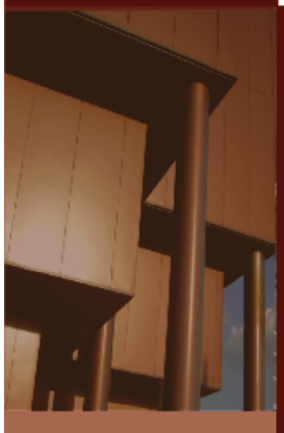
# Electrónica Digital de Comunicaciones
# 1º Máster Ingeniería de Telecomunicación

## Proyecto final de la asignatura



Autor: Fco. Javier Vargas García-Donas

**Escuela Técnica Superior de Ingeniería**
**Universidad de Sevilla**

Sevilla, 2018

# ÍNDICE

# 1 MATLAB

## 1.1. Introducción

### 1.1.1 Fichero del proyecto MATLAB

El fichero que se describe a continuación se encuentra en:

```
./MATLAB/tx_rx_dvbt_mult_symb.m
```

### 1.1.2 Otras consideraciones

Para la realización del simulador del transmisor y receptor DVBT en MATLAB se ha dividido en diferentes subapartados comentados adecuadamente en el código adjunto. Cada subapartado se distingue por un comentado doble

```
%% Subapartado
```

Buscando ser coherentes con la codificación realizada se procede a explicar cada subapartado con el objetivo de dar una visión específica de la resolución de cada problema para obtener al final un enfoque general del programa implementado.

## 1.2. Código

#### 1.1.2.1 Datos iniciales

La simulación de un transmisor y un receptor DVBT exige la definición de una serie de parámetros de funcionamiento. El estándar que define la norma especifica distintos tipos de funcionamiento tanto para el transmisor como para el receptor. Además, propone distintos canales tipo que se dan con frecuencia en este tipo de comunicaciones. Los siguientes parámetros determinan el funcionamiento del sistema de comunicaciones a simular:

- NFFT: Número de portadoras totales potencialmente utilizables en un símbolo OFDM. Se ha determinado realizar la simulación para el modo de funcionamiento 2K, es decir, 2048 portadoras.
- NCP: Para cada modo de funcionamiento existen 4 tipos de prefijos cíclicos, posibles: 1/2, 1/8, 1/16 o 1/32. Para la realización de esta simulación se ha determinado usar un prefijo cíclico de 1/32.
- Lsimb: Número de portadoras usadas por símbolo OFDM, de las NFFT portadoras disponibles se modularán Lsimb (Lenght Simbol).
- Lpilot: Número portadoras piloto por símbolo. El estándar DVBT establece un cierto número de pilotos fijos para propósitos de estimación del canal. Este parámetro determina de las Lsimb portadoras usadas,

cuántas de ellas serán pilotos para la estimación del canal.

- Ldata: Número de portadoras de datos por símbolo. Si a las usadas le quitamos los pilotos obtenemos los datos útiles a transmitir.
- NUM_SYMB: Número de símbolos OFDM por portadora en el tiempo. Con propósitos de simplificación y coste computacional se transmitirá un solo símbolo OFDM.
- SEED: Semilla para la generación de números pseudoaleatorios
- CONSTEL: El estándar DVBT establece distintos tipos de constelación para la transmisión. Esta simulación se ha realizado para la constelación QPSK, pero también funciona para una constelación BPSK (cámbiese adecuadamente para su comprobación).
- SNR : Relación señal a ruido para la transmisión dada en decibelios.

#### 1.1.2.2 Constelación

Se determinan los símbolos con los que se codificará la información en función de la constelación especificada en el parámetro CONSTEL.

- BPSK: dos símbolos reales, en +/-1. El factor de normalización de potencia es 1 (no hace falta normalizar potencia). Y se transmite un bit por símbolo, es decir, M = 1.
- QPSK: cuatro símbolos complejos en cuadratura. Cada uno de ellos capaz de codificar dos bits, por tanto, M = 2. El factor de normalización de potencia es de raíz de dos que coincide con el módulo del vector que va desde el origen de plano real-complejo hasta el símbolo de la constelación.

