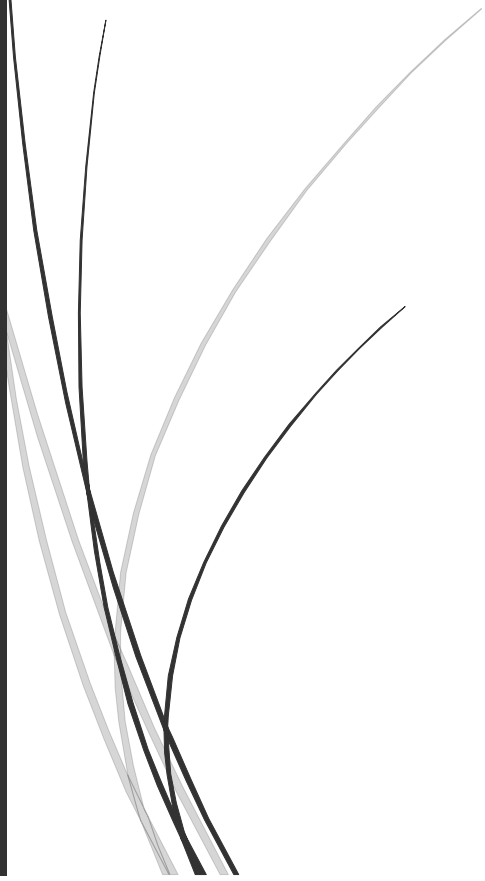
21 de octubre de 2016

Homework 2

Ingeniería de redes y servicios



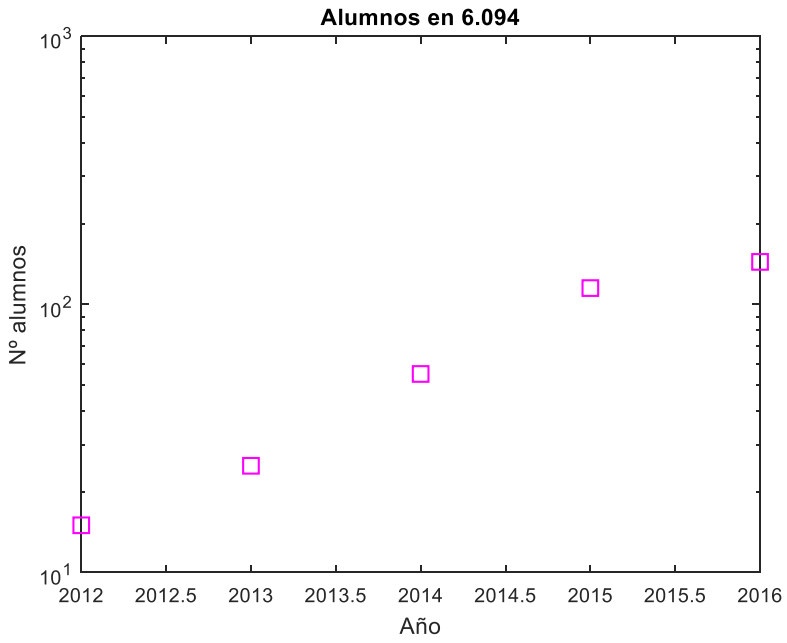
Iago Martínez Colmenero
Juan Francisco García Gómez

Homework 2

ÍNDICE

1 SEMILOG PLOT

En este *script* tenemos que mostrar si una serie de valores conforman una sucesión exponencial. Para ello, trabajamos con gráficas con escala logarítmica. Al ser necesaria la escala logarítmica en el eje *y* nos valemos de **semilogy** y trabajamos como si fuera una función **plot** de la primera práctica.



