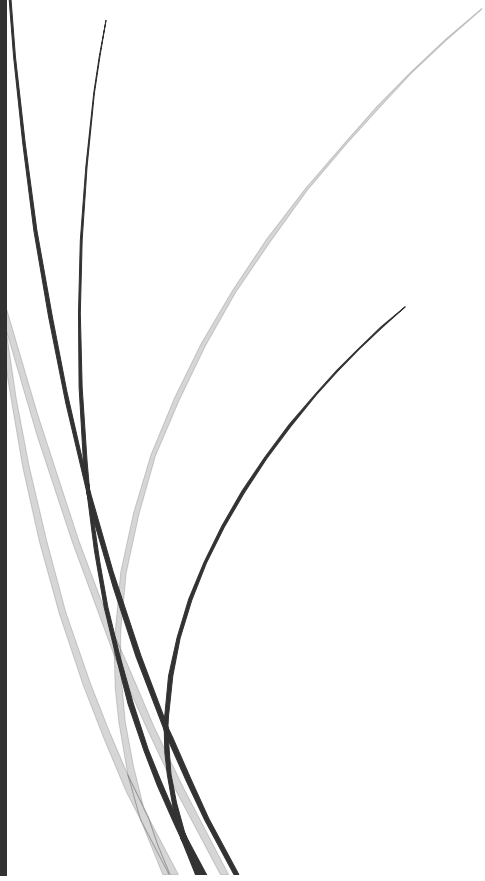
19 de noviembre de 2016

# Homework 4

## Ingeniería de redes y servicios



Iago Martínez Colmenero
Juan Francisco García Gómez

# Homework 4

## ÍNDICE

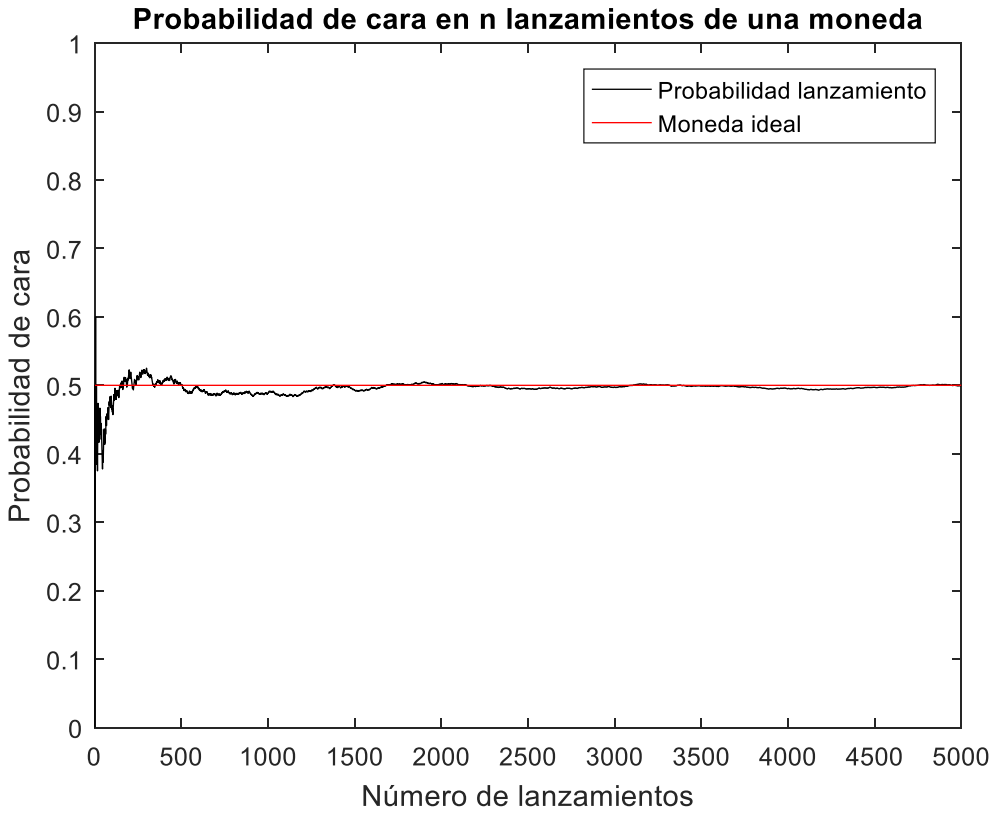

1 Variables Aleatorias ..... 2
2 Lanzamiento de moneda ..... 2
3 Histograma ..... 3
4 Práctica con *cells* ..... 3
5 Usando estructuras ..... 4
6 Excepciones ..... 4
7 Procesado de imagen ..... 5
7.1 Primera versión ..... 5
7.2 Segunda versión ..... 5
7.3 Versión final ..... 5
8 Movimiento Browniano ..... 7
9 Opcional: Animación conjunto de Julia ..... 8

# 1 VARIABLES ALEATORIAS

En este script generamos un vector de 500 valores con **randn** y modificamos su desviación típica multiplicando por el valor deseado y desplazamos su media al sumarle el valor de media buscado.