#### 1.1.2.3 Generación bits en vector columna

Generamos los bits de datos. Se ha calculado en función de los bits usables para datos, esto es, excluyendo las portadoras reservadas para los pilotos. Por tanto, el número de bits a generar es:

```
NUM_SYMB * Ldata * M
```

Se ha usado la función rand que devuelve un número entre 0 y 1 con misma probabilidad por tanto al usar el comparador de mayor que 0.5 aproximadamente la mitad serán cero y la otra mitad unos.

#### 1.1.2.4 PRBS - Generación de secuencia pseudoaleatoria

Para la generación de la secuencia pseudoaleatoria se ha tenido en cuenta las especificaciones del estándar:

- Inicialización del registro de desplazamiento.
- Generación de un número pseudoaleatorio por cada portadora usada en el símbolo (tanto si son de datos como pilotos).

Con propósitos de simplificación se ha generado el vector pseudoaleatorio directamente quedando guardado en memoria para cuando se requiera su utilización, que será solo en el caso de las portadoras pilotos.

Con un bucle for, una operación XOR del primer y último valor del registro de desplazamiento y un desplazamiento de este registro se ha generado la secuencia binaria de bits pseudoaleatorios que queda guardada en la variable “prbs”.

#### 1.1.2.5 Generación de pilotos

De todos los valores generados en la secuencia pseudoaleatoria solo se usan aquellos que se encuentran en la posición de una portadora piloto. El estándar define un piloto fijo cada 10 portadoras de datos siendo la primera portadora útil un piloto. Por tanto solo usamos los valores prbs correspondientes a una posición piloto:

```
pilots = reshape(prbs, Lsimb, NUM_SYMB);
```

```
pilots = pilots(1:12:1705, :);
```

Luego se configura su potencia, si la secuencia toma un valor de 0 para un piloto se le asigna una amplitud de 4/3, si la secuencia toma un valor de 1 para un piloto se le asigna un valor de -4/3.

```
pilots = (pilots > 0) * (-4/3) + (pilots == 0) * 4/3;
```

Luego se le añade a los pilotos la dimensión compleja (aunque sea cero) por compatibilidad con el resto del programa.

#### 1.1.2.6 Pasamos los bits a símbolos

Con ayuda de la orden reshape generamos una matriz MxN/M, donde cada columna representa un símbolo (por eso la matriz tiene M filas, una por cada bit a codificar en el símbolo).

Luego se le calcula la traspuesta con la orden .' que traspone la matriz sin conjugarla (dado que se está operando con números complejos).

Por último, con ayuda de un for y teniendo en cuenta la estructura con la que se han definido los símbolos se puede obtener el valor de cada símbolo a modular en cada portadora, de ello se encarga el siguiente bloque, el mapeador.

#### 1.1.2.7 Mapeador

Se obtienen los valores complejos a modular en las portadas y se guardan en la variable tx_const_point. Aprovechando la manera en la que esta ordenado el vector C, es símbolo de la constelación es constel:

```
C(i)
```

Donde:

```
i= bitToDecimal(b0,b1) + 1
```

Por último se le añade la dimensión compleja (aunque sea cero) para su representación más adelante.

