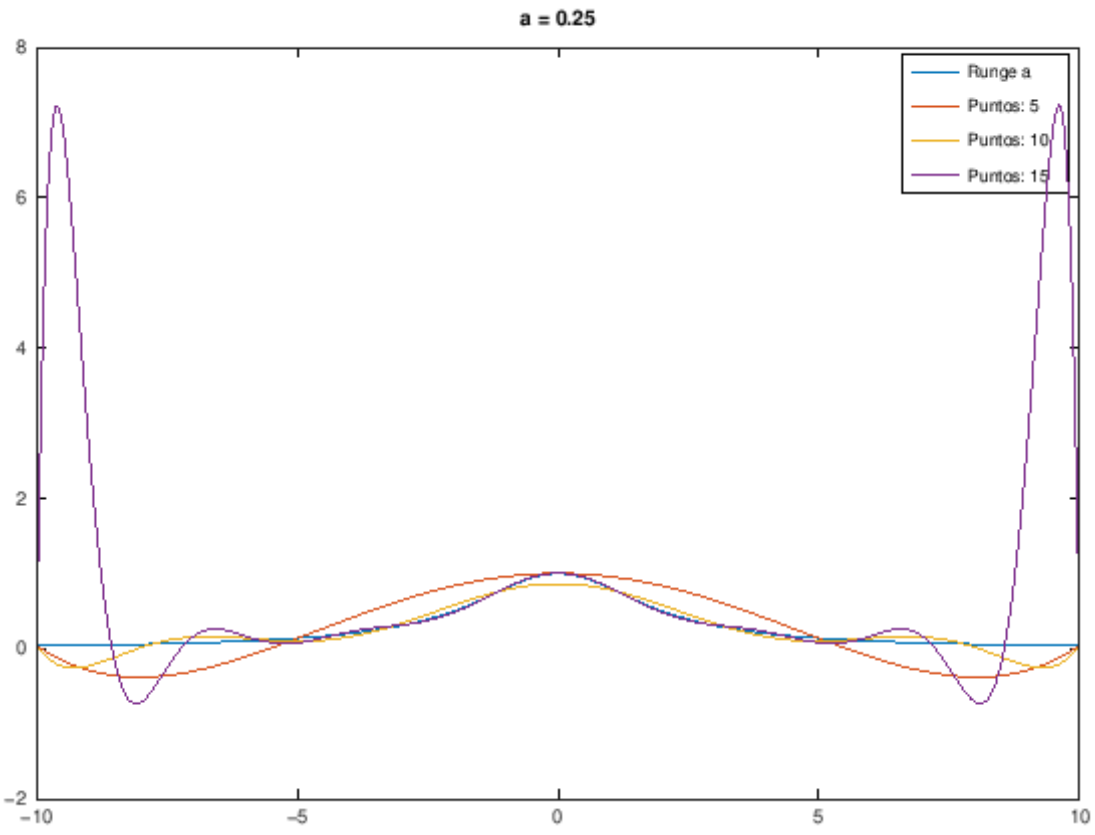
Vamos entonces a ver si con la misma cantidad de nodos, de manera visual, podemos apreciar si el intervalo donde los polinomios interpoladores de n nodos equidistribuidos convergen uniformemente a nuestra función *runge_a()* crece o decrece en función de a.