# 2 LANZAMIENTO DE MONEDA

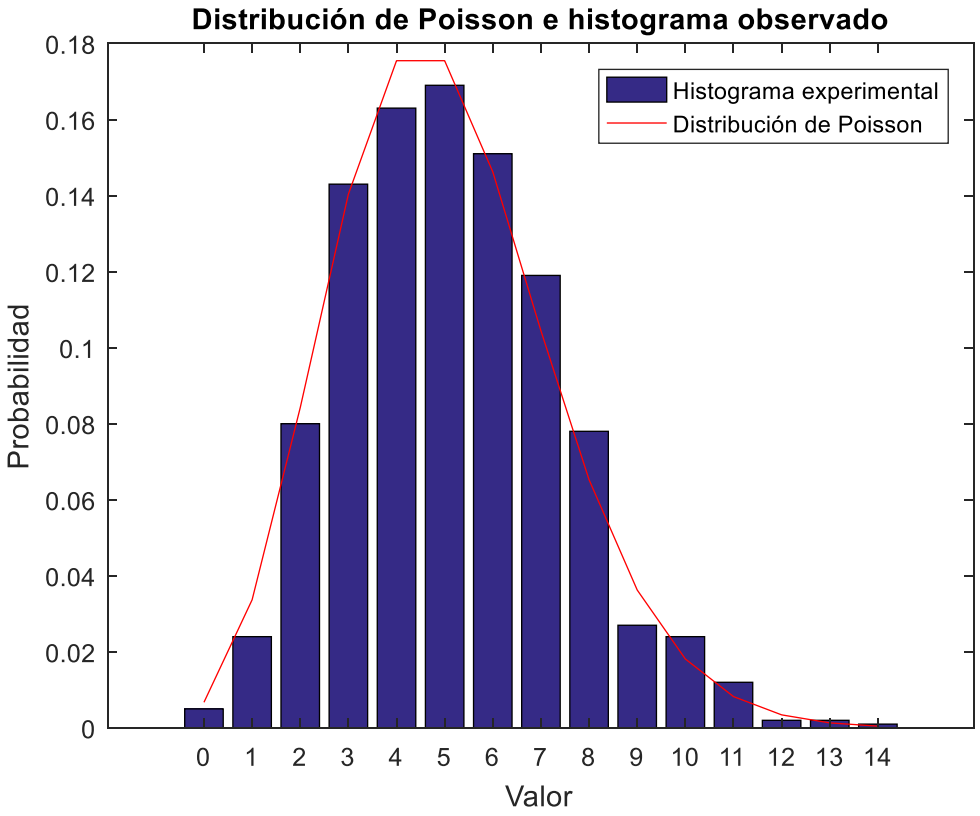
En el desarrollo de esta función, generamos un vector de 5000 valores aleatorios. Después, redondeamos y escogemos el criterio de cara=1 y cruz=0, de esta manera, la suma acumulativa (**cumsum**) del vector redondeado dará el número de caras que hayamos obtenido en n lanzamientos, que, al dividir por el número de lanzamientos que llevamos, nos dará, aplicando la definición de probabilidad de NECESITAMOS ACORDARNOS DE ESTO, la probabilidad de cara en n lanzamientos de nuestra moneda. Al plotear, tal y como muestra la figura, vemos como tiende a una probabilidad de 0,5, como es lógico en un suceso de posibilidades equiprobables.