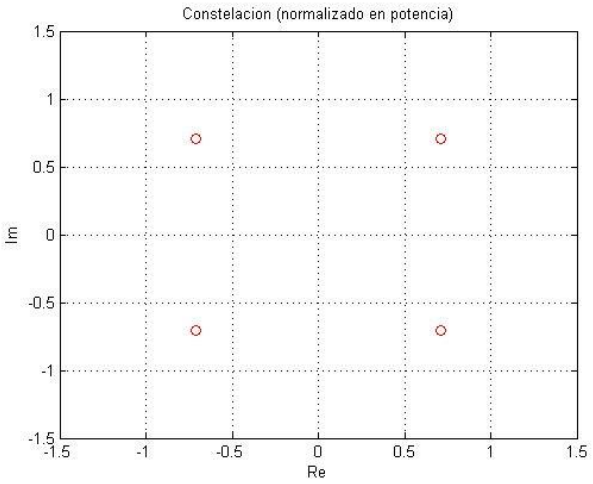
#### 1.1.2.8 Normalización en potencia

Se normaliza para que cada símbolo a transmitir tenga potencia 1. Tenemos en cuenta el caso QPSK y BPSK:

- BPSK - No hace falta, potencia es 1
- QPSK - Normalizamos entre raíz de 2

#### 1.1.2.9 Dibujar los símbolos

Se representa en el plano real - complejo los símbolos de la constelación a transmitir. Esta figura servirá como referencia para comparar con los símbolos recibidos en el receptor y cuantificar cuán buena ha sido la recepción.




#### 1.1.2.10 Inserción de pilotos y datos

Se ha buscado insertar los pilotos sin machacar los datos. Para ello se ha usado una máscara o matriz de indexación, es decir, una matriz de 1 y 0 indicando los lugares pertenecientes a los pilotos y a lo datos.

El procedimiento es:

1. Creamos matriz: [Ldata x NUM_SYMB]

```
simbols = zeros(Lsimb, NUM_SYMB);
```

2. Insertamos los pilotos

```
simbols(1:12:1705,:) = pilots;
```

3. Transformamos datos en columnas, una columna por símbolo

```
data = reshape(tx_constel_data, Ldata, NUM_SYMB);
```

4. Se crea un máscara en el lugar donde no están los pilotos y se usa para indexar las posiciones de los datos:

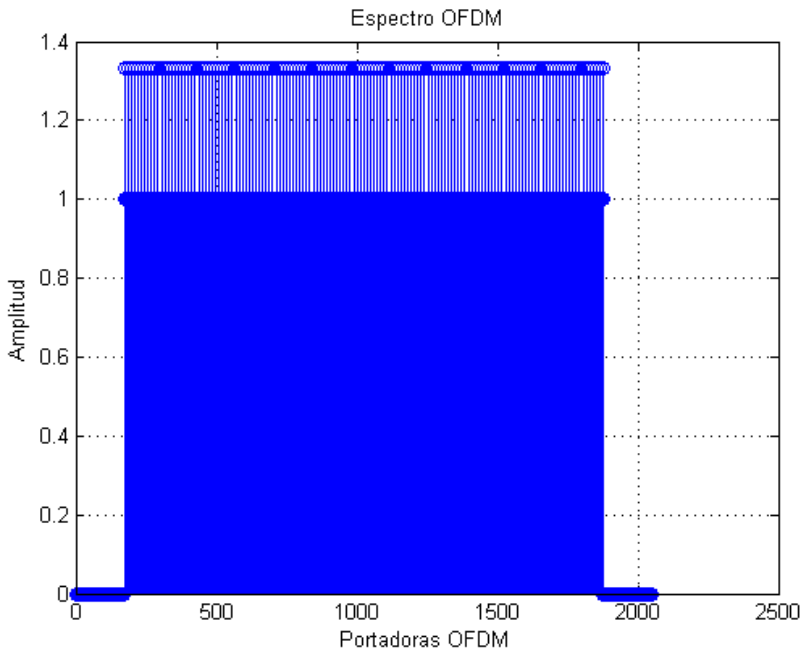
```
mask_vect = ones(Lsimb,1);          % Mascara 1x1705 que tendra 1
mask_vect(1:12:1705) = 0;           % en las posiciones de datos y
data_index = mask_vect == 1;        % 0 en la de los pilotos
simbols(data_index, :) = data;      % Insertando datos
```

#### 1.1.2.11 Creación símbolos OFDM en frecuencia

Se insertan los datos y los pilotos generados en las Lsimb = 1705 portadoras centrales de las 2048 portadoras totales, dejando las portadoras de los extremos a cero. Para ello se usa la orden reshape.