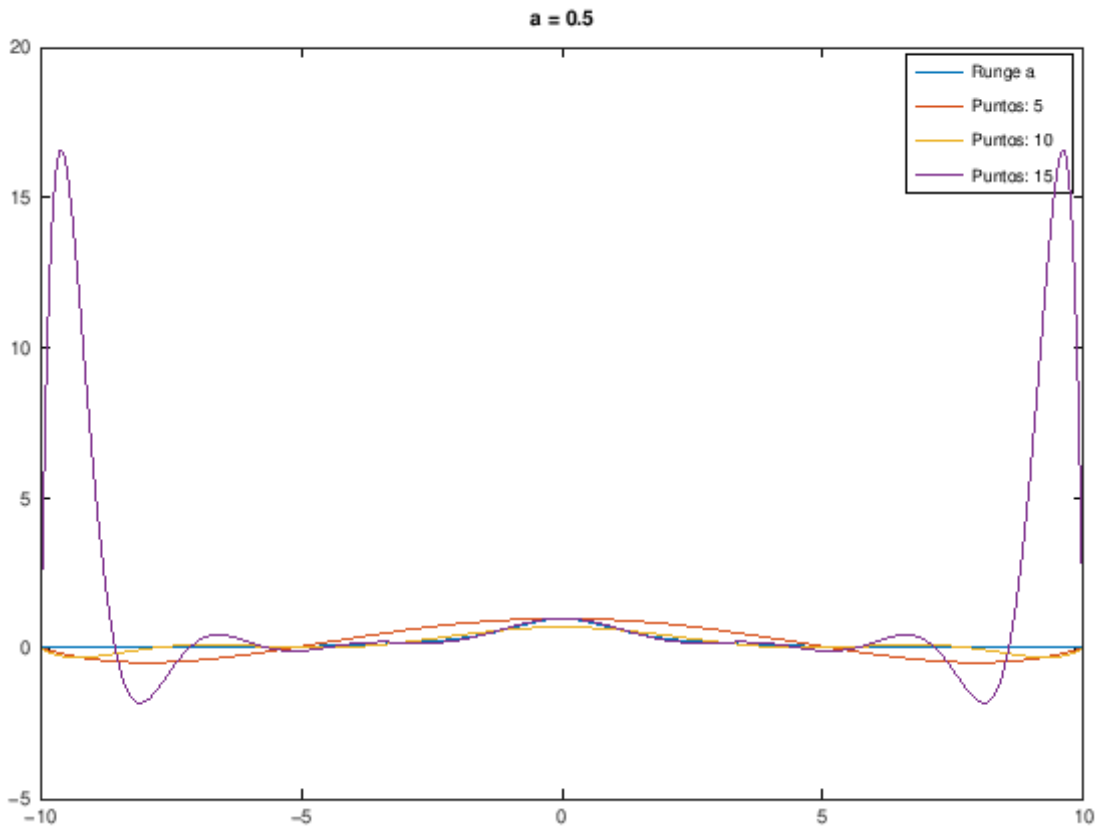
Para ello vamos a tomar 5 valores diferentes de a.

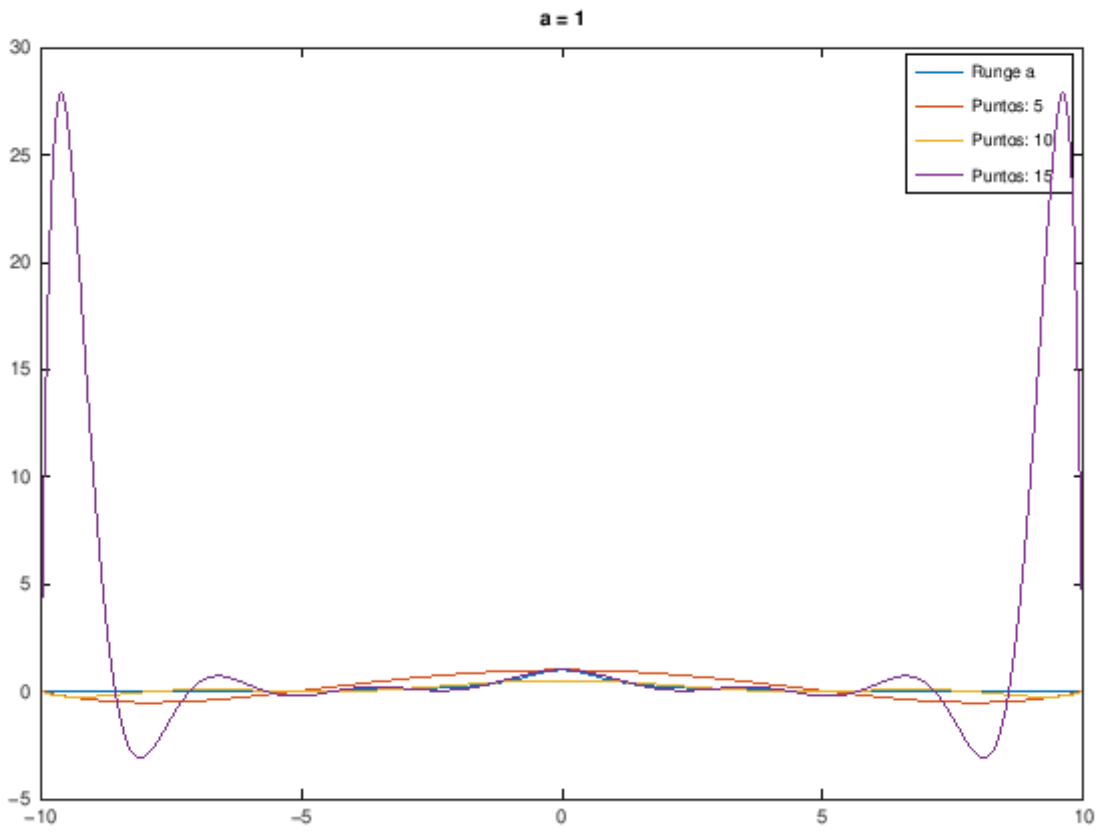
[12]:
```
a = [0.01, 0.25, 0.5, 1, 2, 8, 200];

N = [5, 10, 15];
left = -10;
right = 10;

interval = linspace(left, right, 500);

for i=1:length(a)

    figure(i)
    plot(interval, runge_a(a(i), interval), ';Runge a;')
    title(sprintf('a = %d', a(i)))
    hold on

    for j=1:length(N)

        nodos = linspace(left, right, N(j));
        ordenadas = runge_a(a(i), nodos);

        coef = interpolNewton(nodos, ordenadas);
        poly_eval = polyinterpolador_eval(coef, nodos, interval);

        hold on
        plot(interval, poly_eval, sprintf(';Puntos: %u;', N(j)))


    endfor
```

