Práctica PRIME

Table of Contents

Autores: Alfredo Sánchez Sánchez y Manuel Mora de amarillas.

El proyecto consiste en realizar un modelado de la simulación de la transmisión de tramas de Carga (Payload) según el estándar de PRIME, ITU-T G.9904. No se tendrá en cuenta la parte de la trama destinada a la transmisión del preámbulo (Preamble), Encabezado (Header) ni del CRC.

Apartado 1: Implementación de todos los modos de comunicación de PRIME en el caso de canal sin distorsión y sin FEC

Apartado 1.1: Elección de parámetros de simulación

El numero de bits que se busca transmitir, deben ser superiores a 10^4 o 10000 debido a que se desea una simulación fiable con un BER cercano a 10^-4. En nuestro caso, hemos seleccionado como límite 20000, ya que es un valor fiable para todos los tipos de modulación, especialmente paa la modulación de 2, que es la que enviaría un menor número de bits. El número de tramas que enviaremos será 4, sabiendo que con DBPSK el número de símbolos a transmitir es el doble que en DQPSK y este es el doble que con D8PSK. El motivo por el que queremos enviar un número elevados de bits es que queremos hacer que la simulación, se asemeje más a los cálculos teóricos, pués cuantos más bits se envíen, el factor aleatorio va desapareciendo.

Además, el número de símbolos de OFDM por trama serán 63, que es el máximo que permite el standard PRIME en sus especificaciones con el fin de que la transmisión sea lo más rápida que se pueda.

Para terminar, el número de portadoras por simbolo de OFDM será de 96, tal y como se da en el standard. Esto siginidca que por cada símbolo OFDM hay 96 símbolos modulados en DBPSK, DQPSK o D8PSK.

```
clear; close all, format compact
NFFT  = 512;  % Tamaño de la FFT
Fs    =  25600;  % Frecuencia de muestreo
df    =  Fs/NFFT ;  % Separación entre portadoras
Nf    = 96;  % Numero de portadoras con datos (+1 por el piloto)
N_tramas = 4;
Nofdm = 63; % Número de símbolos OFDM por trama
% El número de bits totales será: N_bits_totales =
```

```
% Nf*Nofdm*log2(M)*N_tramas, donde M será 2, 4, 8 dependiendo del tipo
 de
% modulación.
```

Apartado 1.2.1: Demostrar que en ausencia de ruido sin FEC, sin prefijo cíclico y sin aleatorización, no se generá errores.

```
NFFT  = 512;  % Tamaño de la FFT
Fs    =  256000;  % Frecuencia de muestreo
df    =  Fs/NFFT ;  % Separación entre portadoras
Nf    = 96;  % Numero de portadoras con datos (+1 por el piloto)
N_tramas = 10;
Nofdm = 63; % Número de símbolos OFDM por trama
BERDBPSK_Total = [];
BERDQPSK_Total = [];
BERD8PSK_Total = [];

for i = 1:N_tramas
    [txbitsDBPSK,xDBPSK] = ModulacionOFDM(2, Nf, NFFT, Nofdm);
    [txbitsDQPSK,xDQPSK] = ModulacionOFDM(4, Nf, NFFT, Nofdm);
    [txbitsD8PSK,xD8PSK] = ModulacionOFDM(8, Nf, NFFT, Nofdm);
    % Demodulacion sin ruido
    BERDBPSK=CalcularError(xDBPSK, Nf, NFFT, 2 ,Nofdm, txbitsDBPSK);
    BERDQPSK=CalcularError(xDQPSK, Nf, NFFT, 4 ,Nofdm, txbitsDQPSK);
    BERD8PSK=CalcularError(xD8PSK, Nf, NFFT, 8 ,Nofdm, txbitsD8PSK);
    BERDBPSK_Total = [BERDBPSK_Total BERDBPSK];
    BERDQPSK_Total = [BERDQPSK_Total BERDQPSK];
    BERD8PSK_Total = [BERD8PSK_Total BERD8PSK];

end
BERDBPSK_Total = sum(BERDBPSK_Total)
BERDQPSK_Total = sum(BERDQPSK_Total)
BERD8PSK_Total = sum(BERD8PSK_Total)

BERDBPSK_Total =
     0
BERDQPSK_Total =
     0
BERD8PSK_Total =
     0
```

Efectivamente, anteriormente se muestra que el error es nulo y tiene todo el sentido, puesto que no se introduce ningún tipo de distorsión ni ruido en ningún punto entre la señal transmitida y la recibida. Todos los bloques simulados funcionan correctamente y por tanto, no añaden ningún error, pués los bloques en sí no añaden error.

Apartado 1.2.2: Demostrar que en ausencia de ruido sin FEC, sin prefijo cíclico y con aleatorización, no se generá errores.

En este apartado, se añadirá la parte de Aleatorización y Dealeatorización, con una función Scrambler.

```
% Ruido para el apartado 1.2.3:
BERDBPSK_Total_Scrambler = [];
BERDQPSK_Total_Scrambler = [];
BERD8PSK_Total_Scrambler = [];
SNR_vector = 0:20;
for i = 1:N_tramas
    [txbitsDBPSK,xDBPSK] = ModulacionOFDMScrambler(2, Nf, NFFT,
 Nofdm);
    [txbitsDQPSK,xDQPSK] = ModulacionOFDMScrambler(4, Nf, NFFT,
 Nofdm);
    [txbitsD8PSK,xD8PSK] = ModulacionOFDMScrambler(8, Nf, NFFT,
 Nofdm);
    % Demodulacion con Scrambler
    BERDBPSKScrambler=CalcularErrorScrambler(xDBPSK, Nf, NFFT,
 2 ,Nofdm, txbitsDBPSK);
    BERDQPSKScrambler=CalcularErrorScrambler(xDQPSK, Nf, NFFT,
 4 ,Nofdm, txbitsDQPSK);
    BERD8PSKScrambler=CalcularErrorScrambler(xD8PSK, Nf, NFFT,
 8 ,Nofdm, txbitsD8PSK);
    BERDBPSK_Total_Scrambler = [BERDBPSK_Total_Scrambler
 BERDBPSKScrambler];
    BERDQPSK_Total_Scrambler = [BERDQPSK_Total_Scrambler
 BERDQPSKScrambler];
    BERD8PSK_Total_Scrambler = [BERD8PSK_Total_Scrambler
 BERD8PSKScrambler];
end
BERDBPSK_Total_Scrambler = sum(BERDBPSK_Total_Scrambler)
BERDQPSK_Total_Scrambler = sum(BERDQPSK_Total_Scrambler)
BERD8PSK_Total_Scrambler = sum(BERD8PSK_Total_Scrambler)

BERDBPSK_Total_Scrambler =
     0
BERDQPSK_Total_Scrambler =
     0
BERD8PSK_Total_Scrambler =
     0
```

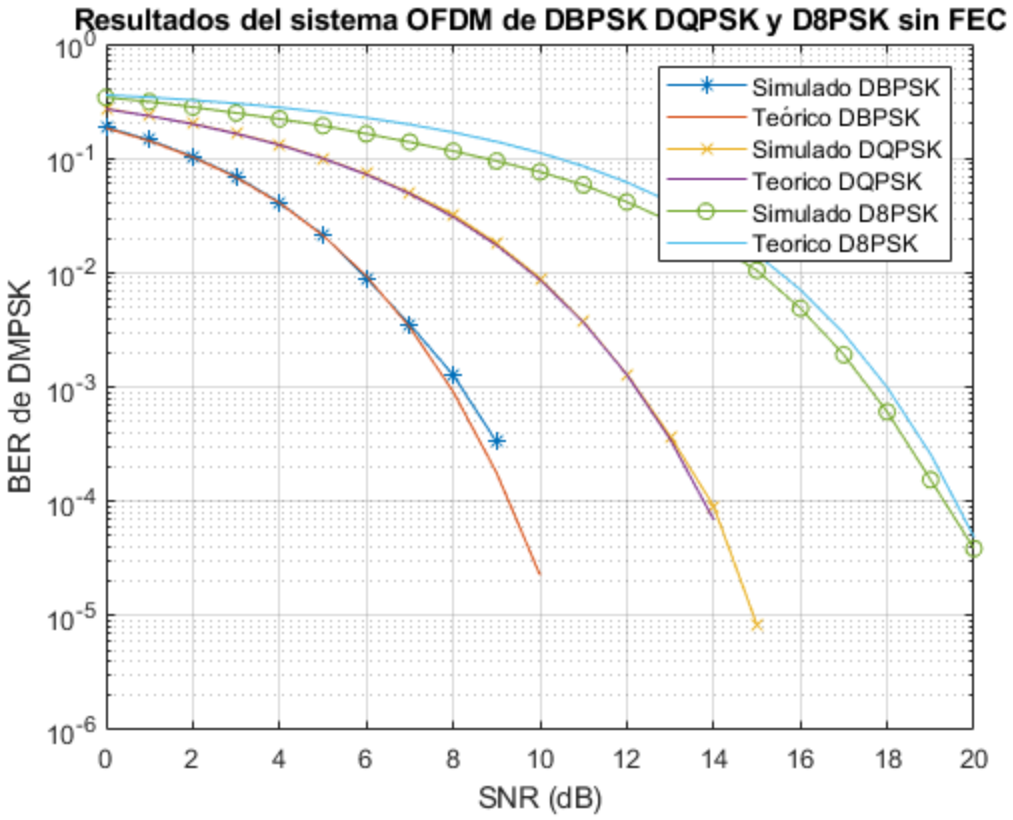
Como ocurre en el apartado anterior, el BER es 0, pués como en el caso anterior, no se introducía ningún error, en este caso y solo al añadir la aleatorización que tampoco introduce ni distorsión, ni errores, ni ruido, al compararar los bits transmitidos y los recibidos, el BER sale 0.

Apartado 1.2.3: Curvas BER vs SNR teóricas y simuladas, estas ultimas empleando secuencias de bits seudoaleatorias.

En este apartado, para ver como responde el sistema a interferencia de ruido blanco, se añade dicho ruido a la señal antes de ser introducida en el receptor. Se representará la final el error obtenido para distintos valores de SNR.

```
for i = 1:N_tramas
    % Inicialización de vectores:
    BER_DBPSK=zeros(N_tramas,length(SNR_vector));
    BER_DQPSK=zeros(N_tramas,length(SNR_vector));
    BER_D8PSK=zeros(N_tramas,length(SNR_vector));
    % Se añase ruido:
    BERDBPSKR=CalcularErrorRuido(xDBPSK, Nf, NFFT, 2 ,Nofdm,
 txbitsDBPSK, SNR_vector);
    BERDQPSKR=CalcularErrorRuido(xDQPSK, Nf, NFFT, 4 ,Nofdm,
 txbitsDQPSK, SNR_vector);
    BERD8PSKR=CalcularErrorRuido(xD8PSK, Nf, NFFT, 8 ,Nofdm,
 txbitsD8PSK, SNR_vector);
    BERDBPSKRuido(i,:) = BERDBPSKR;
    BERDQPSKRuido(i,:) = BERDQPSKR;
    BERD8PSKRuido(i,:) = BERD8PSKR;
end
BERDBPSKRuido_avg = zeros(1,length(SNR_vector));
BERDQPSKRuido_avg = zeros(1,length(SNR_vector));
BERD8PSKRuido_avg = zeros(1,length(SNR_vector));

for i=1:N_tramas
      BERDBPSKRuido_avg = BERDBPSKRuido_avg + BERDBPSKRuido(i,:);
      BERDQPSKRuido_avg = BERDQPSKRuido_avg + BERDQPSKRuido(i,:);
      BERD8PSKRuido_avg = BERD8PSKRuido_avg + BERD8PSKRuido(i,:);
end
BERDBPSKRuido_avg = BERDBPSKRuido_avg./N_tramas;
BERDQPSKRuido_avg = BERDQPSKRuido_avg./N_tramas;
BERD8PSKRuido_avg = BERD8PSKRuido_avg./N_tramas;

theoryBerDBPSK = DBPSK_BER(SNR_vector);
theoryBerDQPSK = DQPSK_BER(SNR_vector);
theoryBerD8PSK = D8PSK_BER(SNR_vector);

theoryBerDBPSK(find(theoryBerDBPSK<1e-5)) = NaN;
theoryBerDQPSK(find(theoryBerDQPSK<1e-5)) = NaN;
theoryBerD8PSK(find(theoryBerD8PSK<1e-5)) = NaN;

figure

semilogy(SNR_vector, BERDBPSKRuido_avg, '-*');  hold on
semilogy(SNR_vector, theoryBerDBPSK);
semilogy(SNR_vector, BERDQPSKRuido_avg, '-x');
semilogy(SNR_vector, theoryBerDQPSK);
```

```
semilogy(SNR_vector, BERD8PSKRuido_avg, '-o');
semilogy(SNR_vector, theoryBerD8PSK);

legend('Simulado DBPSK','Teórico DBPSK', 'Simulado DQPSK', 'Teorico
 DQPSK', 'Simulado D8PSK', 'Teorico D8PSK')
xlabel('SNR (dB)');  ylabel('BER de DMPSK')
grid on
title('Resultados del sistema OFDM de DBPSK DQPSK y D8PSK sin FEC')
```



El sistema PRIME implementado hasta este momento tiene un comportamiento frente al ruido parecido al de la práctica 4 de OFDM, pues no se ha añadido ningún bloque de correción de errores.

Representando las curvas de BER teóricas frente a las curvas de BER simuladas, se aprecia como ambas son similares (menos en el caso de D8PSK que la práctica sale algo desplezada). El resultado es el esperado, ya que en ausencia de FEC, el error cometido para un determinado ruido blanco es igual con PRIME a una modulación OFDM normal y corriente.

Tal y como se ha visto en otras prácticas, para un mismo nivel de BER se necesita mayor SNR en aquellas modulaciones que utilizan mayor número de símbolos (D8PSK), esto es debido a que la separación entre ellos es menor. Para la modulación DBPSK tanto la teórica como la práctica alcanzan unos valoraes de BER de 10^-4 para un SNR de 10dB aproximadamente. Con la modulación DQPSK se alcanza en los 14dB (5 dB más prácticamente). Y con D8PSK se aumenta la SNR se necesitan aproximadamente 20dB para alcanzarlos (el aumento es prácticamente el mismo que el aumento anterior)

```
clear all;
```

Apartado 2: Implementación de todos los modos de comunicación de PRIME en el caso de canal sin distorsión y sin FEC.