#### 1.1.2.12 Dibujar portadoras

Se dibujan las portadoras para verificar los dos pasos anteriores. De momento no se tiene en cuenta el vector de frecuencias por ello se usa la orden stem.




Se observa como:

- Las portadoras de los extremos están a cero.
- Los pilotos tienen amplitud 4/3 (en valor absoluto).
- Hay una portadora piloto cada 11 portadoras de datos.
- La primera y la última portadora son pilotos.
- Todas las portadoras tienen la misma energía porque la diferencia está en la fase.

#### 1.1.2.13 Modulación OFDM

Para realizar el paso del dominio frecuencial al temporal primero se debe organizar la información del espectro frecuencial para que Matlab lo transforme adecuadamente al dominio temporal. Esto se realiza con la orden ifftshift que reordena el espectro en frecuencias positivas.

Se realiza la modulación a continuación, mediante la orden ifft, que es el algoritmo rápido de la anti transformada de Fourier.

#### 1.1.2.14 Prefijo cíclico

Se ha definido que el prefijo cíclico tiene NCP muestras, en este caso NCP = 32. Por tanto, se han tomado las NCP últimas muestras y se han copiado al principio de la trama de modo que se obtiene una trama de NFFT + NCP elementos.

Para ello se ha usado la concatenación de vectores en Matlab.

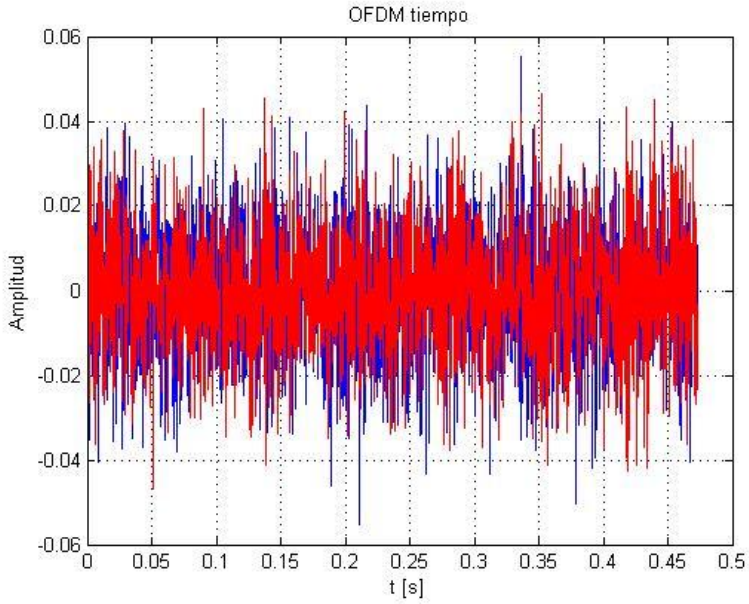
```
[ofdm_time(end-(NCP-1):end, :) ; ofdm_time];
```

#### 1.1.2.15 Configuración de la transmisión

Se configura la transmisión en un vector columna. Esto tiene sentido para cuando NUM_SYMB es mayor que 1. En este caso se haría todo lo descrito anteriormente, pero la trama asociada a cada símbolo OFDM estaría en una columna diferente y se busca enlazarlo todo en un solo vector columna.