Se han utilizado los modificadores *s* y *m* para mostrar los puntos como cuadrados y magenta respectivamente. Asimismo, hemos empleado un grosor de línea de 1 y un tamaño de marcador de 10 (con los parámetros *LineWidth* y *Markersize*)

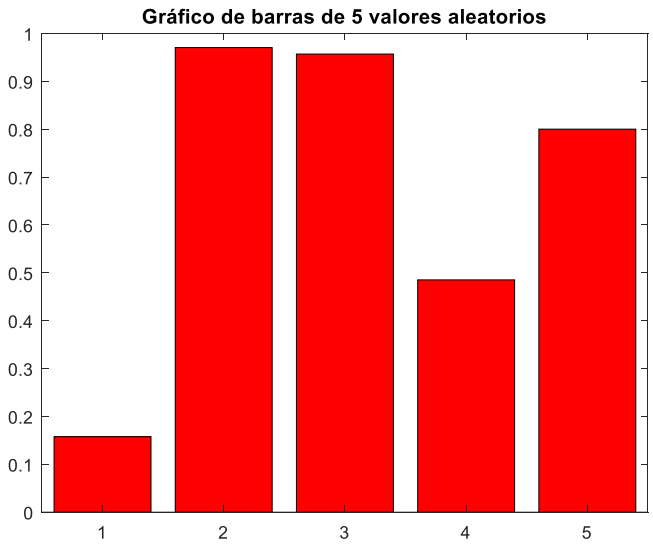
2 BAR GRAPH

La realización de este apartado es tan simple como la utilización análoga a **plot** de la función **bar** con cinco valores aleatorios obtenidos con una función **rand**.

Puesto que considerábamos innecesaria la declaración de una variable, hemos colocado la función **rand** directamente en **bar**. De esta manera, provocamos que dibuje directamente la matriz retornada por **rand**.

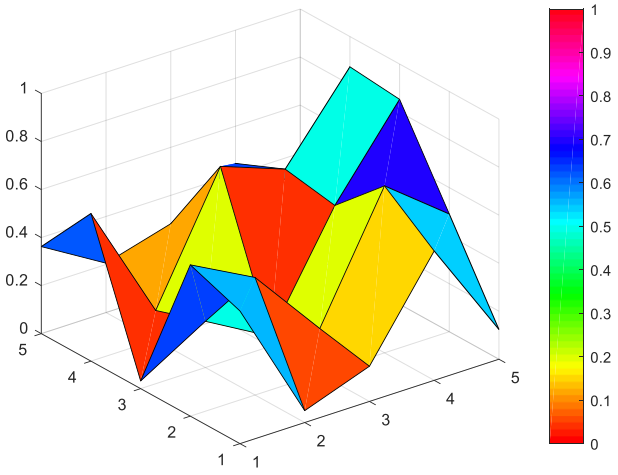
El resultado obtenido en una de las ejecuciones corresponde al gráfico adjunto.

Se ha utilizado el parámetro *r* para su coloreado en rojo.

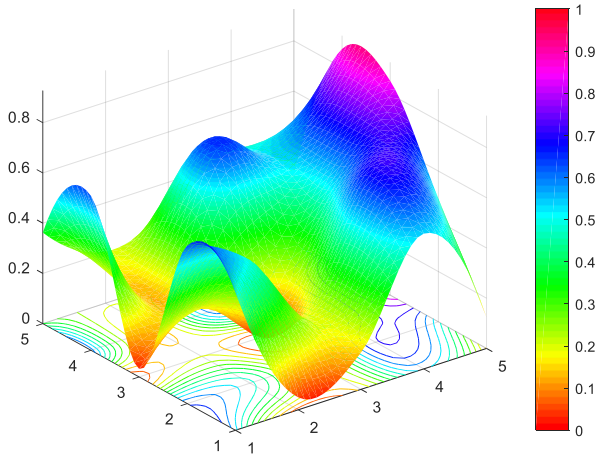


3 INTERPOLATION AND SURFACE PLOTS

Para generar una superficie aleatoria, asignamos a nuestra variable *Z0* una matriz 5x5 de valores aleatorios (utilizando **rand**). Posteriormente empleamos la orden **meshgrid**. Este comando produce coordenadas rectangulares para poder formar las matrices necesarias en la representación. Mediante la orden **surf**, pasando como parámetros las variables *X0, Y0* (obtenidos como resultado del mencionado **meshgrid**) y *Z0* obtenemos el resultado mostrado a continuación. Utilizamos el parámetro *EdgeColor* para marcar la rejilla de color negro (*k*).