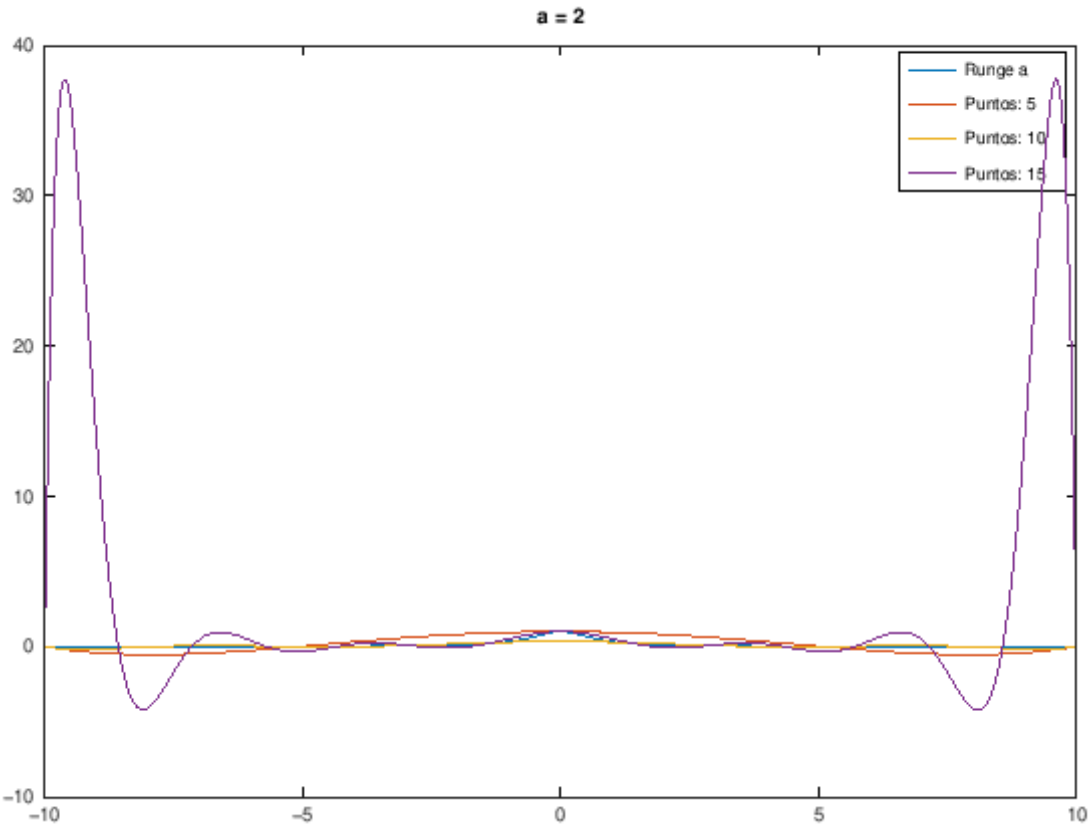
```
endfor
```

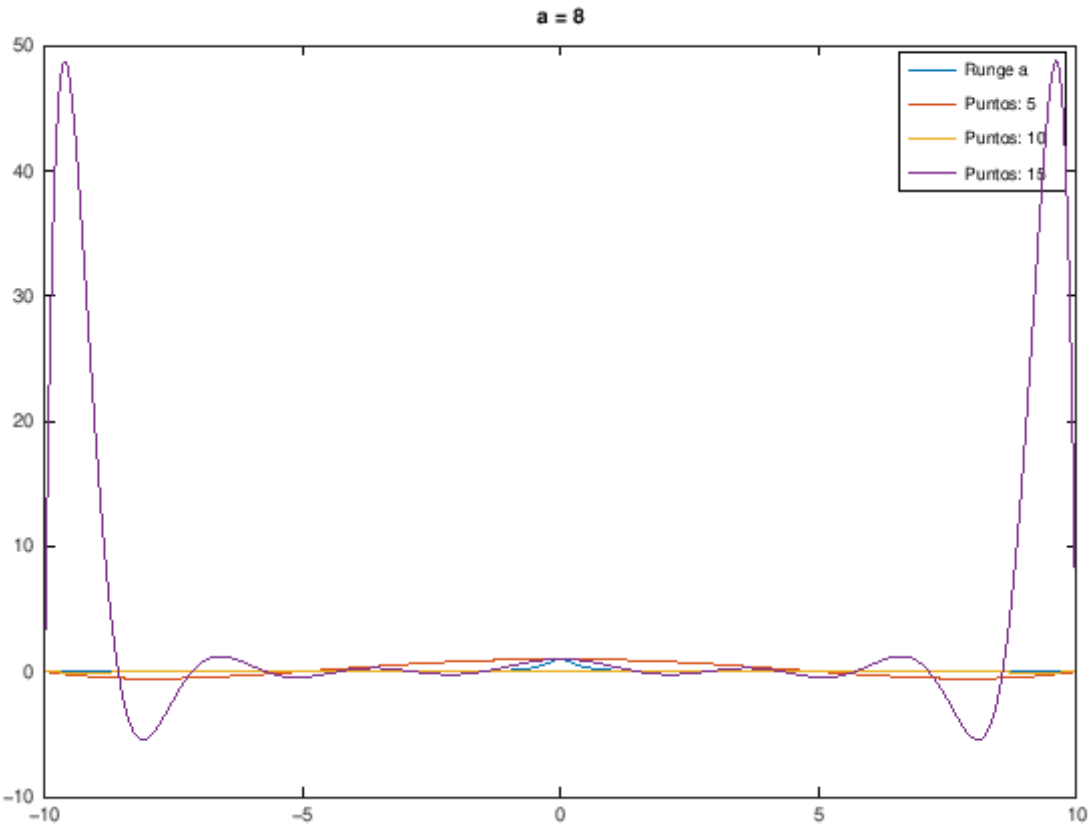

a = 0.01
Runge a
Puntos: 5
Puntos: 10
Puntos: 15
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
−10
−5
0
5
10