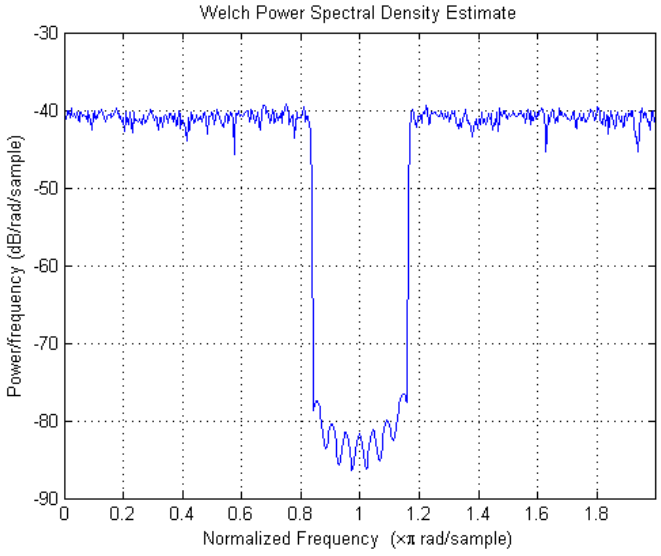
#### 1.1.2.16 Dibujar la transmisión en el tiempo

Se ha superpuesto la parte real e imaginaria. Se observa como tiene una forma aleatoria con algún que otro pico que sobresale de los demás. Esto es uno de los problemas típicos de esta modulación, conocido como el "peak-to-average power ratio" (PAPR).




#### 1.1.2.17 Espectro de frecuencias

Se dibuja el espectro de frecuencias con la orden pwelch del toolbox de procesado de señal de matlab. Se puede observar como el vector de frecuencias está normalizado en pi radianes por muestra.




#### 1.1.2.18 Canal p1 DVBT

Se ha generado el canal p1 definido en la norma acorde a ella. Para ello se han definido las siguientes constantes cuyos valores se han extraído del estándar:

- Rho: potencia
- Tau: retraso o tap
- Theta: fase

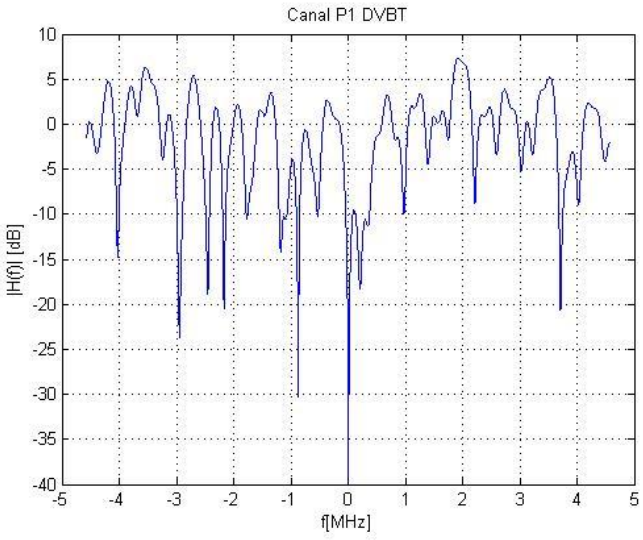
Se ha determinado el cálculo del canal desde su fórmula en el dominio frecuencial para antitrasformarlo luego y obtener su respuesta temporal. Para ello se ha definido primero el vector de frecuencias como sigue:

- Se ha definido el tiempo de muestreo,
- Se ha calculado su inversa para obtener la frecuencia de muestreo.
- Se ha generado las NFFT frecuencias necesarias para la representación del canal.

Luego se ha reservado memoria para el canal generado y se ha resuelto el sumatorio de la fórmula propuesta en el estándar mediante un bucle for.

Tras ello se antitrasforma la respuesta frecuencial mediante la orden ifft y se obtiene la respuesta temporal.

Se ha dibujado a continuación el módulo de la respuesta en frecuencia del canal donde se puede apreciar fading para distintas frecuencias.




#### 1.1.2.19 Ruido

Se desea generar un ruido adaptado en potencia a la SNR dada como parámetro para simular el ruido base de un canal de comunicaciones.

Se ha generado un ruido complejo y se ha normalizado en potencia (por ser complejo entre raíz de dos).

Se ha estimado luego la potencia media de la transmisión para adaptar la potencia del ruido a un nivel de SNR dado. Luego se le suma el ruido a la transmisión.