Para tratar la superficie y *suavizar* el resultado, generamos vectores *X1* y *Y1* con mismos límites que *X0* y *Y0*, empleando esta vez pasos de 0.1 para tener así más puntos. En estas posiciones definidas realizamos una interpolación cúbica de *X0, Y0* y *Z0* que almacenamos en *Z1*. Análogamente a la superficie anterior, la representamos, con el siguiente resultado:



Para mostrar las líneas de cota hemos empleado el comando **contour**. Hemos utilizado una paleta de colores *hsv* y mostrado una barra de colores a modo de leyenda, con los limites definidos con **caxis**.

4 FUN WITH FIND

4.1 PRIMERA VERSIÓN

En nuestra primera versión (*dissapointedfind.m*), ordenamos mediante la orden **sortrows** el vector que se le pasa a la función. Una vez ordenado, aplicamos un **find** para localizar el índice del primer elemento superior y realizamos una comparación entre el valor de esa posición y la anterior para ver cual se acerca más al valor deseado para así retornar el índice adecuado. Pese a su correcto funcionamiento, detectamos escenarios posibles en los que la función no realizaría una correcta ejecución:

1. En el caso de que se pase una matriz en lugar de un vector, la función devolvería una fila de índices
2. Si el valor más cercano al deseado fuera el primero de la matriz, no sería necesario mirar el índice anterior. Esto supondría una modificación de código que incrementaría el tiempo de ejecución y su complejidad.
3. Revisando el algoritmo, debido al uso de ordenación y variables auxiliares, no solo ocupamos más memoria de la necesaria, sino que también la eficiencia de la función se ve mermada por una implementación más compleja.

Por esta razón, llegamos a la

4.2 SEGUNDA VERSIÓN

Esta versión (*funFind.m*) mucho más simplificada corrige todos los fallos detectados. En primer lugar, definimos una variable *distance* que almacena una matriz/vector con los valores resultantes del valor absoluto de la resta del valor deseado y la matriz/vector sobre la que trabajamos. Esto nos permite tener almacenado lo que se aleja el valor de cada índice respecto del número al que nos queremos aproximar.

Una vez hecho esto, convertimos la matriz de dimensiones m,n a un vector. En caso de que se pase vector, la orden `distanceVec = distance(:);` no hará nada.

Al contrario que en la versión anterior, no ordenamos el vector, puesto que con **min** podemos obtener directamente el menor valor (distancia del valor buscado más pequeña), por tanto, asignamos al índice *ind* el valor retornado por la primera coincidencia que encuentra **find** entre la matriz original y el mínimo valor de *distanceVec*.

De esta manera hemos subsanado todos los fallos detectados en la versión anterior tal y como puede verse en la ejecución con una matriz de dimensiones aleatorias.

5 LOOPS AND FLOW CONTROL

Este *script* saca por pantalla los números de 1 a N, diciendo si es divisible entre 2, 3, 2 y 3 o entre ninguno de los dos. Para ello, en un bucle **for** de 1 a N evaluamos en una serie de **if** sucesivos (**if, elseif** y **else**) si es divisible entre dichos números con la función **mod**.

6 SMOOTHING FILTER

En este script tenemos que implementar una función que devuelva el mismo resultado de la función **smooth.** Para ello, contamos con dos métodos.