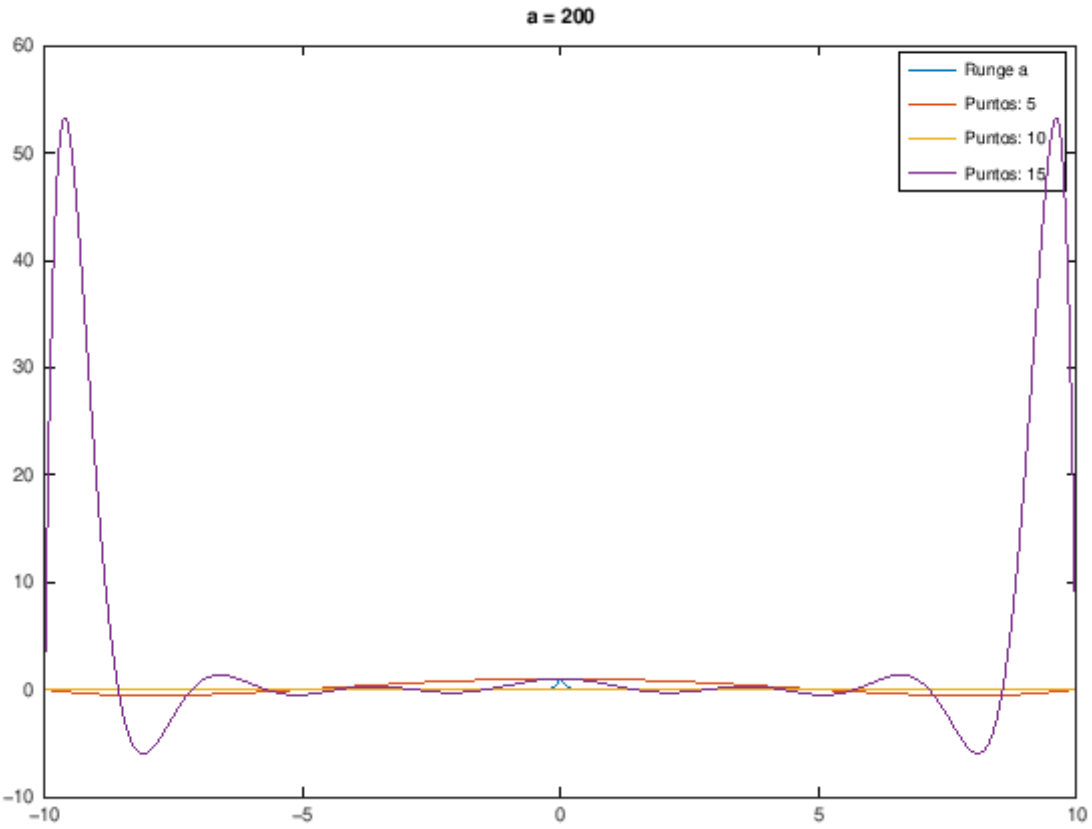

a = 0.25
Runge a
Puntos: 5
Puntos: 10
Puntos: 15
8
6
4
2
0
−2
−10
−5
0
5
10


a = 0.5
Runge a
Puntos: 5
Puntos: 10
Puntos: 15
20
15
10
5
0
−5
−10
−5
0
5
10


a = 1
Runge a
Puntos: 5
Puntos: 10
Puntos: 15
30
25
20
15
10
5
0
−5
−10
−5
0
5
10


a = 2
Runge a
Puntos: 5
Puntos: 10
Puntos: 15
40
30
20
10
0
−10
−10
−5
0
5
10


a = 8
Runge a
Puntos: 5
Puntos: 10
Puntos: 15
50
40
30
20
10
0
−10
−10
−5
0
5
10



Después de esta serie de pruebas parece que el intervalo donde las funciones convergen se mantiene igual independientemente de a.

1.4 Ejercicio 3: Interpolación inversa. Solución de una ecuación.

El propósito del ejercicio es aproximar la solución de la ecuación sen(x) = 0.75, es decir, el valor de x = arc sen(0.75). Esto es un avance de un método de aproximación de soluciones que veremos con detalle en el tema 4. Si queremos calcular una solución de la ecuación $f(x_0) = 0$, lo que buscamos es la función inversa f^{-1} pues, de esa forma, $x_0 = f^{-1}(0)$. Puesto que no conocemos $f^{-1}(y)$ lo que hacemos es aproximarla por un polinomio interpolador.

1.4.1 3.1 (apartado a)