# 3 HISTOGRAMA

Guardamos en un vector R una distribución de Poisson de λ=5 con 1000 valores. A continuación, con el comando **hist** obtenemos un histograma con la frecuencia de repetición de los valores en intervalos y guardamos los resultados en histog. Además, almacenamos los centros que hemos usado para separar los intervalos en centers. Seguidamente, normalizamos histog y dibujamos el gráfico de barras de la misma.

Para terminar, hacemos un **plot** del resultado de pasar nuestro vector a **poisspdf**, dibujando la gráfica de la distribución.




MATLAB recomienda el uso de **histogram** en lugar de **hist**.

# 4 PRÁCTICA CON *CELLS*

En este script trabajamos con *cells*. Para la declaración de esta utilizamos llaves {} separando cada campo por comas y cada fila por punto y coma.

Para referirnos a un elemento concreto de una celda, utilizamos la sintaxis de `nombreCelda{fila, columna}`. De esta manera, podemos modificar el apellido de la señorita Brown directamente, así como de actualizar el sueldo de Pat.

# 5 USANDO ESTRUCTURAS

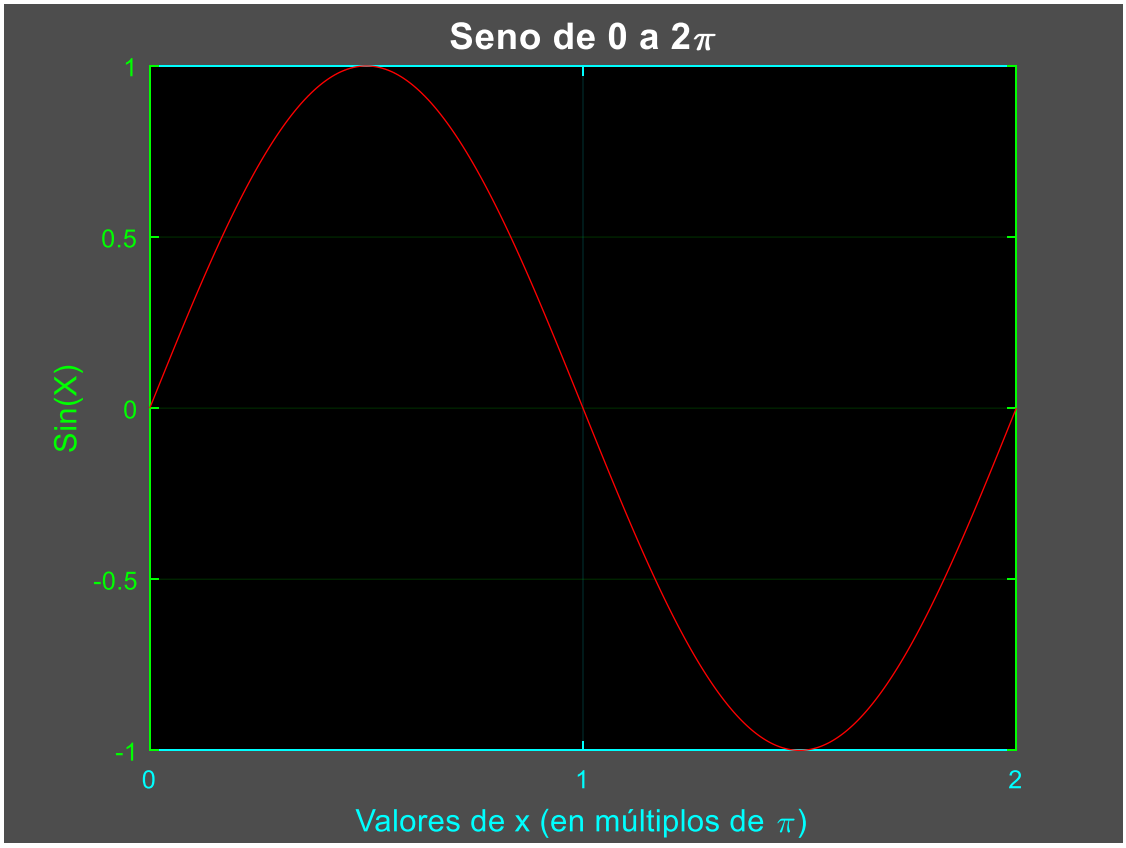
Respondiendo a la pregunta indicada en el enunciado, a es un struct array de 17x1. Los nombres de sus campos son: name, folder, date, bytes, isdir y datenum.