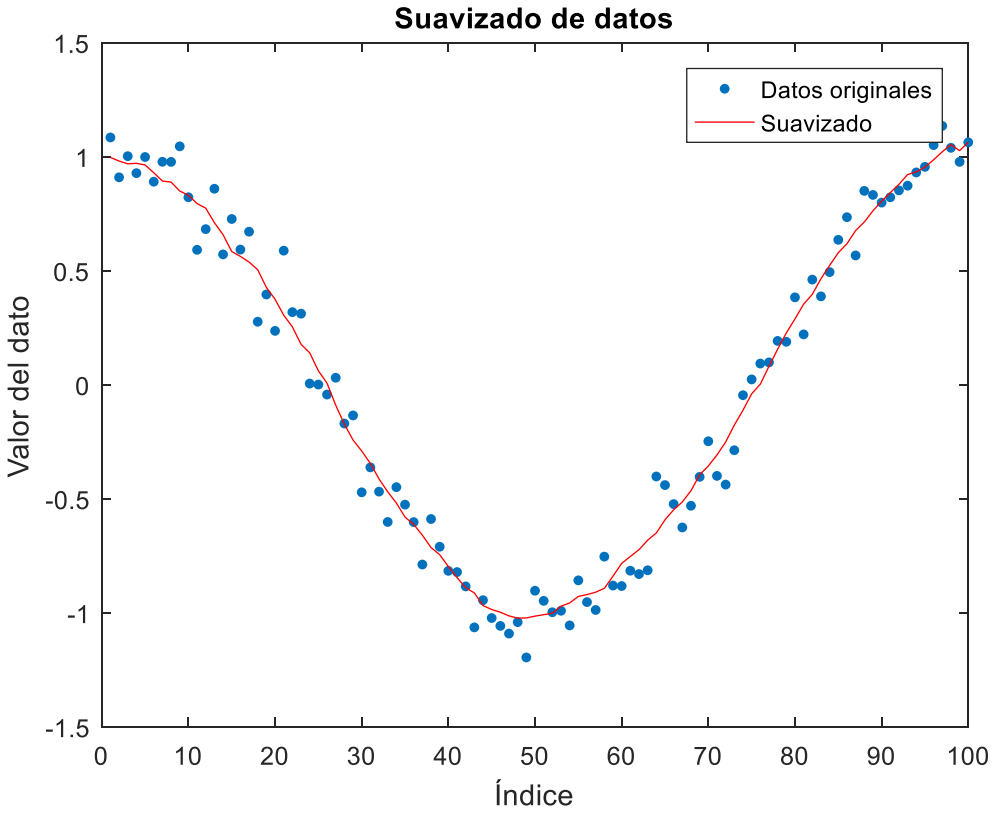
6.1 SUAVIZADO POR MEDIA

Para abordar el problema hemos decidido dividirlo por zonas.

Por un lado, la parte central del vector donde podemos calcular sin problemas medias (con la función **mean**) centradas en el punto deseado con el número de puntos dado por la variable *width*. En estas zonas simplemente hallamos la media del vector que recorremos con un **for** de dichos puntos y la guardamos en el vector *smoothed*.

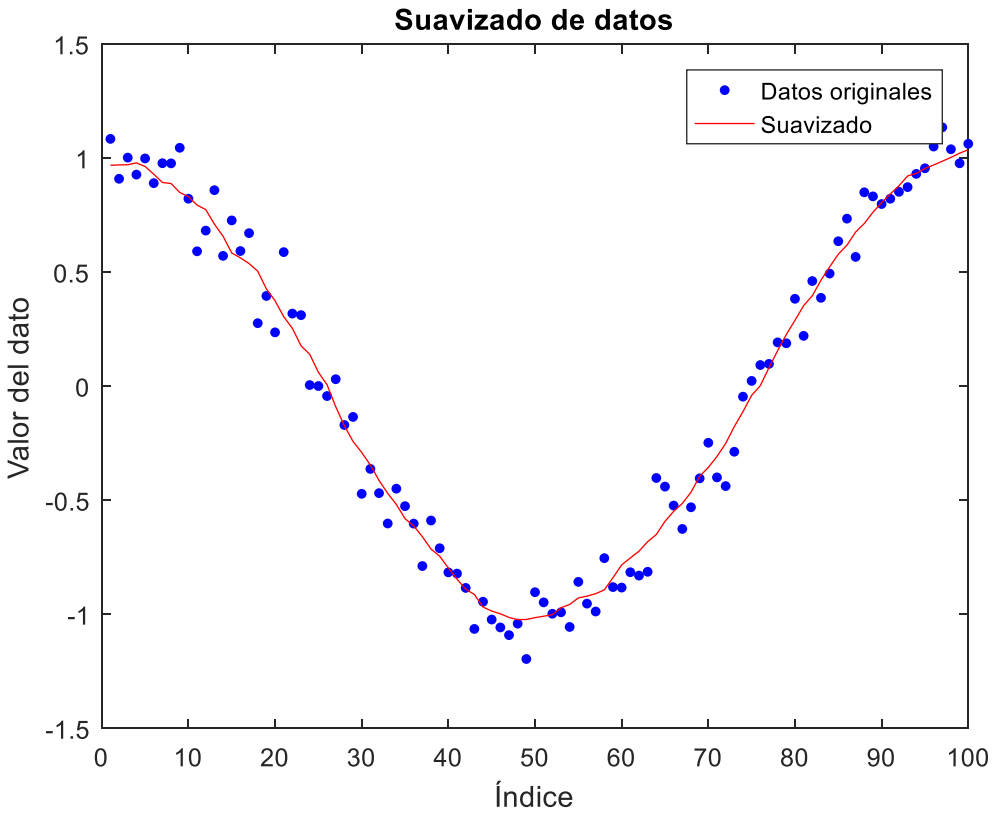
Por otra parte, nos enfrentamos con los extremos, donde al intentar calcular la media, el ancho de *width* hace que excedamos los límites definidos del vector. En estas zonas lo que hacemos es utilizar los puntos que hay en una distancia desde el punto cuya media queremos calcular hasta el extremo del vector, y empleamos dicha distancia como radio para el cálculo de la media. Sabemos que en el caso en que estemos en el extremo del vector este cálculo se realizará tomando un valor más hacia el centro del vector que hacia el extremo del mismo, pero entendemos que es preferible este pequeño sesgo a realizar una media con un número inferior de puntos.