Utiliza la lista de puntos de la gráfica de la función seno,

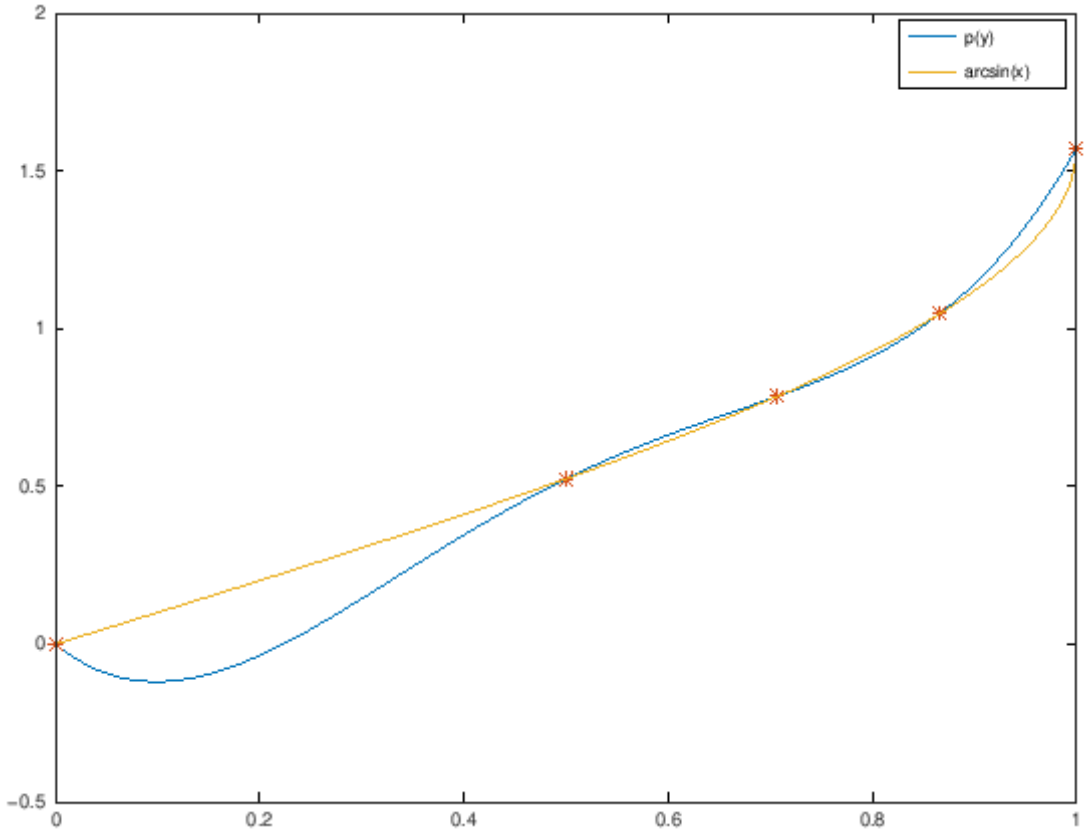
x	0	$\frac{\pi}{6}$	$\frac{\pi}{4}$	$\frac{\pi}{3}$	$\frac{\pi}{2}$
sen(x)	0	0.5	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{3}}{2}$	1

para construir el polinomio interpolador *p(y)* de la lista de puntos *(y, x)* que se obtiene al intercambiar las coordenadas de los puntos de la lisya *(x, y)*.

Solución 3.1

[13]:
```
clear all
% Ejercicio_3.m%

y = [0, pi/6, pi/4, pi/3, pi/2];
x = sin(y);

interval = linspace(0, 1, 500);

coef = interpolNewton(x, y);
coef_ordenados = polyinterpolador(coef, x);
polyout(coef_ordenados, 'x')

plot(interval, polyinterpolador_eval(coef, x, interval), ';p(y) ;', x, y, '*')
hold on
plot(interval, asin(interval), ';arcsin(x) ;')
```

```
12.488*x^4 - 25.035*x^3 + 16.745*x^2 - 2.6276*x^1 + 0
```



1.4.2 3.2 (apartado b)

Considera la aproximación $x = \arcsin(0.75) \approx p(0.75)$.

Solución 3.2 Para una solución visual, mirar gráfica anterior.

[14]:
```
aproximacion = polyinterpolador_eval(coef, x, 0.75)
valor_real = asin(0.75)
error_abs = abs(aproximacion - valor_real)
```

```
aproximacion = 0.8381
valor_real = 0.8481
error_abs = 9.9312e-03
```

1.4.3 3.3 (apartado c)

Evalúa la aproximación conseguida (puedes calcular el valor de arc sen(0.75) que proporciona Octave, función asin(x), donde x se expresa en radianes; también existe asind que devuelve el ángulo en grados sexagesimales).

Solución 3.3 Es lo que está hecho en el apartado anterior.

Nota de Alonso: No he entendido muy bien la estructura del ejercicio XD. Los he ido haciendo en orden según he creído conveniente.