En la función *displayDir.m*, almacenamos en una estructura el nombre, carpeta, fecha de creación, bytes, el flag de directorio y fecha de modificación (como serial de MATLAB). Después, recorremos la estructura y, en caso de no ser un directorio, mostramos el nombre de directorio y el espacio que ocupa dicho fichero.

# 6 EXCEPCIONES

En *handlesPractice.m* utilizamos **set** para modificar las propiedades que nos van pidiendo. Al especificar gca, trabajamos con los ejes (XTick, xticklabel, xcolor, color, etc). Con gcf actuamos sobre propiedades de la propia figura. En nuestro caso, el color.

Además de esto, añadimos una cuadrícula con **grid** y damos formato al texto con los parámetros de los **label.**



# 7 PROCESADO DE IMAGEN

En esta función hemos tenido que tener en cuenta a lo largo de diferentes versiones que el formato correcto para guardar los datos de una imagen es **uint8**. Independientemente de que versión utilizásemos, la función *displayRGB.m* es llamada en un script (*RGBTest.m)* que la representa usando **image** con los modificadores `axis tight; axis equal;`

## 7.1 PRIMERA VERSIÓN

En esta primera versión, no supimos interpretar correctamente el por qué utilizar interpolación para redimensionar una imagen por lo que empleábamos la orden **imresize** para escalar la imagen introducido a 800 px como máximo en el mayor lado.

Una vez hecho esto, reservábamos espacio con una variable del doble de dimensiones con **zeros**. Una vez hecho esto, recorremos cada pixel de la imagen con la que trabajamos y asignamos su valor al mismo punto en la imagen de ceros.

Finalmente, en un bloque de 3 for recorríamos igualmente cada pixel, pero cogiendo solo uno de los valores (RGB) para asignarlo al pixel de la imagen negra correspondiente. Para esto nos ayudábamos de dos vectores booleanos auxiliares para manejar el desplazamiento que tenía que tener dicho pixel finalmente (Para que así estuviera bien colocado).

## 7.2 SEGUNDA VERSIÓN

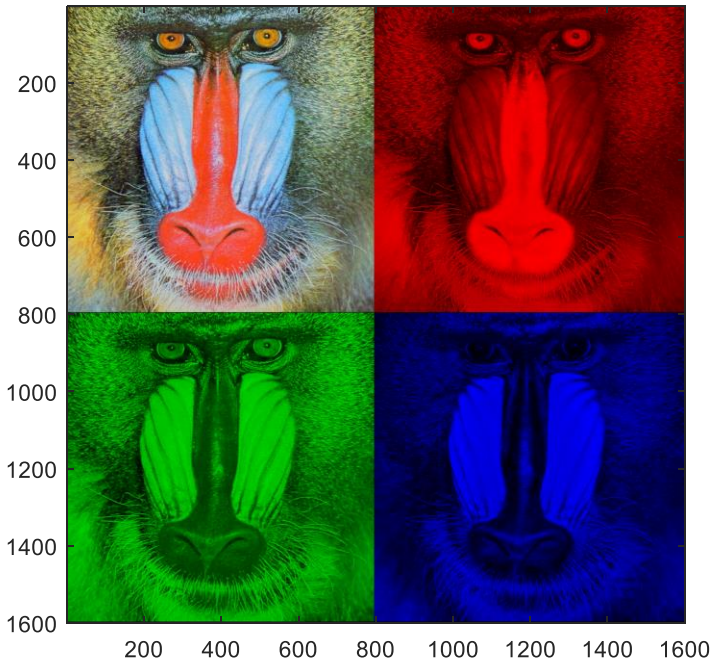
En una segunda versión nos ahorrábamos los bucles trabajando directamente igualando vectores tal y como habíamos hecho en prácticas anteriores. El resultado es el mismo, pero mucho más eficiente en consumo de memoria.