Obtenemos el siguiente resultado.



6.2 SUAVIZADO POR CONVOLUCIÓN

Como mejora de esta función, hemos decidido emplear la convolución para evitar el uso de iteraciones superfluas. Para ello, utilizamos la función **conv** con un pulso de anchura *width* con valor *1/width* para que el resultado sea similar que el que obtendríamos con una media. Esto supone que en los extremos no obtengamos el resultado adecuado, pero podemos corregirlo multiplicando dichos extremos por una matriz de ajuste obtenida como `width./[widthM+1:width-1,width-1:-1:widthM+1]` donde *widthM* es *floor(width/2)*. Puesto que no consideramos necesaria la declaración de dichas variables, en el código lo tenemos sustituido por el contenido de éstas.



7 OPCIONALES

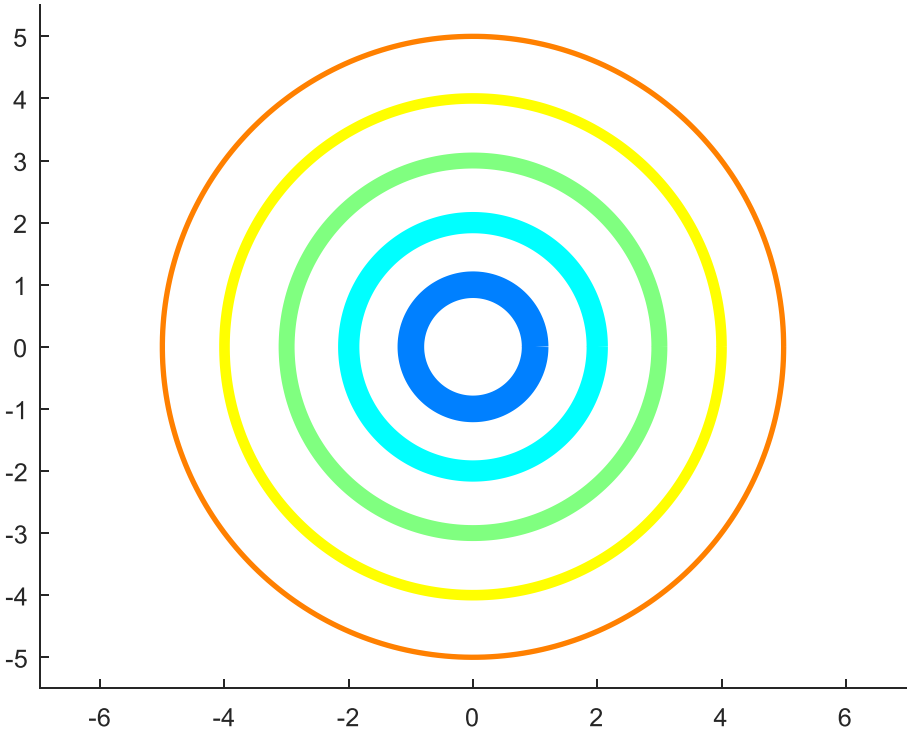
7.1 PLOT A CIRCLE

7.1.1 getCircle.m

Para conseguir el ploteo de la circunferencia, usamos los fundamentos del cambio a coordenadas polares. De esta forma podemos definirla a través de su centro y su radio. Aprovechamos las ecuaciones del cambio a polares para añadir las coordenadas del centro y así trasladar la circunferencia del origen al punto deseado.