Así como en las implementaciones anteriores se ha supuesto un canal invariante en frecuencia, en este apartado se asumirá que el canal cambia con la frecuencia y que se producen retardos.

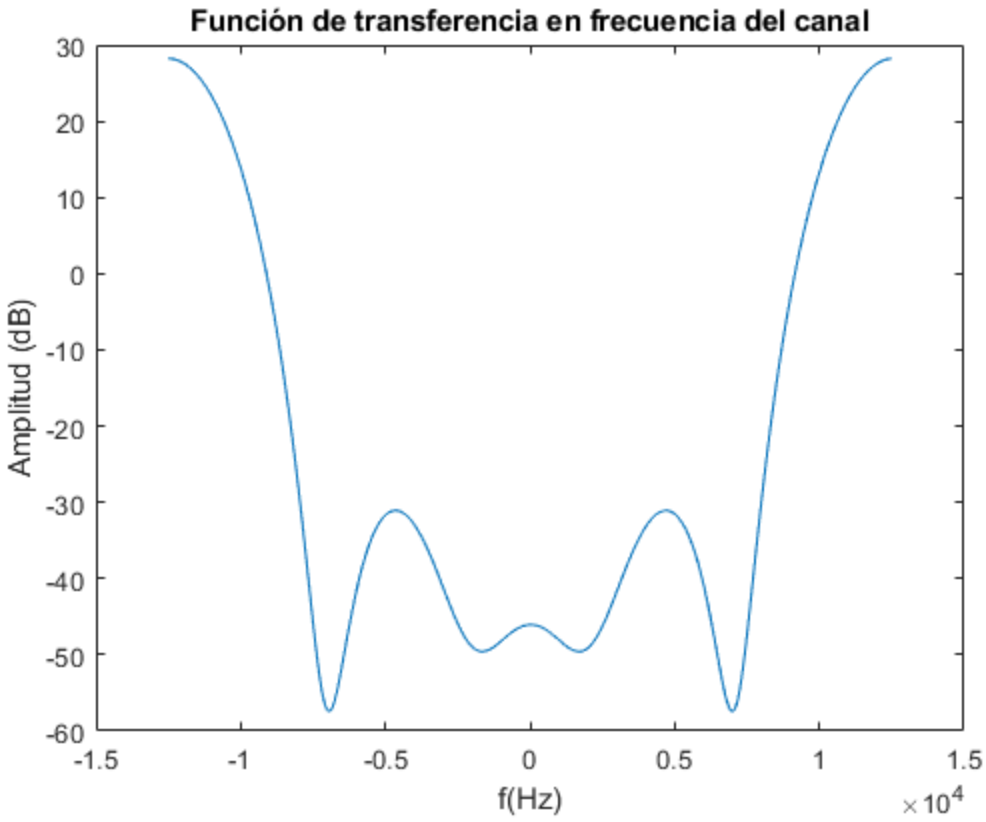
Apartado 2.1: Señal inyectada

$y(t)=x(t)*h(t)$

$y[n]=x[n]*h[n]=-0.1x[n]+0.3x[n-1]-0.5x[n-2]+0.7x[n-3]-0.9x[n-4]+0.7x[n-5]-0.5x[n-6]+0.3x[n-7]-0.1x[n-8]$

Apartado 2.2: Representación gráfica del canal

```
Fs    =  25000;  % Frecuencia de muestreo
NFFT = 512;
h=[-0.1,0.3,-0.5,0.7,-0.9,0.7,-0.5,0.3,-0.1];
f = linspace(-Fs/2, Fs/2, NFFT);
H = fft(h,NFFT);
figure;
plot(f, 20*log(abs(fftshift(H))));
title('Función de transferencia en frecuencia del canal')
xlabel('f(Hz)')
ylabel('Amplitud (dB)')
```

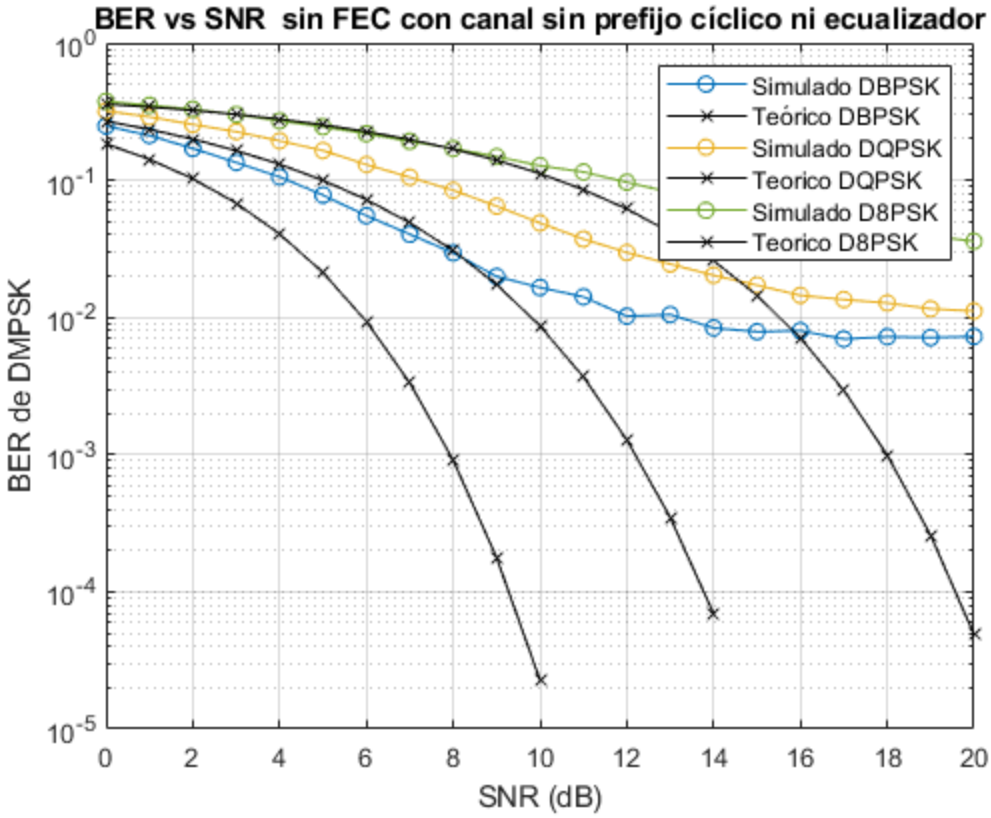


El canal, como se puede apreciar actua como un filtro que atenua las bandas laterales y amplifica las bandas de los extremos.

Apartado 2.3: Curvas BER vs SNR teóricas y simuladas, añadiendo el efecto del canal sin prefijo cíclico ni ecualizador.

```
NFFT  = 512;  % Tamaño de la FFT
Fs    =  250000;  % Frecuencia de muestreo
df    =  Fs/NFFT ;  % Separación entre portadoras
Nf    = 96;  % Numero de portadoras con datos (+1 por el piloto)
N_tramas = 20;
Nofdm = 63; % Número de símbolos OFDM por trama
SNR_vector = 0:20;
for i=1:N_tramas
    BER_DBPSK=zeros(N_tramas,length(SNR_vector));
    BER_DQPSK=zeros(N_tramas,length(SNR_vector));
    BER_D8PSK=zeros(N_tramas,length(SNR_vector));

    [txbitsDBPSK,xDBPSK] = ModulacionOFDMScrambler(2, Nf, NFFT,
 Nofdm);
    [txbitsDQPSK,xDQPSK] = ModulacionOFDMScrambler(4, Nf, NFFT,
 Nofdm);
```

```
    [txbitsD8PSK,xD8PSK] = ModulacionOFDMScrambler(8, Nf, NFFT,
 Nofdm);
    % Se añase ruido:
     BERDBPSK=CalcularErrorRuidoCanal(xDBPSK, Nf, NFFT, 2 ,Nofdm,
 txbitsDBPSK, SNR_vector, h);
     BERDQPSK=CalcularErrorRuidoCanal(xDQPSK, Nf, NFFT, 4 ,Nofdm,
 txbitsDQPSK, SNR_vector, h);
     BERD8PSK=CalcularErrorRuidoCanal(xD8PSK, Nf, NFFT, 8 ,Nofdm,
 txbitsD8PSK, SNR_vector, h);
     BERDBPSKRuido(i,:) = BERDBPSK;
     BERDQPSKRuido(i,:) = BERDQPSK;
     BERD8PSKRuido(i,:) = BERD8PSK;
end
BERDBPSKRuido_avg = zeros(1,length(SNR_vector));
BERDQPSKRuido_avg = zeros(1,length(SNR_vector));
BERD8PSKRuido_avg = zeros(1,length(SNR_vector));

for i=1:N_tramas
      BERDBPSKRuido_avg = BERDBPSKRuido_avg + BERDBPSKRuido(i,:);
      BERDQPSKRuido_avg = BERDQPSKRuido_avg + BERDQPSKRuido(i,:);
      BERD8PSKRuido_avg = BERD8PSKRuido_avg + BERD8PSKRuido(i,:);
end
BERDBPSKRuido_avg = BERDBPSKRuido_avg./N_tramas;
BERDQPSKRuido_avg = BERDQPSKRuido_avg./N_tramas;
BERD8PSKRuido_avg = BERD8PSKRuido_avg./N_tramas;

theoryBerDBPSK = DBPSK_BER(SNR_vector);
theoryBerDQPSK = DQPSK_BER(SNR_vector);
theoryBerD8PSK = D8PSK_BER(SNR_vector);

theoryBerDBPSK(find(theoryBerDBPSK<1e-5)) = NaN;
theoryBerDQPSK(find(theoryBerDQPSK<1e-5)) = NaN;
theoryBerD8PSK(find(theoryBerD8PSK<1e-5)) = NaN;

figure

semilogy(SNR_vector, BERDBPSKRuido_avg, '-o');  hold on
semilogy(SNR_vector, theoryBerDBPSK,     '-kx');
semilogy(SNR_vector, BERDQPSKRuido_avg, '-o');
semilogy(SNR_vector, theoryBerDQPSK,     '-kx');
semilogy(SNR_vector, BERD8PSKRuido_avg, '-o');
semilogy(SNR_vector, theoryBerD8PSK,     '-kx');

legend('Simulado DBPSK','Teórico DBPSK', 'Simulado DQPSK', 'Teorico
 DQPSK', 'Simulado D8PSK', 'Teorico D8PSK')
xlabel('SNR (dB)');  ylabel('BER de DMPSK')
grid on
title('Resultados del sistema OFDM de DBPSK DQPSK y D8PSK sin FEC con
 canal sin prefijo cíclico')
title('BER vs SNR  sin FEC con canal sin prefijo cíclico ni
 ecualizador')
```



Como se podía esperar, cuando se introducen interferencias y retardos en el canal, los errores aumentan. Es curioso que a partir de un umbral de SNR, por mucho que se aumente el porcentaje de errores, se queda estancado ahí.

Dependiendo del tipo de modulación, el umbral varía, para DQPSK, el umbral se encuentra en aproximadamente 19 dB y el BER se estanca en 0.03745 y no se consigue una BER menor a 0.9% a partir de ese umbral. Para DQPSK, ocurre lo mismo, se empieza a estancar el BER en 19 dB con un BER de 0.01108 aproximadamente y no se consigue una BER menor a 2 % a partir de dicho umbral. Para D8PSK, ocurre lo mismo, se empieza a estancar el BER en 19 dB con un BER de 0.007068 aproximadamente y no se consigue una BER menor a 5 % a partir de dicho umbral.

E SNR coincide en todas las modulaciones, el error obtenido varia debido a la separación entre símbolos de cada una.

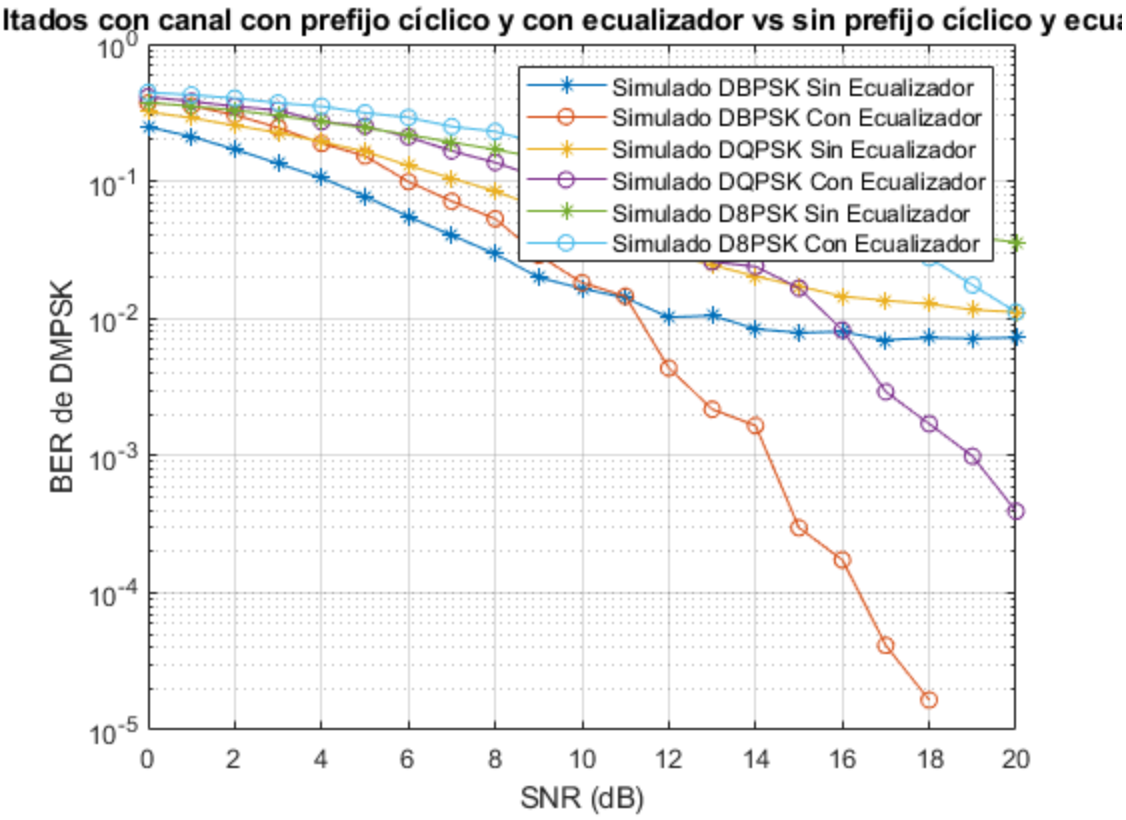
Apartado 2.4: Curvas BER vs SNR Teórica con canal, prefijo cíclico y ecualizador