Una vez generado el ruido AWGN añadimos el fading. Como el desvanecimiento o fading es la respuesta del canal ante la transmisión, para introducir sus efectos en la transmisión se convoluciona la respuesta del canal en tiempo con la transmisión más el ruido.

#### 1.1.2.20 Quitar el prefijo cíclico

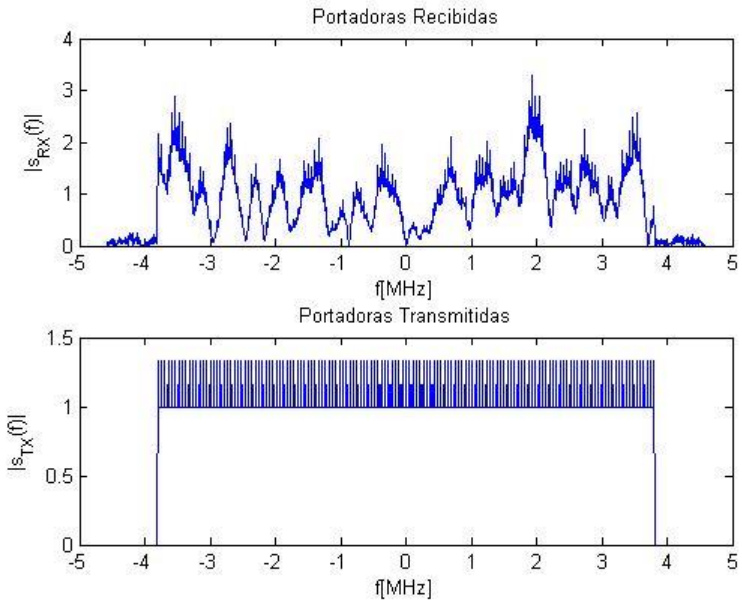
Para quitar el prefijo cíclico transformamos el vector de datos recibidos en una matriz de NCP+NFFT filas y tantas columnas como NUM_SYMB. Las primeras NCP filas corresponden al prefijo cíclico de cada símbolo, por tanto, una vez hecha esta transformación se selecciona la matriz desde NCP+1 filas hasta el final.

Luego se vuelve a transformar en un vector columna quedando así sin prefijo cíclico.

#### 1.1.2.21 Pasar al dominio de la frecuencia

Para pasar al dominio frecuencial se ha computado la transformada de Fourier con la orden fft. Luego se ha transformado el vector para ser representado con frecuencias negativas con la orden fftshift.

Se ha representado a continuación las portadoras recibidas frente a las transmitidas para observar el efecto del fading. Se puede apreciar como la amplitud de las portadoras recibidas están afectadas por el efecto del canal p1. Además, se puede observar la presencia de los pilotos que se usarán más tarde para la estimación del canal.




#### 1.1.2.22 Quitar las portadoras sin información

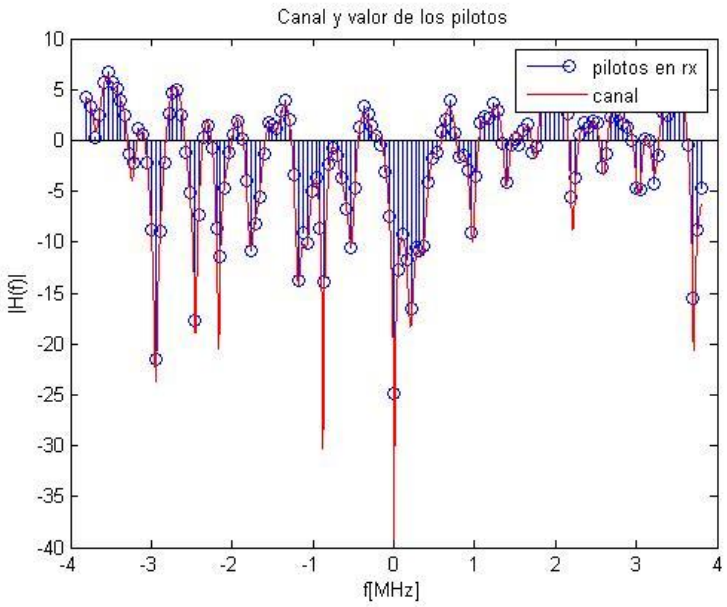
A continuación de las NFFT posibles portadoras del símbolo OFDM se seleccionan las centrales, las Lsimb portadoras útiles (tanto de datos como de pilotos). Obtenemos por tanto una matriz con Lsimb filas y NUM_SYMB columnas con las portadoras correspondientes.