7.1.2 concentric.m

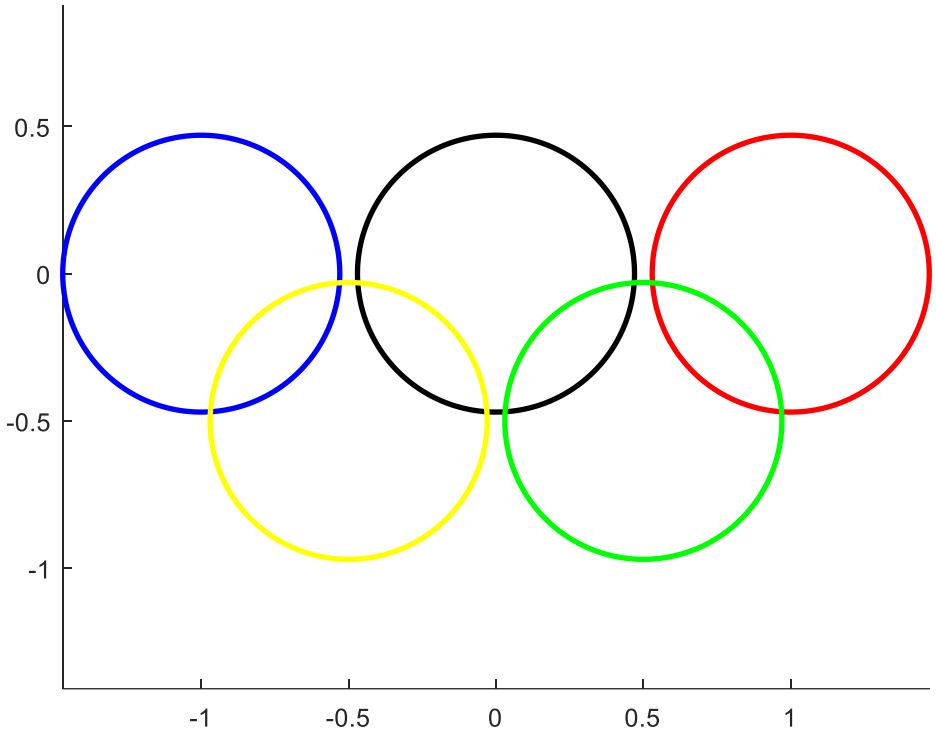
Al tener definida la función **getCircle** en un script externo, podemos invocarlo desde *concentric.m*. Por lo tanto, generamos los 5 círculos llamando a la función dentro de un bucle **for**. La representación por defecto de la función **plot** provocaba que tuviéramos un resultado visual "achatado". Inicialmente lo corregimos utilizando el parámetro *Position* en **figure** en el cual le pasamos un vector de 4 valores en los que definimos la posición de la ventana generada y las dimensiones de esta. Finalmente, vimos una solución más simplificada con el comando **axis** *equal*.



Para trabajar con la gama de colores de *jet* generamos un vector con la función **jet** y vamos asignando en cada iteración el color.

7.1.3 olympic.m

En este caso trabajamos de forma análoga al apartado *concentric.m* pero jugando con los centros de las circunferencias en lugar de con los radios.



Para la selección de colores hemos creado un vector de valor `'bkryg'` de manera que en cada iteración del bucle que genera los anillos vaya seleccionando el color adecuado.