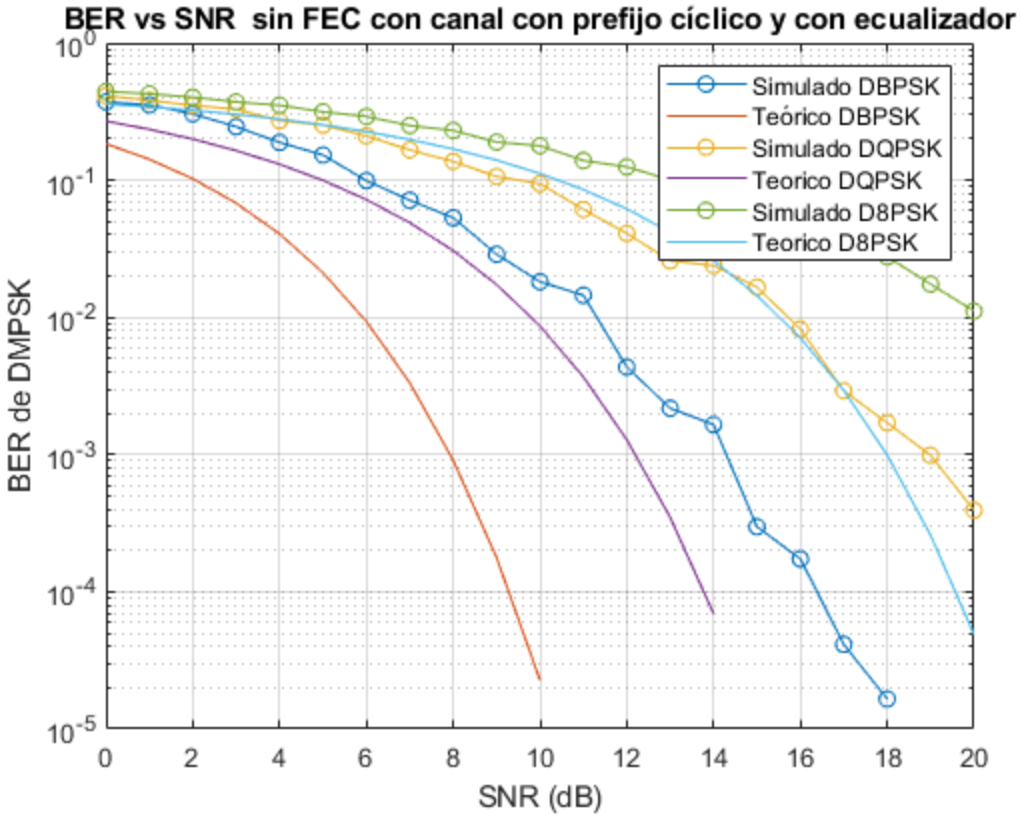
En este introduciremos el prefijo cíclicoy, en el receptor, depués del demodulador un ecualizador. Se empleará como piloto el primer símbolo OFDM, se asumirá que el receptor conoce las amplitudes complejas de las portadoras del primer símbolo OFDM antes de inyectar la señal en línea. Con estos añadidos, se calculará y se representará las nuevas curvas de

```
for i=1:N_tramas
    BER_DBPSK=zeros(N_tramas,length(SNR_vector));
    BER_DQPSK=zeros(N_tramas,length(SNR_vector));
```

```
    BER_D8PSK=zeros(N_tramas,length(SNR_vector));

    [txbitsDBPSK,xDBPSK, pilotoDBPSK] = ModulacionOFDMEcualizacion(2,
 Nf, NFFT, Nofdm);
    [txbitsDQPSK,xDQPSK, pilotoDQPSK] = ModulacionOFDMEcualizacion(4,
 Nf, NFFT, Nofdm);
    [txbitsD8PSK,xD8PSK, pilotoD8PSK] = ModulacionOFDMEcualizacion(8,
 Nf, NFFT, Nofdm);
    % Demodulacion con Ecualizador Zero Forcing
     BERDBPSK_Ecualizador=CalcularErrorRuidoCanalEcualizado(xDBPSK,
 Nf, NFFT, 2 ,Nofdm, txbitsDBPSK, SNR_vector, h, pilotoDBPSK);
     BERDQPSK_Ecualizador=CalcularErrorRuidoCanalEcualizado(xDQPSK,
 Nf, NFFT, 4 ,Nofdm, txbitsDQPSK, SNR_vector, h, pilotoDQPSK);
     BERD8PSK_Ecualizador=CalcularErrorRuidoCanalEcualizado(xD8PSK,
 Nf, NFFT, 8 ,Nofdm, txbitsD8PSK, SNR_vector, h, pilotoD8PSK);
     BERDBPSKRuido_Ecualizador(i,:) = BERDBPSK_Ecualizador;
     BERDQPSKRuido_Ecualizador(i,:) = BERDQPSK_Ecualizador;
     BERD8PSKRuido_Ecualizador(i,:) = BERD8PSK_Ecualizador;
end
BERDBPSKRuido_avg_Ecualizador = zeros(1,length(SNR_vector));
BERDQPSKRuido_avg_Ecualizador = zeros(1,length(SNR_vector));
BERD8PSKRuido_avg_Ecualizador = zeros(1,length(SNR_vector));

for i=1:N_tramas
      BERDBPSKRuido_avg_Ecualizador = BERDBPSKRuido_avg_Ecualizador +
 BERDBPSKRuido_Ecualizador(i,:);
      BERDQPSKRuido_avg_Ecualizador = BERDQPSKRuido_avg_Ecualizador +
 BERDQPSKRuido_Ecualizador(i,:);
      BERD8PSKRuido_avg_Ecualizador = BERD8PSKRuido_avg_Ecualizador +
 BERD8PSKRuido_Ecualizador(i,:);
end
BERDBPSKRuido_avg_Ecualizador = BERDBPSKRuido_avg_Ecualizador./
N_tramas;
BERDQPSKRuido_avg_Ecualizador = BERDQPSKRuido_avg_Ecualizador./
N_tramas;
BERD8PSKRuido_avg_Ecualizador = BERD8PSKRuido_avg_Ecualizador./
N_tramas;
figure

semilogy(SNR_vector, BERDBPSKRuido_avg, '-*');  hold on
semilogy(SNR_vector, BERDBPSKRuido_avg_Ecualizador, '-o');
semilogy(SNR_vector, BERDQPSKRuido_avg, '-*');
semilogy(SNR_vector, BERDQPSKRuido_avg_Ecualizador, '-o');
semilogy(SNR_vector, BERD8PSKRuido_avg, '-*');
semilogy(SNR_vector, BERD8PSKRuido_avg_Ecualizador, '-o');

legend('Simulado DBPSK Sin Ecualizador','Simulado DBPSK Con
 Ecualizador', 'Simulado DQPSK Sin Ecualizador', 'Simulado DQPSK Con
 Ecualizador', 'Simulado D8PSK Sin Ecualizador', 'Simulado D8PSK Con
 Ecualizador')
xlabel('SNR (dB)');  ylabel('BER de DMPSK')
grid on
title(' Resultados con canal con prefijo cíclico y con ecualizador vs
 sin prefijo cíclico y ecualización')
```

```
figure

semilogy(SNR_vector, BERDBPSKRuido_avg_Ecualizador, '-o');  hold on
semilogy(SNR_vector, theoryBerDBPSK);
semilogy(SNR_vector, BERDQPSKRuido_avg_Ecualizador, '-o');
semilogy(SNR_vector, theoryBerDQPSK);
semilogy(SNR_vector, BERD8PSKRuido_avg_Ecualizador, '-o');
semilogy(SNR_vector, theoryBerD8PSK);

legend('Simulado DBPSK','Teórico DBPSK', 'Simulado DQPSK', 'Teorico
 DQPSK', 'Simulado D8PSK', 'Teorico D8PSK')
xlabel('SNR (dB)');  ylabel('BER de DMPSK')
grid on
title('BER vs SNR  sin FEC con canal con prefijo cíclico y con
 ecualizador')
```





Se muestra el resultado de introducir el ecualizador en recepción. Se muestran dos figuras, una figura muestra las curvas de error BER en función de la SNR de las señales recibidas con y sin ecualizador. La siguiente figura, muestra las curvas creadas con ecualizador junto con las teóricas sin distorsión de canal. Se aprecia en las figuras, como el error de fondo que limitaba todas las señales con un BER muy grande, el ecualizador lo soluciona. Se observa que tal y como se esperaba la curva teórica es parecida a la de con ecualizador, pero las curvas se desplezan un poco a la derecha. Esto se debe a que el ecualizador es Zero-Forcing y como vimos hace que se aumente el SNR y por tanto los errores. Para DBPSK, encontramos que para un BER de 10^-4 hay una diferencia de SNR de 7 dB con respecto a la teórica. Para DQPSK, con un BER de 10^-4 hay una direncia aproximada de unos 6 dB. Por último, para un BER de 10^-4, tenemos una diferenca de SNR de casi 8 dB.

```
clear all;
```

Apartado 3: Implementación de todos los modos de comunicación de PRIME con FEC

Partiendo del sistema ya definido con su código para la obtención de curvas de BER frente a SNR, incluyendo canal real, incluir las técnicas de corrección de errores, FEC, definidas en el estándar. Para implementar FEC es necesario modificar los valores de algunos parámetros de simulación. Se debe justificar la modificación, teniendo en cuenta que las tramas payload deben construirse de acuerdo al estándar PRIME, que se establece una diferencia entre bits de información, o antes de codificar, y bits codificados, la existencia de bits de vaciado (flushing), etc.

Apartado 3.1: Modificación de parámetros.

Cuando aplicamos FEC, usaremos el codificador standard usado por PRIME según se especifica en el documento standard. Se duplican los bits de entrada y del mismo modo, los paquetes necesarios para enviar toda la cadena. El codificador convolucional que usa PRIME, usa 7 registros que deben de estar a cero en el inicio, por lo que para para la entrada de cada paquete se deben reiniciar a cero, se añaden 8 bits de valor 0 al inicio de la cadena, estos bits son los bits de flushing.

Para poder computar la variación de paquetes transmitidos con FEC, se han creado dos funciones para calcularlo, se mostrarán los paquetes que se necesitan para enviar una cadena de 80000 bits en todos los formatos de modulación aceptados por prime, en los dos modos, con y sin FEC.

Se puede comprobar, que tal y como esperabamos, por el aumento debido a tener que poner los registros de desplazamiento a cero, que no en todos los casos con FEC es el doble exactamente de paquetes a sin FEC. Esto también es porque el número de bits no es exactamente redondo para rellenar todos paquetes, por lo que el último paquete se rellena con ceros al final para que sea exacto y cuadre.

```
clear; close all, format compact
Nf    = 96;  % Numero de portadoras con datos (+1 por el piloto)
Nofdm = 63; % Número de símbolos OFDM por trama
numPaquetesDBPSKFEC = calcularNumeroPaquetesFEC(80000, 2, Nofdm, Nf)
numPaquetesDBPSKnoFEC = calcularNumeroPaquetesnoFEC(80000, 2, Nofdm,
 Nf)

numPaquetesDQPSKFEC = calcularNumeroPaquetesFEC(80000, 4, Nofdm, Nf)
numPaquetesDQPSKnoFEC = calcularNumeroPaquetesnoFEC(80000, 4, Nofdm,
 Nf)

numPaquetesD8PSKFEC = calcularNumeroPaquetesFEC(80000, 8, Nofdm, Nf)
numPaquetesD8PSKnoFEC = calcularNumeroPaquetesnoFEC(80000, 8, Nofdm,
 Nf)

numPaquetesDBPSKFEC =
    27
numPaquetesDBPSKnoFEC =
    14
numPaquetesDQPSKFEC =
    14
numPaquetesDQPSKnoFEC =
     7
numPaquetesD8PSKFEC =
     9
numPaquetesD8PSKnoFEC =
     5
```

Apartado 3.2.1: Cadena con entrelazado en ausencia de ruido.

```
NFFT  = 512;  % Tamaño de la FFT
Fs    =  250000;  % Frecuencia de muestreo
df    =  Fs/NFFT ;  % Separación entre portadoras
```

```
Nf    = 96;  % Numero de portadoras con datos (+1 por el piloto)
N_tramas = 20;
Nofdm = 63; % Número de símbolos OFDM por trama
SNR_vector = 0:20;

BERDBPSK_Entrelazado_Total = [];
BERDQPSK_Entrelazado_Total = [];
BERD8PSK_Entrelazado_Total = [];
BERDBPSK_FEC_Total = [];
BERDQPSK_FEC_Total = [];
BERD8PSK_FEC_Total = [];

BER_DBPSK_conFEC=zeros(N_tramas,length(SNR_vector));
BER_DQPSK_conFEC=zeros(N_tramas,length(SNR_vector));
BER_D8PSK_conFEC=zeros(N_tramas,length(SNR_vector));

BER_DBPSK_sinFEC=zeros(N_tramas,length(SNR_vector));
BER_DQPSK_sinFEC=zeros(N_tramas,length(SNR_vector));
BER_D8PSK_sinFEC=zeros(N_tramas,length(SNR_vector));
for iter=1:N_tramas
    [txbitsDBPSK,xDBPSK] = ModulacionInterleaver(2, Nf, NFFT, Nofdm);
    [txbitsDQPSK,xDQPSK] = ModulacionInterleaver(4, Nf, NFFT, Nofdm);
    [txbitsD8PSK,xD8PSK] = ModulacionInterleaver(8, Nf, NFFT, Nofdm);
    % Demodulacion con Interleaver
    h=[-0.1,0.3,-0.5,0.7,-0.9,0.7,-0.5,0.3,-0.1];
    BERDBPSK_Entrelazado=CalcularErrorInterleaver(xDBPSK, Nf, NFFT,
 2 ,Nofdm, txbitsDBPSK);
    BERDQPSK_Entrelazado=CalcularErrorInterleaver(xDQPSK, Nf, NFFT,
 4 ,Nofdm, txbitsDQPSK);
    BERD8PSK_Entrelazado=CalcularErrorInterleaver(xD8PSK, Nf, NFFT,
 8 ,Nofdm, txbitsD8PSK);
    BERDBPSK_Entrelazado_Total = [BERDBPSK_Entrelazado_Total
 BERDBPSK_Entrelazado];
    BERDQPSK_Entrelazado_Total = [BERDQPSK_Entrelazado_Total
 BERDQPSK_Entrelazado];
    BERD8PSK_Entrelazado_Total = [BERD8PSK_Entrelazado_Total
 BERD8PSK_Entrelazado];
% No hay errores al añadir el entrelazado
end
BERDBPSK_Entrelazado_Total = sum(BERDBPSK_Entrelazado_Total)
BERDQPSK_Entrelazado_Total = sum(BERDQPSK_Entrelazado_Total)
BERD8PSK_Entrelazado_Total = sum(BERD8PSK_Entrelazado_Total)

BERDBPSK_Entrelazado_Total =
     0
BERDQPSK_Entrelazado_Total =
     0
BERD8PSK_Entrelazado_Total =
     0
```

Tal y como se esperaba, en ausencia de ruido y unicamente añadiendo el entrelazado y desentrelazado, no se producen errores y por tanto, al igual que el apartado 1, podemos decir que se ha añadido correctamente los bloques de entrelazado y desentralazado.