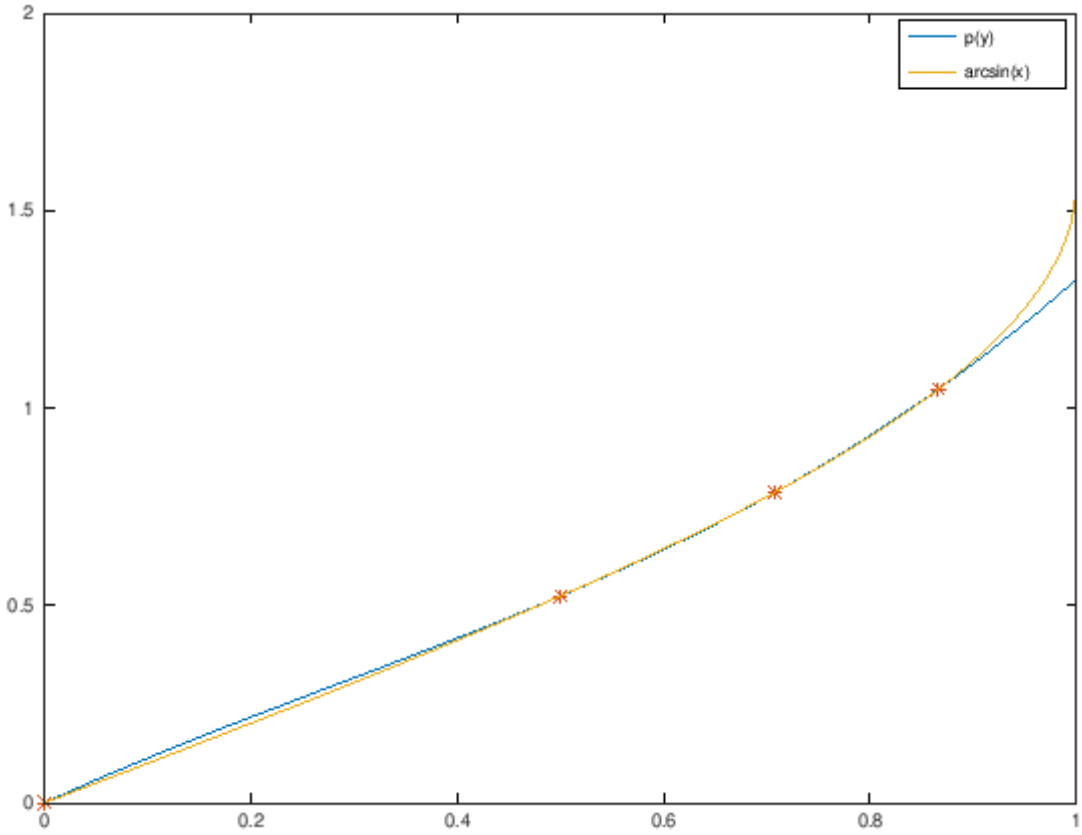
1.4.4 3.4 (apartado d)

Elimina de la lista de datos el último punto, ¿mejora la aproximación?

Solución 3.4

[15]:
```
y_mod = [0, pi/6, pi/4, pi/3]; % mod viene de modificado
x_mod = sin(y_mod);

coef_mod = interpolNewton(x_mod, y_mod);

plot(interval, polyinterpolador_eval(coef_mod, x_mod, interval), ';p(y) ;',␣
 ↪x_mod, y_mod, '*')
hold on
plot(interval, asin(interval), ';arcsin(x) ;')

aproximacion_mod = polyinterpolador_eval(coef_mod, x_mod, 0.75)
valor_real = asin(0.75)
error_abs_mod = abs(aproximacion_mod - valor_real)

if error_abs_mod < error_abs
    disp('La segunda aproximación es mejor.')
else
    disp('La primera aproximación es mejor.')
endif
```

```
aproximacion_mod = 0.8498
valor_real = 0.8481
error_abs_mod = 1.7221e-03
La segunda aproximación es mejor.
```



Sorprendentemente, quitando el último punto, en x = 0.75 podemos observar que la segúnda aproximación es mejor.

Sin embargo, no es tan sorprendente si nos damos cuenta que *arcsin(x)* presenta una singularidad en x = 1.