## 7.3 VERSIÓN FINAL

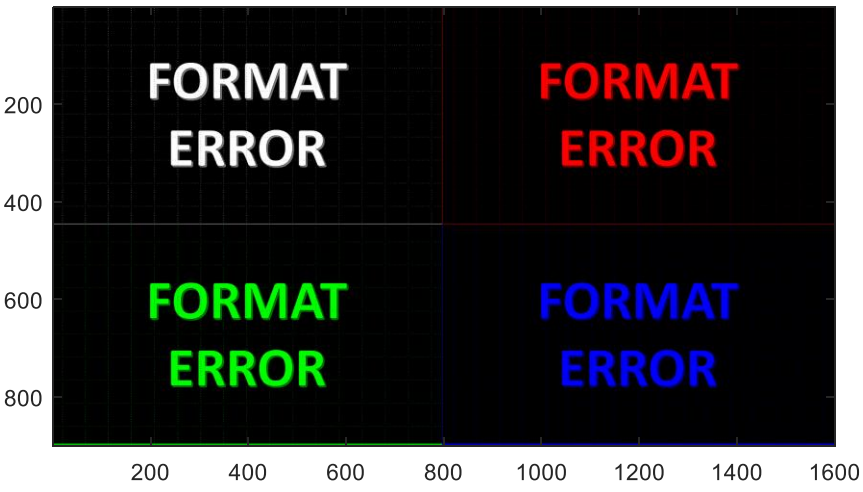
En esta versión evitamos el uso de **imresize** y trabajamos con interpolación.

En primer lugar, guardamos en una variable la relación de aspecto de la imagen (esto nos permitirá decidir qué lado es el que hay que redimensionar). Después, análogamente al script *randomSurface.m* de una práctica anterior, hacemos dos meshgrid; uno sobre las dimensiones originales y otro sobre las dimensiones que queremos que tenga la imagen. Reservamos espacio en una variable con una matriz de ceros para aumentar la eficiencia del programa y asignamos a es. En una variable (en la que hemos reservado el correspondiente espacio con **zeros** para mejorar la eficiencia del programa) asignamos el resultado de interpolar la imagen original con los meshgrid mencionados. Hay que tener en cuenta que para el correcto funcionamiento de **interp2** realizamos una conversión *double* de dichos datos. Por esta razón, finalmente convertimos a **uint8**.

Una vez hecho esto guardamos por separado las capas R G B y devolvemos la imagen como resultado de la concatenación de cada una de las matrices mencionadas. Obtenemos el siguiente resultado:




Por otro lado, el enunciado nos especifica que trabajemos solo con aquellos archivos con extensión .jpg. Para ello, buscamos con **find** el último punto de la cadena de caracteres del nombre de fichero (así sabemos a partir de donde empieza la extensión) y observamos si es *.jpg. En caso de no serlo, se muestra un ejemplo de ejecución con un mensaje de error.

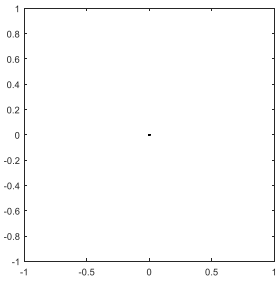
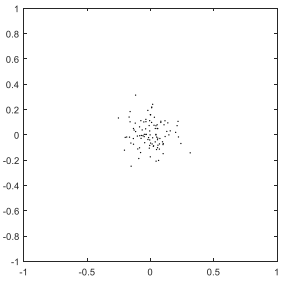
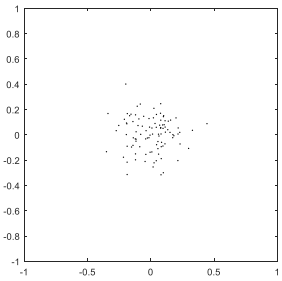


# 8 MOVIMIENTO BROWNIANO

En *brown2D.m* definimos una matriz de Nx2 iniciado a cero que representa las coordenadas en las que se sitúan cada uno de los puntos. Una vez hecho esto, representamos y guardamos la propia línea en *L*. Esto, como es lógico, nos representa un único punto en el centro (aunque tenemos N puntos, al estar todos situados en (0, 0) se ve un único punto).