Apartado 3.2.2: Cadena con entrelazado y FEC en ausencia de ruido.

```
%Añadimos la codificación
h=[-0.1,0.3,-0.5,0.7,-0.9,0.7,-0.5,0.3,-0.1];
NFFT  = 512;  % Tamaño de la FFT
Fs    =  250000;  % Frecuencia de muestreo
df    =  Fs/NFFT ;  % Separación entre portadoras
Nf    = 96;  % Numero de portadoras con datos (+1 por el piloto)
N_tramas = 20;
Nofdm = 63; % Número de símbolos OFDM por trama
SNR_vector = 0:20;
for i=1:N_tramas
    lg = 7;
    enrejado = poly2trellis([lg],[171,133]);
    [txbitsDBPSK,xDBPSK] = ModulacionConvolutionalEncoder(2, Nf, NFFT,
 Nofdm, enrejado);
    [txbitsDQPSK,xDQPSK] = ModulacionConvolutionalEncoder(4, Nf, NFFT,
 Nofdm, enrejado);
    [txbitsD8PSK,xD8PSK] = ModulacionConvolutionalEncoder(8, Nf, NFFT,
 Nofdm, enrejado);
    % Demodulacion con Convolutional Encoder
    BERDBPSK_FEC=CalcularErrorConvolutionalEncoder(xDBPSK, Nf, NFFT,
 2 ,Nofdm, txbitsDBPSK, enrejado);
    BERDQPSK_FEC=CalcularErrorConvolutionalEncoder(xDQPSK, Nf, NFFT,
 4 ,Nofdm, txbitsDQPSK, enrejado);
    BERD8PSK_FEC=CalcularErrorConvolutionalEncoder(xD8PSK, Nf, NFFT,
 8 ,Nofdm, txbitsD8PSK, enrejado);
    BERDBPSK_FEC_Total = [BERDBPSK_FEC_Total BERDBPSK_FEC];
    BERDQPSK_FEC_Total = [BERDQPSK_FEC_Total BERDQPSK_FEC];
    BERD8PSK_FEC_Total = [BERD8PSK_FEC_Total BERD8PSK_FEC];
end
BERDBPSK_FEC_Total = sum(BERDBPSK_FEC_Total)
BERDQPSK_FEC_Total = sum(BERDQPSK_FEC_Total)
BERD8PSK_FEC_Total = sum(BERD8PSK_FEC_Total)

BERDBPSK_FEC_Total =
     0
BERDQPSK_FEC_Total =
     0
BERD8PSK_FEC_Total =
     0
```

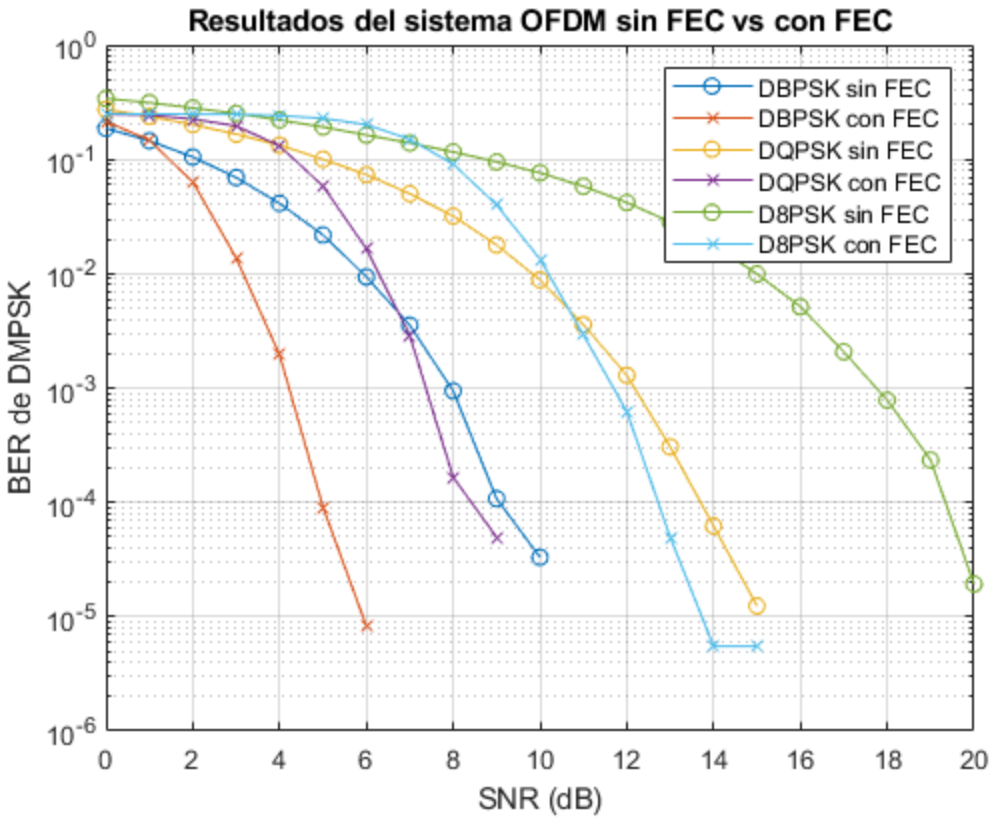
Tal y como se esperaba, en ausencia de ruido y unicamente añadiendo la codificación y decodificación, no se producen errores y por tanto, al igual que el apartado 1, podemos decir que se ha añadido correctamente los bloques de entrelazado y desentralazado y de convolutional encoder.

Apartado 3.2.3: Curvas BER vs SNR

Primero representaremos las curvas de BER vs SNR sin FEC y sin canal frente a las propias curvas con FEC sin canal:

```
for i=1:N_tramas
    BER_DBPSK=zeros(N_tramas,length(SNR_vector));
    BER_DQPSK=zeros(N_tramas,length(SNR_vector));
    BER_D8PSK=zeros(N_tramas,length(SNR_vector));
    BER_DBPSK_FEC=zeros(N_tramas,length(SNR_vector));
    BER_DQPSK_FEC=zeros(N_tramas,length(SNR_vector));
    BER_D8PSK_FEC=zeros(N_tramas,length(SNR_vector));

    [txbitsDBPSK,xDBPSK] = ModulacionOFDMScrambler(2, Nf, NFFT,
 Nofdm);
    [txbitsDQPSK,xDQPSK] = ModulacionOFDMScrambler(4, Nf, NFFT,
 Nofdm);
    [txbitsD8PSK,xD8PSK] = ModulacionOFDMScrambler(8, Nf, NFFT,
 Nofdm);

    % Se añase ruido:
    BERDBPSK=CalcularErrorRuido(xDBPSK, Nf, NFFT, 2 ,Nofdm,
 txbitsDBPSK, SNR_vector);
    BERDQPSK=CalcularErrorRuido(xDQPSK, Nf, NFFT, 4 ,Nofdm,
 txbitsDQPSK, SNR_vector);
    BERD8PSK=CalcularErrorRuido(xD8PSK, Nf, NFFT, 8 ,Nofdm,
 txbitsD8PSK, SNR_vector);
    BERDBPSKRuido(i,:) = BERDBPSK;
    BERDQPSKRuido(i,:) = BERDQPSK;
    BERD8PSKRuido(i,:) = BERD8PSK;

    % Ahora con FEC:
    lg = 7;
    enrejado = poly2trellis([lg],[171,133]);
    [txbitsDBPSK_FEC,xDBPSK_FEC] = ModulacionConvolutionalEncoder(2,
 Nf, NFFT, Nofdm, enrejado);
    [txbitsDQPSK_FEC,xDQPSK_FEC] = ModulacionConvolutionalEncoder(4,
 Nf, NFFT, Nofdm, enrejado);
    [txbitsD8PSK_FEC,xD8PSK_FEC] = ModulacionConvolutionalEncoder(8,
 Nf, NFFT, Nofdm, enrejado);
    BERDBPSK_FEC=CalcularErrorConvolutionalEncoderRuido(xDBPSK_FEC,
 Nf, NFFT, 2 ,Nofdm, txbitsDBPSK_FEC, enrejado, SNR_vector);
    BERDQPSK_FEC=CalcularErrorConvolutionalEncoderRuido(xDQPSK_FEC,
 Nf, NFFT, 4 ,Nofdm, txbitsDQPSK_FEC, enrejado, SNR_vector);
    BERD8PSK_FEC=CalcularErrorConvolutionalEncoderRuido(xD8PSK_FEC,
 Nf, NFFT, 8 ,Nofdm, txbitsD8PSK_FEC, enrejado, SNR_vector);
    BERDBPSKRuido_FEC(i,:) = BERDBPSK_FEC;
    BERDQPSKRuido_FEC(i,:) = BERDQPSK_FEC;
    BERD8PSKRuido_FEC(i,:) = BERD8PSK_FEC;
end
BERDBPSKRuido_avg = zeros(1,length(SNR_vector));
BERDQPSKRuido_avg = zeros(1,length(SNR_vector));
BERD8PSKRuido_avg = zeros(1,length(SNR_vector));
BERDBPSKRuido_avg_FEC = zeros(1,length(SNR_vector));
BERDQPSKRuido_avg_FEC = zeros(1,length(SNR_vector));
BERD8PSKRuido_avg_FEC = zeros(1,length(SNR_vector));
for i=1:N_tramas
      BERDBPSKRuido_avg = BERDBPSKRuido_avg + BERDBPSKRuido(i,:);
      BERDQPSKRuido_avg = BERDQPSKRuido_avg + BERDQPSKRuido(i,:);
```

```
        BERD8PSKRuido_avg = BERD8PSKRuido_avg + BERD8PSKRuido(i,:);
        BERDBPSKRuido_avg_FEC = BERDBPSKRuido_avg_FEC +
 BERDBPSKRuido_FEC(i,:);
        BERDQPSKRuido_avg_FEC = BERDQPSKRuido_avg_FEC +
 BERDQPSKRuido_FEC(i,:);
        BERD8PSKRuido_avg_FEC = BERD8PSKRuido_avg_FEC +
 BERD8PSKRuido_FEC(i,:);
end
BERDBPSKRuido_avg = BERDBPSKRuido_avg./N_tramas;
BERDQPSKRuido_avg = BERDQPSKRuido_avg./N_tramas;
BERD8PSKRuido_avg = BERD8PSKRuido_avg./N_tramas;

BERDBPSKRuido_avg_FEC = BERDBPSKRuido_avg_FEC./N_tramas;
BERDQPSKRuido_avg_FEC = BERDQPSKRuido_avg_FEC./N_tramas;
BERD8PSKRuido_avg_FEC = BERD8PSKRuido_avg_FEC./N_tramas;
figure
semilogy(SNR_vector, BERDBPSKRuido_avg, '-o');  hold on
semilogy(SNR_vector, BERDBPSKRuido_avg_FEC,     '-x');
semilogy(SNR_vector, BERDQPSKRuido_avg, '-o');
semilogy(SNR_vector, BERDQPSKRuido_avg_FEC,     '-x');
semilogy(SNR_vector, BERD8PSKRuido_avg, '-o');
semilogy(SNR_vector, BERD8PSKRuido_avg_FEC,     '-x');

legend('DBPSK sin FEC','DBPSK con FEC', 'DQPSK sin FEC', 'DQPSK con
 FEC', 'D8PSK sin FEC', 'D8PSK con FEC')
xlabel('SNR (dB)');  ylabel('BER de DMPSK')
grid on
title('Resultados del sistema OFDM sin FEC vs con FEC')
```



Se muestra una representación de BER vs SNR con y sin FEC cuando no se añade el canal, ni prefijo cíclico, ni ecualización. Como se puede apreciar, el rendimiento es peor cuando no se introduce FEC que cuando se introduce FEC, en líneas generales. Al principio (con SNR=0 o cercano), sin embargo, se observa que la modulación con FEC actua peor que la modulación sin FEC, esto se puede deber a que la codificación de un bit con FEC siempre influenciada por las anteriores codificaciones, por lo que el error se arrastra. Esto no ocurre en la codificación sin FEC. El umbral para cada modulación varía para cada modulación: a partir de 2dB para DBPSK, a partir de 4 dB para DQPSK y a partir de 8 dB para D8PSK.