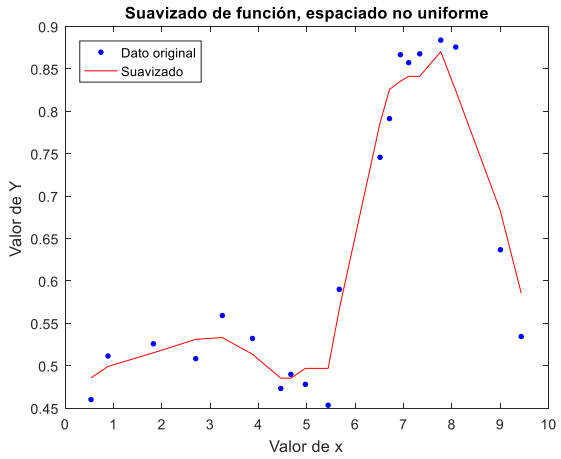
7.2 THROW A BALL 2

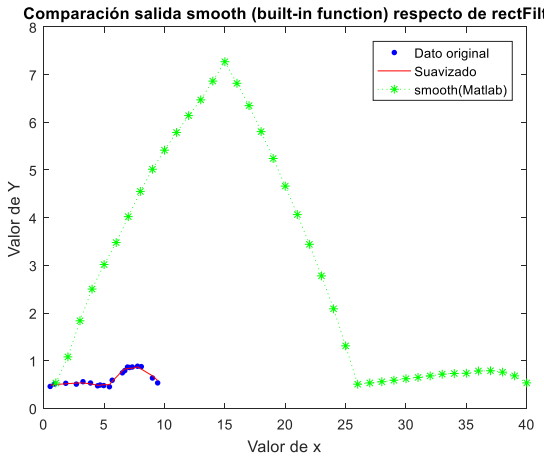
De forma análoga a lo que hemos hecho en scripts como los referentes a *getCircle.m*, modificamos la función para que pueda ser invocada desde scripts externos.

7.3 SMOOTHING NONUNIFORMLY SAMPLED DATA

Considerando que la función **rectFilt** que hemos explicado en el apartado 6 puede ser invocado desde otras funciones, hemos hecho la función externa en un script *rectFilt.m* y sobre ella hemos añadido un **if** para evaluar si los datos dados son un vector o una matriz de valores. De no ser un vector, lo convertimos en uno para facilitar su análisis.

Debido a que en este caso la frecuencia de muestreo no es necesariamente constante, no podemos utilizar los métodos descritos anteriormente de convolución. Sin embargo, podríamos sustituir el código dado por una interpolación evitando así, de nuevo, un **for**.





Finalmente, como puede verse en las imágenes, hemos comparado la función **smooth** implementada por defecto en MatLab con la función descrita en este apartado.

7.4 BUY AND SELL A STOCK

En este ejercicio contamos con la evolución temporal del precio de las acciones de Google a lo largo del tiempo. Además, tenemos dos vectores que contienen los índices del vector de precios coincidentes con máximos y mínimos respectivamente.

Como es lógico, nuestro primer movimiento tendrá que ser de compra de acciones, y venderemos y compraremos acciones según convenga y nos permita nuestro presupuesto inicial. Hemos de tener en cuenta que cada transacción de compraventa de títulos supone un coste de 12.95 $.

Lo primero que hacemos es unir (en orden ascendente con la orden **sort**) los índices de compra y venta en una misma matriz. De esta manera, tenemos una matriz con los índices de los eventos en los que conviene realizar un movimiento. Una vez hecho esto, recorremos dicho vector en un bucle **for** y evaluamos en si es un evento de compra o venta.

En caso de que se trate de un movimiento de compra, comprobaremos que tenemos dinero suficiente como para comprar una acción (contando con su coste) y de ser así, ayudándonos de la función **floor** almacenamos en una variable el número de acciones que tenemos descontando el dinero correspondiente de la inversión inicial o activos restante.

Por otro lado, en los movimientos de venta evaluaremos si vale la pena vender las acciones (tenemos que tener en cuenta nuevamente el coste) y descontaremos acciones en caso de venta, acumulando el resultado correspondiente.

Finalmente, devolvemos el resultante y lo imprimimos por pantalla.