Finalmente, en un bucle **for** definimos en cada iteración una distribución normal con desviación típica 0.005 con **randn** (con el mismo tamaño que la matriz de puntos y se lo sumamos a la matriz de puntos inicial. Con **set** modificamos los valores de *L*. Para testearlo basta con hacer la llamada a la función.

Como curiosidad, cabe mencionar que si cerramos la representación antes de que finalice saltará un error. Esto es debido a que el **set** no tiene una gráfica sobre la que actuar.

# 9 OPCIONAL: ANIMACIÓN CONJUNTO DE JULIA

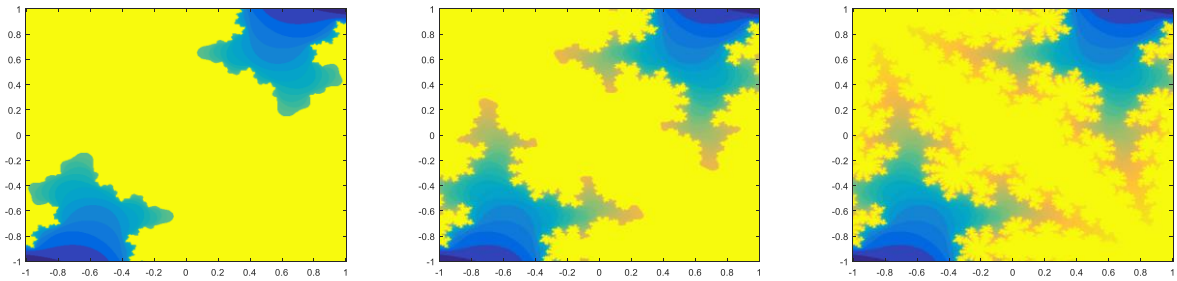
En este ejercicio simplemente seguimos el guion de la práctica. Lo primero que hacemos es el vector temporal en el que guardamos 500 valores dentro del radio de *zMax*. Tras ello usamos **meshgrid** para crear las matrices R e I de los espacios real e imaginario.

A continuación, en la variable Z vamos a almacenar el resultado del conjunto $\mathbb{C}$. Reservamos espacio en la matriz M asignándole como valor a sus puntos N (número máximo de iteraciones antes del escape).

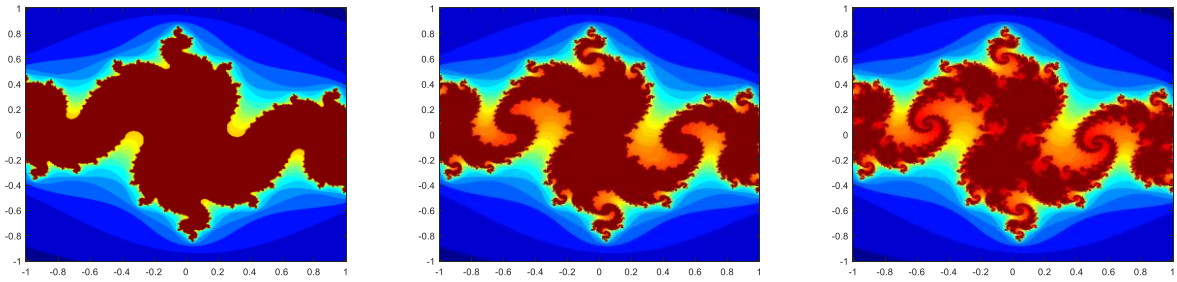
Recorremos el bucle de representación, en el calculamos Z para cada punto. Para ello, almacenamos en *indices* los puntos en los que la velocidad tiende a infinito y sustituimos esos puntos en la matriz M por la iteración en la estamos, al mismo tiempo que anulamos esos puntos de la matriz Z.

Para terminar, creamos la imagen con **imagesc**. Con **drawnow** conseguimos que plotee sin terminar de calcular la matriz entera (así dibujamos cada iteración). Al invocar a dichas funciones en cada iteración al cerrar ventana de representación antes de que ésta acabe se seguirá abriendo la figura con los parámetros por defecto.

```
julia(.35,-.297491+.641051i,100);
```



```
julia(1,-.8+.156i,100);
```



```
julia(1,-.4+.6i,100);
```