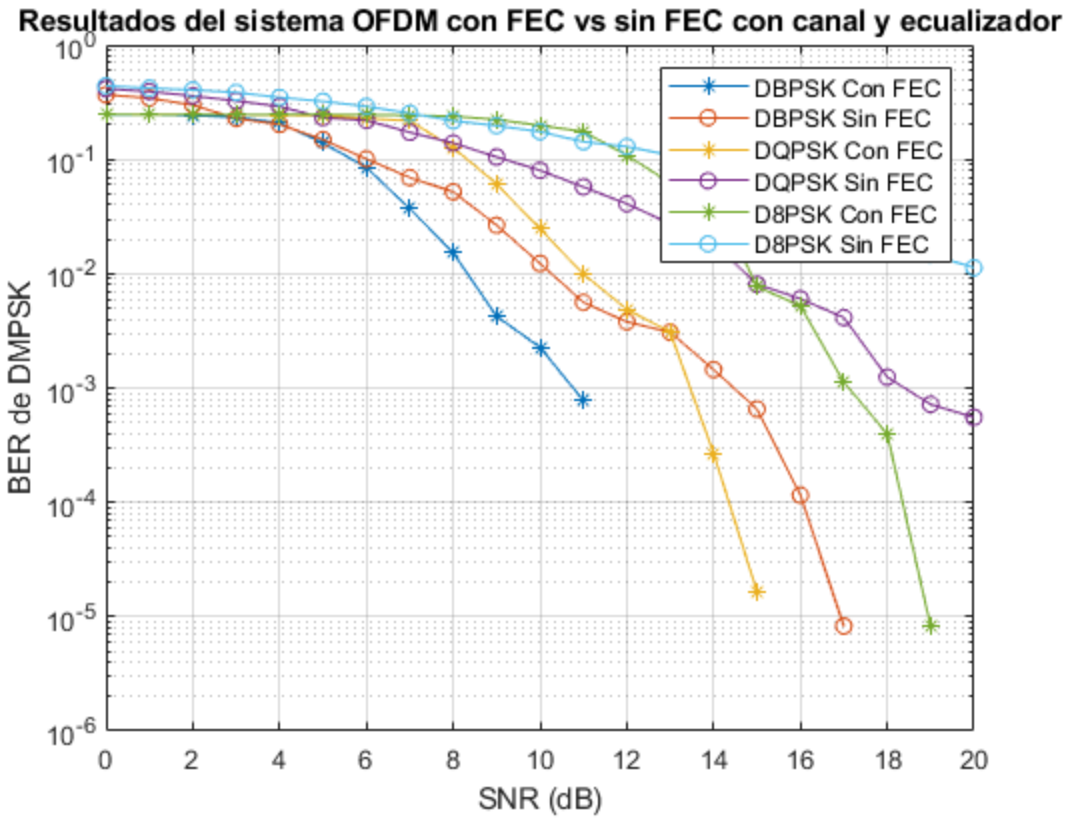
Podemos decir que con FEC se mandan más bits para un número fijo de bits de información, estos bits permiten corregir errores al decodificar, por lo que necesita menor SNR y la potencia para transmitir información va a ser mayor. Por otro lado, se reduce la velocidad de transmisión y se aumenta considerablemente la complejidad.

Canal con ecualización:

```
h=[-0.1,0.3,-0.5,0.7,-0.9,0.7,-0.5,0.3,-0.1];
for i=1:N_tramas
    BER_DBPSK=zeros(N_tramas,length(SNR_vector));
    BER_DQPSK=zeros(N_tramas,length(SNR_vector));
    BER_D8PSK=zeros(N_tramas,length(SNR_vector));

    [txbitsDBPSK,xDBPSK, pilotoDBPSK] = ModulacionOFDMEcualizacion(2,
 Nf, NFFT, Nofdm);
    [txbitsDQPSK,xDQPSK, pilotoDQPSK] = ModulacionOFDMEcualizacion(4,
 Nf, NFFT, Nofdm);
```

```
    [txbitsD8PSK,xD8PSK, pilotoD8PSK] = ModulacionOFDMEcualizacion(8,
 Nf, NFFT, Nofdm);
    % Demodulacion con Ecualizador Zero Forcing
     BERDBPSK_Ecualizador=CalcularErrorRuidoCanalEcualizado(xDBPSK,
 Nf, NFFT, 2 ,Nofdm, txbitsDBPSK, SNR_vector, h, pilotoDBPSK);
     BERDQPSK_Ecualizador=CalcularErrorRuidoCanalEcualizado(xDQPSK,
 Nf, NFFT, 4 ,Nofdm, txbitsDQPSK, SNR_vector, h, pilotoDQPSK);
     BERD8PSK_Ecualizador=CalcularErrorRuidoCanalEcualizado(xD8PSK,
 Nf, NFFT, 8 ,Nofdm, txbitsD8PSK, SNR_vector, h, pilotoD8PSK);
     BERDBPSKRuido_Ecualizador(i,:) = BERDBPSK_Ecualizador;
     BERDQPSKRuido_Ecualizador(i,:) = BERDQPSK_Ecualizador;
     BERD8PSKRuido_Ecualizador(i,:) = BERD8PSK_Ecualizador;
end
BERDBPSKRuido_avg_Ecualizador = zeros(1,length(SNR_vector));
BERDQPSKRuido_avg_Ecualizador = zeros(1,length(SNR_vector));
BERD8PSKRuido_avg_Ecualizador = zeros(1,length(SNR_vector));

for i=1:N_tramas
      BERDBPSKRuido_avg_Ecualizador = BERDBPSKRuido_avg_Ecualizador +
 BERDBPSKRuido_Ecualizador(i,:);
     BERDQPSKRuido_avg_Ecualizador = BERDQPSKRuido_avg_Ecualizador +
 BERDQPSKRuido_Ecualizador(i,:);
     BERD8PSKRuido_avg_Ecualizador = BERD8PSKRuido_avg_Ecualizador +
 BERD8PSKRuido_Ecualizador(i,:);
end
BERDBPSKRuido_avg_Ecualizador = BERDBPSKRuido_avg_Ecualizador./
N_tramas;
BERDQPSKRuido_avg_Ecualizador = BERDQPSKRuido_avg_Ecualizador./
N_tramas;
BERD8PSKRuido_avg_Ecualizador = BERD8PSKRuido_avg_Ecualizador./
N_tramas;
for i=1:N_tramas
    BER_DBPSK_FEC=zeros(N_tramas,length(SNR_vector));
    BER_DQPSK_FEC=zeros(N_tramas,length(SNR_vector));
    BER_D8PSK_FEC=zeros(N_tramas,length(SNR_vector));

    lg = 7;
    enrejado = poly2trellis([lg],[171,133]);
    [txbitsDBPSK_FEC,xDBPSK_FEC,piloto_ecualizadorDBPSK] =
 ModulacionConvolutionalEncoderPC(2, Nf, NFFT, Nofdm, enrejado);
    [txbitsDQPSK_FEC,xDQPSK_FEC,piloto_ecualizadorDQPSK] =
 ModulacionConvolutionalEncoderPC(4, Nf, NFFT, Nofdm, enrejado);
    [txbitsD8PSK_FEC,xD8PSK_FEC,piloto_ecualizadorD8PSK] =
 ModulacionConvolutionalEncoderPC(8, Nf, NFFT, Nofdm, enrejado);

 BERDBPSK_FEC=CalcularErrorConvolutionalEncoderRuidoCanal(xDBPSK_FEC,
 Nf, NFFT, 2 ,Nofdm, txbitsDBPSK_FEC, enrejado, SNR_vector, h,
 piloto_ecualizadorDBPSK);

 BERDQPSK_FEC=CalcularErrorConvolutionalEncoderRuidoCanal(xDQPSK_FEC,
 Nf, NFFT, 4 ,Nofdm, txbitsDQPSK_FEC, enrejado, SNR_vector, h,
 piloto_ecualizadorDQPSK);

 BERD8PSK_FEC=CalcularErrorConvolutionalEncoderRuidoCanal(xD8PSK_FEC,
```

```
 Nf, NFFT, 8 ,Nofdm, txbitsD8PSK_FEC, enrejado, SNR_vector, h,
 piloto_ecualizadorD8PSK);
    BERDBPSKRuido_FEC(i,:) = BERDBPSK_FEC;
    BERDQPSKRuido_FEC(i,:) = BERDQPSK_FEC;
    BERD8PSKRuido_FEC(i,:) = BERD8PSK_FEC;
end
BERDBPSKRuido_avg_FEC = zeros(1,length(SNR_vector));
BERDQPSKRuido_avg_FEC = zeros(1,length(SNR_vector));
BERD8PSKRuido_avg_FEC = zeros(1,length(SNR_vector));
for i=1:N_tramas
      BERDBPSKRuido_avg_FEC = BERDBPSKRuido_avg_FEC +
 BERDBPSKRuido_FEC(i,:);
      BERDQPSKRuido_avg_FEC = BERDQPSKRuido_avg_FEC +
 BERDQPSKRuido_FEC(i,:);
      BERD8PSKRuido_avg_FEC = BERD8PSKRuido_avg_FEC +
 BERD8PSKRuido_FEC(i,:);
end
BERDBPSKRuido_avg_FEC = BERDBPSKRuido_avg_FEC./N_tramas;
BERDQPSKRuido_avg_FEC = BERDQPSKRuido_avg_FEC./N_tramas;
BERD8PSKRuido_avg_FEC = BERD8PSKRuido_avg_FEC./N_tramas;

figure

semilogy(SNR_vector, BERDBPSKRuido_avg_FEC, '-*');  hold on
semilogy(SNR_vector, BERDBPSKRuido_avg_Ecualizador, '-o');
semilogy(SNR_vector, BERDQPSKRuido_avg_FEC, '-*');
semilogy(SNR_vector, BERDQPSKRuido_avg_Ecualizador, '-o');
semilogy(SNR_vector, BERD8PSKRuido_avg_FEC, '-*');
semilogy(SNR_vector, BERD8PSKRuido_avg_Ecualizador, '-o');

legend('DBPSK Con FEC','DBPSK Sin FEC', 'DQPSK Con FEC', 'DQPSK Sin
 FEC', 'D8PSK Con FEC', 'D8PSK Sin FEC')
xlabel('SNR (dB)');  ylabel('BER de DMPSK')
grid on
title('Resultados del sistema OFDM con FEC vs sin FEC con canal y
 ecualizador')
```



En este caso y como esperábamos, si comparamos las curvas con canal y ecualización con FEC y sin FEC, se puede observar las curvas sin FEC tienen mejor respuesta que las curvas sin FEC, los rendimientos ademas cambian para cada modulación: Para DBPSK para un BER de 10^-4 hay una diferencia de SNR de unos 5 dB, para DQPSK para una BER de 10^-3 hay una diferencia de SNR de unos 6 dB (sin FEC se necesita más SNR para llegar a 10^-4) para D8PSK para una BER de 10^-2, hay una diferencia de 9 dB (sin FEC se necesita mucho más SNR para bajar el BER).

ANEXO: Funciones

En este anexo Se muestran las funciones utilizadas durante la práctica con sus respectivas explicaciones y comentarios.

A1: addPrefijoCiclico.m

```
function [signal] = addPrefijoCiclico(x)
% Función: Función que se utiliza para añadir el prefijo cíclico al
 vector
% entrante. Se coje el final del vector (48 últimas muestras según
 indica
% el standard) y se pone al principio de la señal, finalmente se
 devuelbe
% la señal como un vector.
% Input: x= Vector al que se añadirá el prefijo cíclico.
% Output: signal= Vector con el prefijo cíclico añadido.
```

```
    prefijo_ciclico = x(end-47:end,:);
    signal = [prefijo_ciclico;x];
    signal = signal(:)';
end
```

A2: BPSKMod.m

```
function [output] = BPSKMod(bits)
    for i=1:1:length(bits);
        if bits(i) == 0
            output(i) = 1;
        else
            output(i) = -1;
        end
    end
end
```

A3: CalcularError.m

```
function BER = CalcularError(signal, Nf, NFFT, M ,Nofdm, tx_bits)
% Función: Función que recibe la señal transmitida y la recibida,
% demodulada la recibida, las compará y devuelve el BER de la
 transmisión.
% Input:Signal= La señal modulada que se recibé del receptor. Nf= El
 número de portadoras con datos, viene especificado en
% el standard, el máximo será 96. NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. M= Tipo de modulación
 que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK.  Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 tx_bits= Bits que se transmiten, se usará para
% compararlos con los bits recibidos después de demodular y calcular
 los errores.
% Output: BER= El BER computado en la transmisión, con los errores
 dividido
% entre el número de bits transmitidos.
% Demodulación OFDM
    [Y,pilotos_angulos] = OFDM_Demodulador(signal,NFFT,Nofdm);
% Demodulación DMPSK
    rx_bits = DMPSK_Demod(Y, M, pilotos_angulos);
% Cálculo de los errores comparando los transmitidos con los recibidos
    errores = sum(bitxor(tx_bits(:), rx_bits(:)));
    BER = errores / (Nf*Nofdm*log2(M));
end
```

A4: CalcularErrorConvolutionalEncoder.m

```
function BER = CalcularErrorConvolutionalEncoder(signal, Nf, NFFT,
 M ,Nofdm, tx_bits, enrejado)
% Función: Función que recibe la señal transmitida y la recibida,
% demodulada la recibida, las compará y devuelve el BER de la
 transmisión.
% En este caso al final, se debe añadir una fase de desentralazado,
 dealeatorización y decodificación.
% Input:Signal= La señal modulada que se recibé del receptor. Nf= El
 número de portadoras con datos, viene especificado en
% el standard, el máximo será 96. NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. M= Tipo de modulación
 que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK.  Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 tx_bits= Bits que se transmiten, se usará para
% compararlos con los bits recibidos después de demodular y calcular
 los
% errores. enrejado=  resultado de la función polytrellis que funciona
 como registro
% de desplazamiento.
% Output: BER= El BER computado en la transmisión, con los errores
 dividido
% entre el número de bits transmitidos.
% Demodulación OFDM
    [Y,pilotos_angulos] = OFDM_Demodulador(signal,NFFT,Nofdm);
% Demodulación DMPSK
    rx_bits_aleatorios = DMPSK_Demod(Y, M, pilotos_angulos);
% Deshacer entrelazado con la llamada a la función DeEntrelazado
    rx_bits_aleatorios = DeEntrelazado(rx_bits_aleatorios(:), M,
 Nofdm, Nf);
% Dealeatorización del vector con la llamada a la función Scrambler

    rx_bits = Scrambler(rx_bits_aleatorios(:)');
% Decodificador
    rx_bits = vitdec(rx_bits,enrejado,5*7,'trunc','hard');
% Cálculo de los errores comparando los transmitidos con los recibidos
    errores = sum(bitxor(tx_bits(:), rx_bits'));
    BER = errores / (Nf*Nofdm*log2(M));
end
```

A5: CalcularErrorConvolutionalEncoderRuido.m

```
function BER = CalcularErrorConvolutionalEncoderRuido(signal, Nf,
 NFFT, M ,Nofdm, tx_bits, enrejado, ruido, canal)
```

```
% Función: Función que recibe la señal transmitida y la recibida,
% demodulada la recibida, las compará y devuelve el BER de la
 transmisión. En
% este apartado además, se añadirá ruido dado el vector y se devolverá
 un
% vector de BERs. En este caso al final, se debe añadir una fase de
 desentralazado, dealeatorización y decodificación.
% Input:Signal= La señal modulada que se recibé del receptor. Nf= El
 número de portadoras con datos, viene especificado en
% el standard, el máximo será 96. NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. M= Tipo de modulación
 que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK.  Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 tx_bits= Bits que se transmiten, se usará para
% compararlos con los bits recibidos después de demodular y calcular
 los
% errores. enrejado=  resultado de la función polytrellis que funciona
 como registro
% de desplazamiento. ruido= Vector de ruido SNR.
% Output: BER= El BER computado en la transmisión, con los errores
 dividido
% entre el número de bits transmitidos, en este caso será un vector,
 un
% elemento para cada ruido.
% Bucle para recorrer todo el vector de ruidos
    for i=1:length(ruido)
% Se añade ruido a la señal recibida
        fb = 10*log10( (NFFT/2)/Nf );
        y  = awgn(signal,ruido(i)-fb,'measured');
% Ecualización y demodulación OFDM:
        [Y,pilotos_angulos] = OFDM_Demodulador(y,NFFT,Nofdm);
% Demodulación DMPSK:
        rx_bits_aleatorios = DMPSK_Demod(Y, M, pilotos_angulos);
% Deshacer entrelazado con la llamada a la función DeEntrelazado
        rx_bits_aleatorios = DeEntrelazado(rx_bits_aleatorios(:), M,
 Nofdm, Nf);
% Dealeatorización del vector con la llamada a la función Scrambler

        rx_bits = Scrambler(rx_bits_aleatorios(:)');
% Decodificador
        rx_bits = vitdec(rx_bits,enrejado,5*7,'trunc','hard');
% Cálculo de los errores comparando los transmitidos con los recibidos
        errores = sum(bitxor(tx_bits(:), rx_bits'));
        BER(i) = errores / (Nf*Nofdm*log2(M));
    end
end
```

A6: CalcularErrorConvolutionalEncoderRuido-Canal.m

```
function BER = CalcularErrorConvolutionalEncoderRuidoCanal(signal, Nf,
 NFFT, M ,Nofdm, tx_bits, enrejado, ruido, canal, piloto)
% Función: Función que recibe la señal transmitida y la recibida,
% demodulada la recibida, las compará y devuelve el BER de la
 transmisión. En
% este apartado además, se añadirá ruido dado el vector y se devolverá
 un
% vector de BERs. En este caso al final, se debe añadir una fase de
% desentralazado, dealeatorización y decodificación.  Se añade además
 el canal indicado en el enunciado. Se le
% añadirá un apartado de prefijo cíclico y ecualizador.
% Input:Signal= La señal modulada que se recibé del receptor. Nf= El
 número de portadoras con datos, viene especificado en
% el standard, el máximo será 96. NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. M= Tipo de modulación
 que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK.  Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 tx_bits= Bits que se transmiten, se usará para
% compararlos con los bits recibidos después de demodular y calcular
 los
% errores. enrejado=  resultado de la función polytrellis que funciona
 como registro
% de desplazamiento. ruido= Vector de ruido SNR. piloto= Piloto que se
% usará para el ecualizador Zero-Forcing. canal= respuesta del canal
% indicado en el enunciado.
% Output: BER= El BER computado en la transmisión, con los errores
 dividido
% entre el número de bits transmitidos, en este caso será un vector,
 un
% elemento para cada ruido.
    freq = fft(piloto, NFFT);
% Bucle para recorrer todo el vector de ruidos
    for i=1:length(ruido)
% Se añade ruido a la señal recibida
        fb = 10*log10( (NFFT/2)/Nf );
        y  = awgn(signal,ruido(i)-fb,'measured');
% Efecto del canal:
        y_conv = conv(y, canal);
        y = y_conv(1:length(y));
        y = reshape(y,[NFFT+48, Nofdm]);
% Eliminar prefijo cíclico
        y = y(49:end,:);
% Ecualización y demodulación OFDM:
        [pilotos_angulos , Y] = Ecualizador(y, NFFT, Nofdm, freq);
% Demodulación DMPSK:
        rx_bits_aleatorios = DMPSK_Demod(Y, M, pilotos_angulos);
```

```
% Deshacer el entrelazado
        rx_bits_aleatorios = DeEntrelazado(rx_bits_aleatorios(:), M,
 Nofdm, Nf);
% Dealeatorización del vector con la llamada a la función Scrambler
        rx_bits = Scrambler(rx_bits_aleatorios(:)');
% Decodificador
        rx_bits = vitdec(rx_bits,enrejado,5*7,'trunc','hard');
% Cálculo de los errores comparando los transmitidos con los recibidos
        errores = sum(bitxor(tx_bits(:), rx_bits'));
        BER(i) = errores / (Nf*Nofdm*log2(M));
    end
end
```

A7: CalcularErrorInterleaver.m

```
function BER = CalcularErrorInterleaver(signal, Nf, NFFT, M ,Nofdm,
 tx_bits)
% Función: Función que recibe la señal transmitida y la recibida,
% demodulada la recibida, las compará y devuelve el BER de la
 transmisión.
% En este caso al final, se debe añadir una fase de desentralazado y
 dealeatorización.
% Input:Signal= La señal modulada que se recibé del receptor. Nf= El
 número de portadoras con datos, viene especificado en
% el standard, el máximo será 96. NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. M= Tipo de modulación
 que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK.  Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 tx_bits= Bits que se transmiten, se usará para
% compararlos con los bits recibidos después de demodular y calcular
 los errores.
% Output: BER= El BER computado en la transmisión, con los errores
 dividido
% entre el número de bits transmitidos.
% Demodulación OFDM
    [Y,pilotos_angulos] = OFDM_Demodulador(signal,NFFT,Nofdm);
% Demodulación DMPSK
     rx_bits_aleatorios = DMPSK_Demod(Y, M, pilotos_angulos);
% Dealeatorización con la llamada a la función DeEntrelazado
     rx_bits_aleatorios = DeEntrelazado(rx_bits_aleatorios(:), M,
 Nofdm, Nf);
% Dealeatorización del vector con la llamada a la función Scrambler

     rx_bits = Scrambler(rx_bits_aleatorios(:)');
% Cálculo de los errores comparando los transmitidos con los recibidos
     errores = sum(bitxor(tx_bits(:), rx_bits'));
     BER = errores / (Nf*Nofdm*log2(M));
end
```

A8: CalcularErrorRuido.m

```
function BER = CalcularErrorRuido(signal, Nf, NFFT, M ,Nofdm, tx_bits,
 ruido)
% Función: Función que recibe la señal transmitida y la recibida,
% demodulada la recibida, las compará y devuelve el BER de la
 transmisión.
% En este caso al final, se debe añadir una fase de dealeatorización.
 En
% este apartado además, se añadirá ruido dado el vector y se devolverá
 un
% vector de BERs.
% Input:Signal= La señal modulada que se recibé del receptor. Nf= El
 número de portadoras con datos, viene especificado en
% el standard, el máximo será 96. NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. M= Tipo de modulación
 que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK.  Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 tx_bits= Bits que se transmiten, se usará para
% compararlos con los bits recibidos después de demodular y calcular
 los errores. ruido= Vector de ruido SNR.
% Output: BER= El BER computado en la transmisión, con los errores
 dividido
% entre el número de bits transmitidos, en este caso será un vector,
 un
% elemento para cada ruido.
% Bucle para recorrer todo el vector de ruidos
    for i=1:length(ruido)
% Añadimos ruido a la señal recibida
        fb = 10*log10( (NFFT/2)/Nf );
        y  = awgn(signal,ruido(i)-fb,'measured');
% Demodulación OFDM
        [Y,pilotos_angulos] = OFDM_Demodulador(y,NFFT,Nofdm);
% Demodulación DMPSK
        rx_bits_aleatorios = DMPSK_Demod(Y, M, pilotos_angulos);
% Dealeatorización del vector con la llamada a la función Scrambler
        rx_bits = Scrambler(rx_bits_aleatorios(:)');
% Cálculo de los errores comparando los transmitidos con los recibidos
        errores = sum(xor(tx_bits(:), rx_bits(:)));
        BER(i) = errores / length(tx_bits);
    end
end
```

A9: CalcularErrorRuidoCanal.m

```
function BER = CalcularErrorRuidoCanal(signal, Nf, NFFT, M ,Nofdm,
 tx_bits, ruido, canal)
```

```
% Función: Función que recibe la señal transmitida y la recibida,
% demodulada la recibida, las compará y devuelve el BER de la
 transmisión.
% En este caso al final, se debe añadir una fase de dealeatorización.
 En
% este apartado además, se añadirá ruido dado el vector y se devolverá
 un
% vector de BERs. Se añade además el canal indicado en el enunciado.
% Input:Signal= La señal modulada que se recibé del receptor. Nf= El
 número de portadoras con datos, viene especificado en
% el standard, el máximo será 96. NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. M= Tipo de modulación
 que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK.  Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 tx_bits= Bits que se transmiten, se usará para
% compararlos con los bits recibidos después de demodular y calcular
 los errores. ruido= Vector de ruido SNR.
% Output: BER= El BER computado en la transmisión, con los errores
 dividido
% entre el número de bits transmitidos, en este caso será un vector,
 un
% elemento para cada ruido.
% Bucle para recorrer todo el vector de ruidos
    for i=1:length(ruido)
% Añadimos ruido a la señal recibida
        fb = 10*log10( (NFFT/2)/Nf );
        y  = awgn(signal,ruido(i)-fb,'measured');
% Efecto del canal:
        y_conv = conv(y, canal);
        y = y_conv(1:length(y));
% Demodulación OFDM
        [Y,pilotos_angulos] = OFDM_Demodulador(y,NFFT,Nofdm);
% Demodulación DMPSK
        rx_bits_aleatorios = DMPSK_Demod(Y, M, pilotos_angulos);
% Dealeatorización del vector con la llamada a la función Scrambler
        rx_bits = Scrambler(rx_bits_aleatorios(:)');
% Cálculo de los errores comparando los transmitidos con los recibidos
        errores = sum(bitxor(tx_bits(:), rx_bits(:)));
        BER(i) = errores / (Nf*Nofdm*log2(M));
    end
end
```

A10: CalcularErrorRuidoCanalEcualizado.m

```
function BER = CalcularErrorRuidoCanalEcualizado(signal, Nf, NFFT,
 M ,Nofdm, tx_bits, ruido, canal, piloto)
% Función: Función que recibe la señal transmitida y la recibida,
% demodulada la recibida, las compará y devuelve el BER de la
 transmisión.
```

```
% En este caso al final, se debe añadir una fase de dealeatorización.
 En
% este apartado además, se añadirá ruido dado el vector y se devolverá
 un
% vector de BERs. Se añade además el canal indicado en el enunciado.
 Se le
% añadirá un apartado de prefijo cíclico y ecualizador.
% Input:Signal= La señal modulada que se recibé del receptor. Nf= El
 número de portadoras con datos, viene especificado en
% el standard, el máximo será 96. NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. M= Tipo de modulación
 que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK.  Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 tx_bits= Bits que se transmiten, se usará para
% compararlos con los bits recibidos después de demodular y calcular
 los
% errores. piloto= Piloto que se usará para el ecualizador Zero-
Forcing. canal= respuesta del canal
% indicado en el enunciado.
% ruido= Vector de ruido SNR.
% Output: BER= El BER computado en la transmisión, con los errores
 dividido
% entre el número de bits transmitidos, en este caso será un vector,
 un
% elemento para cada ruido.
    freq = fft(piloto, NFFT);
% Bucle para recorrer todo el vector de ruidos
    for i=1:length(ruido)
% Se añade ruido a la señal recibida
        fb = 10*log10( (NFFT/2)/Nf );
        y  = awgn(signal,ruido(i)-fb,'measured');
% Efecto del canal:
        y_conv = conv(y, canal);
        y = y_conv(1:length(y));
        y = reshape(y,[NFFT+48, Nofdm]);
% Eliminar prefijo cíclico
        y = y(49:end,:);
% Ecualización y demodulación OFDM:
        [pilotos_angulos , Y] = Ecualizador(y, NFFT, Nofdm, freq);
% Demodulación DMPSK:
        rx_bits_aleatorios = DMPSK_Demod(Y, M, pilotos_angulos);
% Dealeatorización del vector con la llamada a la función Scrambler
        rx_bits = Scrambler(rx_bits_aleatorios(:)');
% Cálculo de los errores comparando los transmitidos con los recibidos
        errores = sum(bitxor(tx_bits(:), rx_bits(:)));
        BER(i) = errores / (Nf*Nofdm*log2(M));
    end
end
```

A11: CalcularErrorRuidoCanalPrefijoCiclico.m

```
function BER = CalcularErrorRuidoCanalPrefijoCiclico(signal, Nf, NFFT,
 M ,Nofdm, tx_bits, ruido, canal)
    for i=1:length(ruido)
        fb = 10*log10( (NFFT/2)/Nf );
        y  = awgn(signal,ruido(i)-fb,'measured');
        % Efecto del canal:
        y_conv = conv(y, canal);
        y = y_conv(1:length(y));
        y = reshape(y,[NFFT+48, Nofdm]);
        % Eliminar prefijo cíclico
        y = y(49:end,:);
        [Y,pilotos_angulos] = OFDM_Demodulador(y,NFFT,Nofdm);
        rx_bits_aleatorios = DMPSK_Demod(Y, M, pilotos_angulos);
        rx_bits = Scrambler(rx_bits_aleatorios(:)');
        errores = sum(bitxor(tx_bits(:), rx_bits(:)));
        BER(i) = errores / (Nf*Nofdm*log2(M));
    end
end
```

A12: CalcularErrorScrambler.m

```
function BER = CalcularErrorScrambler(signal, Nf, NFFT, M ,Nofdm,
 tx_bits)
% Función: Función que recibe la señal transmitida y la recibida,
% demodulada la recibida, las compará y devuelve el BER de la
 transmisión.
% En este caso al final, se debe añadir una fase de dealeatorización.
% Input:Signal= La señal modulada que se recibé del receptor. Nf= El
 número de portadoras con datos, viene especificado en
% el standard, el máximo será 96. NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. M= Tipo de modulación
 que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK.  Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 tx_bits= Bits que se transmiten, se usará para
% compararlos con los bits recibidos después de demodular y calcular
 los errores.
% Output: BER= El BER computado en la transmisión, con los errores
 dividido
% entre el número de bits transmitidos.
% Demodulación OFDM
    [Y,pilotos_angulos] = OFDM_Demodulador(signal,NFFT,Nofdm);
% Demodulación DMPSK
    rx_bits_aleatorios = DMPSK_Demod(Y, M, pilotos_angulos);
% Dealeatorización del vector con la llamada a la función Scrambler
    rx_bits = Scrambler(rx_bits_aleatorios(:)');
```

```
% Cálculo de los errores comparando los transmitidos con los recibidos
    errores = sum(bitxor(tx_bits(:), rx_bits(:)));
    BER = errores / (Nf*Nofdm*log2(M));
end
```

A13: calcularNumeroPaquetesFEC.m

```
function numPaquetes = calcularNumeroPaquetesFEC(numBits, M, Nofdm,
 Nf)
    for i=1:numBits
        tx(i)=randi([0,1]);
    end
    Nbits=(Nofdm*log2(M)*Nf/2)-8;
    numPaquetes = ceil(length(tx)/Nbits);
end
```

A14: calcularNumeroPaquetesnoFEC.m

```
function numPaquetes = calcularNumeroPaquetesFEC(numBits, M, Nofdm,
 Nf)
    for i=1:numBits
        tx(i)=randi([0,1]);
    end
    Nbits=Nofdm*log2(M)*Nf;
    numPaquetes = ceil(length(tx)/Nbits);
end
```

A15: Convolutional_Encoder.m

```
function [outputArg1,outputArg2] = Convolutional_Encoder(tx_bits)
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

end
```

A16: D8PSK_BER.m

```
function Int = D8PSK_BER(SNR_dB)
```

```
M = 8;
k = log2(M);
N = 1e4;

SNR = 10.^(SNR_dB/10);
EbNo = SNR/k;

phi = linspace(-pi/2,pi/2,N);

Int = zeros(length(SNR_dB),1); % Pre-alloc
for iter = 1:length(SNR_dB)
    A = 1-(cos(pi/M)*cos(phi));
    B = -k*EbNo(iter);
    C = 1-(cos(pi/M)*cos(phi));

    I = exp(A.*B)./C;

    deltaPhi = phi(2)-phi(1);
    Int(iter) = sin(pi/M)/4/pi*sum(I(1:end-1))*deltaPhi;
end
%
% close all
% semilogy(SNR_dB,Int,'*')
```

A17: DBPSK_BER.m

```
function theoryBerDBPSK = DBPSK_BER(SNR_dB)

if nargin == 0
    SNR_dB = -5:20;
end

theoryBerDBPSK = (1/2)*exp(-(10.^(SNR_dB(1,:)/10)));

if nargin == 0
    figure
    semilogy(SNR_dB,theoryBerDBPSK,'-*')
    xlabel('SNR [dB]')
    ylabel('BER')
    grid
end
```

A18: DeEntrelazado.m

```
function rx_bits_no_interleaver = DeEntrelazado(rx_bits, M, Nofdm, Nf)
% Función: Función que se utiliza para ejecutar el desentrelazado de
 los bits,
```

```
% se transforman los bits en una matriz que se transpone y se vuelve a
% convertir en un vector, se utiliza para evitar que se produzcan
 muchos
% errores seguidos. Se realiza justo la operación inversa al
 entrelazado.
% Input: rx_bits= vector de bits a desentrelazar. M= Tipo de
 modulación que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK. Nf= El número de portadoras con datos, viene
 especificado en
% el standard, el máximo será 96, Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
% Output: rx_bits_no_interleaver= vector de bits desentrelazados.
            rx_bits_no_interleaver=[];

            for i=0:Nofdm-1

                if(M==2)

                    matriz= vec2mat(rx_bits(i*(12*8)+ 1: (i
+1)*(12*8)),8,12);

                elseif(M==4)

                    matriz= vec2mat(rx_bits(i*(12*16)+ 1: (i
+1)*(12*16)),16,12);

                elseif(M==8)

                    matriz= vec2mat(rx_bits(i*(16*18)+ 1: (i
+1)*(16*18)),16,18);

                end

                matriz=matriz(:)';

                rx_bits_no_interleaver=[rx_bits_no_interleaver
 matriz];

            end

    rx_bits_no_interleaver = rx_bits_no_interleaver';

end
```

A19: DMPSK_Demod.m

```
function rx_bits = DMPSK_Demod(Y, M,piloto_fase)
% Función: Función que calcula la modulación DBPSK, DQPSK y D8PSK,
% dependiendo de M y usando los pilotos para calcular la primera
% diferencia.
```

```
% Output: Y= Vector de salida del demodulador OFDM que tiene el vector
 que se introducirá en el demodulador.
% Input:M= Tipo de modulación que se empleará, M = 2 DBPSK, M = 4
 DQPSK,
% M = 8 D8PSK. piloto_fase= Piloto para el cálculo de la primera
 diferencia
% para la modulación diferencial.
    if(M==2)
        demod = comm.DBPSKDemodulator(pi/4);
        rx_bits = demod(Y(:));
    elseif(M==4)
        demod = comm.DQPSKDemodulator('BitOutput',true);
        rx_bits = demod(Y(:));
    elseif(M==8)
        demod = comm.DPSKDemodulator(8,pi/8,'BitOutput',true);
        rx_bits = demod(Y(:));
    end

  %demodDPSK=modem.dpskdemod('M',M,'OutputType','Bit','InitialPhase',piloto_fase);
    %rx_bits=demodulate(demodDPSK,Y);

end
```

A20: DMPSK_Modulador.m

```
function modSymbols = DMPSK_Modulador(txbits, M, piloto_fase)
% Función: Función que calcula la modulación DBPSK, DQPSK y D8PSK,
% dependiendo de M y usando los pilotos para calcular la primera
% diferencia.
% Output: txbits= Vector de bits a los que se realizará la modulación.
 M= Tipo de modulación que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK. piloto_fase= Piloto para el cálculo de la primera
 diferencia
% para la modulación diferencial.
% Input: modSymbols= Resultado de la modulación del vector de bits
 txbits.
    if(M==2)
        mod = comm.DBPSKModulator(pi/4);
        modSymbols= mod(txbits(:));
    elseif(M==4)
        mod = comm.DQPSKModulator('BitInput',true);
        modSymbols= mod(txbits(:));
    elseif(M==8)
        mod = comm.DPSKModulator(8,pi/8,'BitInput',true);
        modSymbols= mod(txbits(:));
    end
    modSymbols=reshape(modSymbols, [96,63]);

  %modDPSK=modem.dpskmod('M',M,'InputType','Bit','InitialPhase',piloto_fase);
    %modSymbols=modulate(modDPSK,txbits);
end
```

A21: DQPSK_BER.m

```
function theoryBer_dqpsk_noncoh = DQPSK_BER(SNR_dB)

if nargin == 0
    SNR_dB = -5:20;
end

M = 4;
k = log2(M);
Eb_N0_dB = SNR_dB-10*log10(k);

theoryBer_dqpsk_noncoh = zeros(1,length(SNR_dB)); % Pre-alloc
for snrIter = 1:length(SNR_dB)
    a = sqrt(2*10.^(Eb_N0_dB(snrIter)/10)*(1-sqrt(1/2)));
    b = sqrt(2*10.^(Eb_N0_dB(snrIter)/10)*(1+sqrt(1/2)));

    k_bessel = 0:10;
    temp = exp(-((a.^2+b.^2)/2)).*sum((a/
b).^k_bessel.*besseli(k_bessel,a*b));
    theoryBer_dqpsk_noncoh(snrIter) = temp - 0.5*besseli(0,a*b)*exp(-
((a.^2+b.^2)/2));
end


if nargin == 0
    figure
    semilogy(SNR_dB,theoryBer_dqpsk_noncoh,'-*')
    xlabel('SNR [dB]')
    ylabel('BER')
    grid
end
```

A22: Ecualizador.m

```
function [piloto_fase , Y] = Ecualizador(y, NFFT, Nofdm, freq)
% Función: Función que se utiliza para ecualizar, se divide el vector
 que
% se recibe con el que debería ser para saber cuál es la respuesta del
% canal y poder contrarrestarla después.
% Input: y= Señal sin prefijo cíclico para ecualizarla .NFFT= Elemento
 que indica el tamaño de la
% FFT, se usará a la hora de calcular la FFT. Nofdm= Número de
 símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 freq= Piloto que se usará para el ecualizador Zero-Forcing.
% Output: Y= Señal demodulada en OFDM sin los pilotos y ecualizada.
 piloto_fase= Pilotos
```

```
% que se introdujeron en el modulador y que se devuelven para la
% demodulacion DMPSK.
     Y = fft(y,NFFT);
    Respuesta_canal = Y(:,1)./freq;

    for i=1:Nofdm
        Y_ecualizada(:,i) = Y(:,i)./Respuesta_canal;
    end
    piloto_fase = angle(Y_ecualizada(87,:));
    Y = Y_ecualizada(88:183,:);
end
```

A23: Entrelazar.m

```
function tx_bits_interleaver= Entrelazar(tx_bits, M, Nofdm, Nf)
% Función: Función que se utiliza para ejecutar el entrelazado de los
 bits,
% se transforman los bits en una matriz que se transpone y se vuelve a
% convertir en un vector, se utiliza para evitar que se produzcan
 muchos
% errores seguidos.
% Input: tx_bits= vector de bits a entrelazar. M= Tipo de modulación
 que se empleará, M = 2 DBPSK, M = 4 DQPSK,
% M = 8 D8PSK. Nf= El número de portadoras con datos, viene
 especificado en
% el standard, el máximo será 96, Nofdm= Número de símbolos por
% trama para la modulación OFDM, especificado en el standard también.
% Output: tx_bits_interleaver= vector de bits entrelazados.
    tx_bits_interleaver_def=[];

        for i=1:Nofdm

           tx_bits_interleaver = tx_bits(:,i)';
           tx_bits_interleaver = flip(tx_bits_interleaver);
           if M==2
               nc = 12;
               nf = 8;
               tx_bits_interleaver = reshape(tx_bits_interleaver,
 [nc,nf])';

           elseif M==4
               nc = 12;
               nf = 16;
               tx_bits_interleaver = reshape(tx_bits_interleaver,
 [nc,nf])';

           else
               nc = 18;
               nf = 16;
```

```
                tx_bits_interleaver = reshape(tx_bits_interleaver,
 [nc,nf])';
            end

            tx_bits_interleaver = flip(tx_bits_interleaver);


            for j=1:nc
                intermedio = tx_bits_interleaver(:,j)';
                if j==1
                    tx_bits_interleaver_s = intermedio;
                else
                    tx_bits_interleaver_s =
 horzcat(intermedio,tx_bits_interleaver_s);
                end
            end
            tx_bits_interleaver_def=
 horzcat(tx_bits_interleaver_def,tx_bits_interleaver_s);

        end
    tx_bits_interleaver = vec2mat(tx_bits_interleaver_def,Nf*log2(M));
    tx_bits_interleaver=transpose(tx_bits_interleaver);
end
```

A24: ModulacionConvolutionalEncoder.m

```
function [tx_bits, signal] = ModulacionConvolutionalEncoder(M, Nf,
 NFFT, Nofdm, enrejado)
% Función: Función que crea un vector a transmitir dados una serie de
% parámetros y que devuelve el vector que se va a transmitir y su
% modulación OFDM que finalmente se transmitirá. Se añade además
 después de
% crear el vector de bits que transmitimos, una parte en la que se
% aleatoriza ese vector, una parte en la que se realiza el entrelazado
 y un codificador convolucional que además cambiará el número de bits
 transmitidos.
% Input: M= Tipo de modulación que se empleará, M = 2 DBPSK, M = 4
 DQPSK,
% M = 8 D8PSK. Nf= El número de portadoras con datos, viene
 especificado en
% el standard, el máximo será 96, NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. Nofdm= Número de
 símbolos por
% trama para la modulación OFDM, especificado en el standard también.
% enrejado= resultado de la función polytrellis que funciona como
 registro
% de desplazamiento.
% Output: tx_bits= Bits que se transmiten, se usarán para
 posteriormente
```

```
% compararlos con los bits recibidos y calcular los errores. Signal=
 Bits que vamos a transmitir.
% Hay que modicar los bits que se envían a la mitad:
% Bits que vamos a transmitir:
    tx_bits = Nofdm*log2(M)*Nf/2;
    tx_bits = round(rand(tx_bits,1));
    tx_bits(end-8+1:end) = 0;
% Codificación
    tx_bits_codificados = convenc(tx_bits, enrejado);
% Aleatorización del vector de bits a transmitir.
    tx_bits_aleatorios = Scrambler(tx_bits_codificados');
    tx_bits_aleatorios = double(vec2mat(tx_bits_aleatorios',
 Nf*log2(M)));
    tx_bits_aleatorios = tx_bits_aleatorios';
% Obtenemos los pilotos:
    [piloto_fase, piloto_mod]=vectorPrefijos(Nofdm);
% Entrelazado
    tx_bits_aleatorios = Entrelazar(tx_bits_aleatorios, M, Nofdm, Nf);
% Modulación DMPSK:
    mod_DMPSK = DMPSK_Modulador(tx_bits_aleatorios, M, piloto_fase);
% Modulación OFDM:
    signal=OFDM_Modulador(mod_DMPSK,piloto_fase,NFFT,Nofdm);
end
```

A25: ModulacionConvolutionalEncoderPC.m

```
function [tx_bits, signal, piloto_ecualizador] =
 ModulacionConvolutionalEncoderPC(M, Nf, NFFT, Nofdm, enrejado)
% Función: Función que crea un vector a transmitir dados una serie de
% parámetros y que devuelve el vector que se va a transmitir y su
% modulación OFDM que finalmente se transmitirá. Se añade además
 después de
% crear el vector de bits que transmitimos, una parte en la que se
% aleatoriza ese vector, una parte en la que se añade el prefijo
 cíclico ,una parte en la que se realiza el entrelazado y un
 codificador convolucional que además cambiará el número de bits
 transmitidos.
% Input: M= Tipo de modulación que se empleará, M = 2 DBPSK, M = 4
 DQPSK,
% M = 8 D8PSK. Nf= El número de portadoras con datos, viene
 especificado en
% el standard, el máximo será 96, NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. Nofdm= Número de
 símbolos por
% trama para la modulación OFDM, especificado en el standard también.
 enrejado= resultado de la función polytrellis que funciona como
 registro
% de desplazamiento.
% Output: tx_bits= Bits que se transmiten, se usarán para
 posteriormente
```

```
% compararlos con los bits recibidos y calcular los errores. Signal=
% La señal modulada que se enviará al receptor. piloto_ecualizador=
 Piloto
% que se usará para el ecualizador Zero-Forcing.
% Hay que modicar los bits que se envían a la mitad:
% Bits que vamos a transmitir:
    tx_bits = Nofdm*log2(M)*Nf/2;
    tx_bits = round(rand(tx_bits,1));
    tx_bits(end-8+1:end) = 0;
% Codificación
    tx_bits_codificados = convenc(tx_bits, enrejado);
% Aleatorización del vector de bits a transmitir.
    tx_bits_aleatorios = Scrambler(tx_bits_codificados');
    tx_bits_aleatorios = double(vec2mat(tx_bits_aleatorios',
 Nf*log2(M)));
    tx_bits_aleatorios = tx_bits_aleatorios';
% Obtenemos los pilotos:
    [piloto_fase, piloto_mod]=vectorPrefijos(Nofdm);
% Entrelazado
    tx_bits_aleatorios = Entrelazar(tx_bits_aleatorios, M, Nofdm, Nf);
% Modulación DMPSK:
    mod_DMPSK = DMPSK_Modulador(tx_bits_aleatorios, M, piloto_fase);
% Modulación OFDM:
    signal=OFDM_Modulador(mod_DMPSK,piloto_fase,NFFT,Nofdm);
    piloto_ecualizador = signal(:,1);
% Se añade el prefijo cíclico y se devuelve como un vector.
    signal = addPrefijoCiclico(signal);
end
```

A26: ModulacionInterleaver.m

```
function [tx_bits, signal] = ModulacionInterleaver(M, Nf, NFFT, Nofdm)
% Función: Función que crea un vector a transmitir dados una serie de
% parámetros y que devuelve el vector que se va a transmitir y su
% modulación OFDM que finalmente se transmitirá. Se añade además
 después de
% crear el vector de bits que transmitimos, una parte en la que se
% aleatoriza ese vector y una parte en la que se realiza el
 entrelazado.
% Input: M= Tipo de modulación que se empleará, M = 2 DBPSK, M = 4
 DQPSK,
% M = 8 D8PSK. Nf= El número de portadoras con datos, viene
 especificado en
% el standard, el máximo será 96, NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. Nofdm= Número de
 símbolos por
% trama para la modulación OFDM, especificado en el standard también.
% Output: tx_bits= Bits que se transmiten, se usarán para
 posteriormente
```

```
% compararlos con los bits recibidos y calcular los errores. Signal=
 Bits que vamos a transmitir.
    tx_bits = Nofdm*log2(M)*Nf;
    tx_bits = round(rand(tx_bits,1));
% Aleatorización del vector de bits a transmitir.
    tx_bits_aleatorios = Scrambler(tx_bits');
    tx_bits_aleatorios = double(vec2mat(tx_bits_aleatorios',
 Nf*log2(M)));
    tx_bits_aleatorios = tx_bits_aleatorios';
% Obtenemos los pilotos:
    [piloto_fase, piloto_mod]=vectorPrefijos(Nofdm);
% Entrelazado
    tx_bits_aleatorios = Entrelazar(tx_bits_aleatorios, M, Nofdm, Nf);
% Modulación DMPSK:
    mod_DMPSK = DMPSK_Modulador(tx_bits_aleatorios, M, piloto_fase);
% Modulación OFDM:
    signal=OFDM_Modulador(mod_DMPSK,piloto_fase,NFFT,Nofdm);
end
```

A27: ModulacionOFDM.m

```
function [tx_bits, signal] = ModulacionOFDM(M, Nf, NFFT, Nofdm)
% Función: Función que crea un vector a transmitir dados una serie de
% parámetros y que devuelve el vector que se va a transmitir y su
% modulación OFDM que finalmente se transmitirá.
% Input: M= Tipo de modulación que se empleará, M = 2 DBPSK, M = 4
 DQPSK,
% M = 8 D8PSK. Nf= El número de portadoras con datos, viene
 especificado en
% el standard, el máximo será 96, NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. Nofdm= Número de
 símbolos por
% trama para la modulación OFDM, especificado en el standard también.
% Output: tx_bits= Bits que se transmiten, se usarán para
 posteriormente
% compararlos con los bits recibidos y calcular los errores. Signal=
% La señal modulada que se enviará al receptor.
    tx_bits = Nofdm*log2(M)*Nf;
    tx_bits = round(rand(tx_bits,1));
    tx_bits = vec2mat(tx_bits, Nf*log2(M));
    tx_bits = tx_bits';
% Obtenemos los pilotos:
    [piloto_fase, piloto_mod]=vectorPrefijos(Nofdm);
% Modulación DMPSK:
    mod_DMPSK = DMPSK_Modulador(tx_bits, M, piloto_fase);
% Modulación OFDM:
    signal=OFDM_Modulador(mod_DMPSK,piloto_fase,NFFT,Nofdm);
% Se devuelve como un vector:
    signal = signal(:)';
end
```

A28: ModulacionOFDMConPrefijoCiclico.m

```
function [tx_bits, signal] = ModulacionOFDMConPrefijoCiclico(M, Nf,
 NFFT, Nofdm)
    % Bits que vamos a transmitir:
    tx_bits = Nofdm*log2(M)*Nf;
    tx_bits = round(rand(tx_bits,1));
    tx_bits_aleatorios = Scrambler(tx_bits');
    tx_bits_aleatorios = double(vec2mat(tx_bits_aleatorios',
 Nf*log2(M)));
    tx_bits_aleatorios = tx_bits_aleatorios';
    [piloto_fase, piloto_mod]=vectorPrefijos(Nofdm);
    % Modulación:
    mod_DMPSK = DMPSK_Modulador(tx_bits_aleatorios, M, piloto_fase);
    signal=OFDM_Modulador(mod_DMPSK,piloto_fase,NFFT,Nofdm);
    signal = addPrefijoCiclico(signal)
end
```

A29: ModulacionOFDMEcualizacion.m

```
function [tx_bits, signal, piloto_ecualizador] =
 ModulacionOFDMEcualizacion(M, Nf, NFFT, Nofdm)
% Función: Función que crea un vector a transmitir dados una serie de
% parámetros y que devuelve el vector que se va a transmitir y su
% modulación OFDM que finalmente se transmitirá. Se añade además
 después de
% crear el vector de bits que transmitimos, una parte en la que se
% aleatoriza ese vector. Se le añadirá una parte en la que se añade el
% prefijo cíclico y además se devuelve la señal que se usará a la hora
 de
% comparar para ecualizar.
% Input: M= Tipo de modulación que se empleará, M = 2 DBPSK, M = 4
 DQPSK,
% M = 8 D8PSK. Nf= El número de portadoras con datos, viene
 especificado en
% el standard, el máximo será 96, NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. Nofdm= Número de
 símbolos por
% trama para la modulación OFDM, especificado en el standard también.
% Output: tx_bits= Bits que se transmiten, se usarán para
 posteriormente
% compararlos con los bits recibidos y calcular los errores. Signal=
% La señal modulada que se enviará al receptor. piloto_ecualizador=
 Piloto
% que se usará para el ecualizador Zero-Forcing.
```

```
% Bits que vamos a transmitir:
    tx_bits = Nofdm*log2(M)*Nf;
    tx_bits = round(rand(tx_bits,1));
% Aleatorización del vector de bits a transmitir.
    tx_bits_aleatorios = Scrambler(tx_bits');
    tx_bits_aleatorios = double(vec2mat(tx_bits_aleatorios',
 Nf*log2(M)));
    tx_bits_aleatorios = tx_bits_aleatorios';
% Obtenemos los pilotos:
    [piloto_fase, piloto_mod]=vectorPrefijos(Nofdm);
% Modulación DMPSK:
    mod_DMPSK = DMPSK_Modulador(tx_bits_aleatorios, M, piloto_fase);
% Modulación OFDM:
    signal=OFDM_Modulador(mod_DMPSK,piloto_fase,NFFT,Nofdm);
    piloto_ecualizador = signal(:,1);
% Se añade el prefijo cíclico y se devuelve como un vector.
    signal = addPrefijoCiclico(signal);
end
```

A30: ModulacionOFDMScrambler.m

```
function [tx_bits, signal] = ModulacionOFDMScrambler(M, Nf, NFFT,
 Nofdm)
% Función: Función que crea un vector a transmitir dados una serie de
% parámetros y que devuelve el vector que se va a transmitir y su
% modulación OFDM que finalmente se transmitirá. Se añade además
 después de
% crear el vector de bits que transmitimos, una parte en la que se
% aleatoriza ese vector.
% Input: M= Tipo de modulación que se empleará, M = 2 DBPSK, M = 4
 DQPSK,
% M = 8 D8PSK. Nf= El número de portadoras con datos, viene
 especificado en
% el standard, el máximo será 96, NFFT= Elemento que indica el tamaño
 de la
% FFT, se usará a la hora de calcular la FFT. Nofdm= Número de
 símbolos por
% trama para la modulación OFDM, especificado en el standard también.
% Output: tx_bits= Bits que se transmiten, se usarán para
 posteriormente
% compararlos con los bits recibidos y calcular los errores. Signal=
% La señal modulada que se enviará al receptor.
% Bits que vamos a transmitir:
    tx_bits = Nofdm*log2(M)*Nf;
    tx_bits = round(rand(tx_bits,1));
% Aleatorización del vector de bits a transmitir.
    tx_bits_aleatorios = Scrambler(tx_bits');
    tx_bits_aleatorios = double(vec2mat(tx_bits_aleatorios',
 Nf*log2(M)));
    tx_bits_aleatorios = tx_bits_aleatorios';
```

```
% Obtenemos los pilotos:
    [piloto_fase, piloto_mod]=vectorPrefijos(Nofdm);
% Modulación DMPSK:
    mod_DMPSK = DMPSK_Modulador(tx_bits_aleatorios, M, piloto_fase);
% Modulación OFDM:
    signal=OFDM_Modulador(mod_DMPSK,piloto_fase,NFFT,Nofdm);
% Se devuelve como un vector:
    signal = signal(:)';
end
```

A31: OFDM_Demodulador.m

```
function [Y,piloto_fase] = OFDM_Demodulador(signal,NFFT,Nofdm)
% Función: Función que realiza de demodulación OFDM inversa, haciendo
 la
% FFT en vez de la IFFT.
% Input: signal= La señal modulada que se recibé del receptor. NFFT=
 Elemento que indica el tamaño de la
% FFT, se usará a la hora de calcular la FFT. Nofdm= Número de
 símbolos por
% trama para la modulación OFDM, especificado en el standard también.
% Output: Y= Señal demodulada en OFDM sin los pilotos. piloto_fase=
 Pilotos
% que se introdujeron en el modulador y que se devuelven para la
% demodulacion DMPSK.
    y = reshape(signal, [NFFT, Nofdm]);
    Y = fft(y,NFFT);
    piloto_fase = angle((Y(87,:)));
    Y = Y(88:183,:);
end
```

A32: OFDM_Modulador.m

```
function x = OFDM_Modulador(mod_DMPSK,piloto_fase,NFFT,Nofdm)
% Función: Función que calcula la modulación OFDM, es decir, colocá
 los
% valores modulados en DMPSK en el lugar indicado en el standard y
 realiza
% la IFFT.
% Input: mod_DMPSK= vector modulado a partir del vector a transmitir.
% piloto_fase= Vector de pilotos que se colocará antés del vector
 modulado.
% NFFT= Elemento que indica el tamaño de la
% FFT, se usará a la hora de calcular la FFT. Nofdm= Número de
 símbolos por
% trama para la modulación OFDM, especificado en el standard también.
% Output: x= Resultado de la modulación en una matriz de NFFTxNofdm.
    X = zeros(NFFT, Nofdm);
```

```
    X(87,:) = exp(1i*piloto_fase); % piloto
    X(88:183,:) = mod_DMPSK;

    X(331,:)=flipud(conj( X(87,:))); % piloto
    X(332:427,:) = flipud(conj(X(88:183,:)));

    x = ifft(X,NFFT,'symmetric')*NFFT;
end
```

A33: Scrambler.m

```
function [txbits_aleatorizados] = Scrambler(tx_bits)
% Función: Función que convierte el vector de bits que entra, en un
 vector
% aleatorizado aplicando un xor entre los bits y un vector estipulado
 en el
% standard.
% Input: Vector de bits que se desea aleatorizar o dealeatorizar.
% Output: Vector de bits dealeatorizado o aleatorizado, según lo que
% corresponda.

% Vector de aleatorización que vamos a usar que viene especificado en
 la
% documentación prime.
     pref = [0 0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 1 1, ...
        0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0, ...
        1 1 0 0 0 1 0 1 1 1 0 1 0 1 1 0 1 1 0, ...
        0 0 0 0 1 1 0 0 1 1 0 1 0 1 0 0 1 1 1, ...
        0 0 1 1 1 1 0 1 1 0 1 0 0 0 0 1 0 1 0, ...
        1 0 1 1 1 1 1 0 1 0 0 1 0 1 0 0 0 1 1, ...
        0 1 1 1 0 0 0 1 1 1 1 1 1 1 1];

    cadena=[];
    %%
    % Si los bits que enviamos son menos que los del vector de
 aleatorización:
    if length(tx_bits) < length(pref)
        cadena = [cadena pref(1:length(tx_bits))];
    else
        %%
        % En el caso de que los bits que transmitimos son mayores que
 los del
        % vector de aleatorización:
        while length(cadena) < length(tx_bits) - length(pref)
            cadena = [cadena pref];
        end
        cadena = [cadena pref(1:length(tx_bits)-length(cadena))];
    end
    %%
```

```
    % Hacemos el xor de los bits que transmitimos con los de la cadena
 de
    % ayuda en la que está el vector de aleatorizacion.
     txbits_aleatorizados = xor(tx_bits, cadena);
end
```

A34: vectorPrefijos.m

```
function [pref_fase, pref] = vectorPrefijos(Nsimbolos)
% Función: Función que dado el número de símbolos que se enviarán,
% devolverá el vector que se usará de pilotos, ya dividido en fase y
% módulo.
% Input: Nsimbolos= Número de símbolos que se usarán, siempre será
 Nofdm.
% Output: pref_fase= Los pilotos ya en fase (ángulos). pref= Los
 pilotos en
% módulo.
pref = [0 0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 1 1, ...
        0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0, ...
        1 1 0 0 0 1 0 1 1 1 0 1 0 1 1 0 1 1 0, ...
        0 0 0 0 1 1 0 0 1 1 0 1 0 1 0 0 1 1 1, ...
        0 0 1 1 1 1 0 1 1 0 1 0 0 0 0 1 0 1 0, ...
        1 0 1 1 1 1 1 0 1 0 0 1 0 1 0 0 0 1 1, ...
        0 1 1 1 0 0 0 1 1 1 1 1 1 1 1];
pref = pref(1:Nsimbolos);
pref_fase = pref*pi;

end
```

Published with MATLAB® R2019b