#### 1.1.2.23 Estimador de canal

Para la ecualización de la transmisión se necesita primero estimar el canal por el que se ha transmitido. Para ello se usan los pilotos, de modo que el primer paso es sacar los pilotos de la transmisión y compararlos con los pilotos transmitidos para obtener una estimación de los pilotos recibidos.

Luego se reserva una variable para el canal estimado y se introducen los pilotos que se acaban de estimar.

Se hace también una comprobación de la bondad de la estimación del canal por parte de los pilotos, para ello se ha representado los pilotos estimados del canal contra el canal real.




Se realiza luego la interpolación, para la que se ha definido:

- Intervalo de interpolación: vector con los índices de los datos a interpolar por cada ristra (datos comprendidos entre pilotos), es decir, vector de 1 a 11.
- Índice de la posición inicial de cara ristra de datos a interpolar, es decir, la posición del primer dato después de cada piloto, que coincide con el índice de la posición de los pilotos más 1.

Para cada índice se han interpolado durante el intervalo de interpolación, es decir, para cada índice se ha interpolado los 11 datos de esa ristra y se han introducido en la variable reservada para el canal estimado.

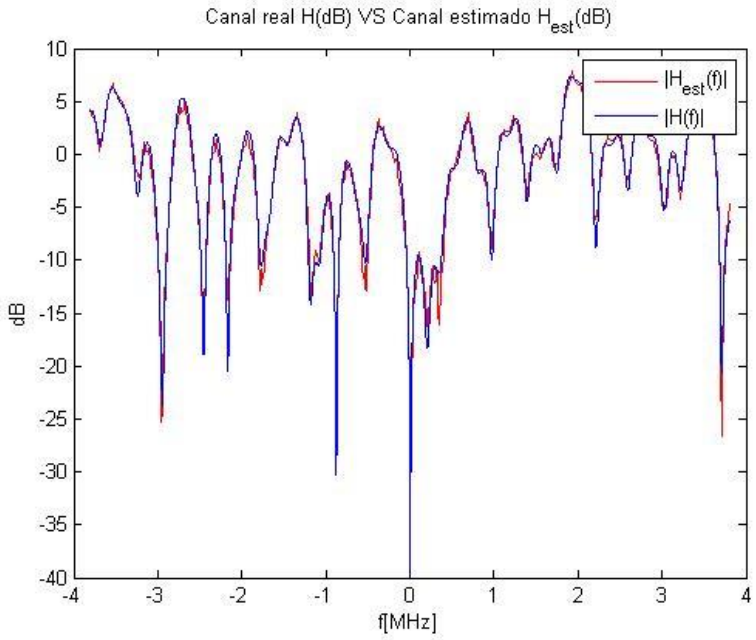
#### 1.1.2.24 Dibujar canal estimado VS canal real

Se ha representado la respuesta del canal estimado en frecuencias negativas, para ello adaptamos el vector de frecuencias entre +/- fs/2 (la frecuencia de muestreo o sampling).

Además, se han obviado las portadoras sin información (para las que no se ha generado respuesta del canal ya que no afectan a la transmisión). Para ellos se ha truncado el vector de frecuencias.

Luego se ha escalado la frecuencia para representar la gráfica en MHz.

Por último, se representa el canal real (en azul), frente al estimado (en rojo).

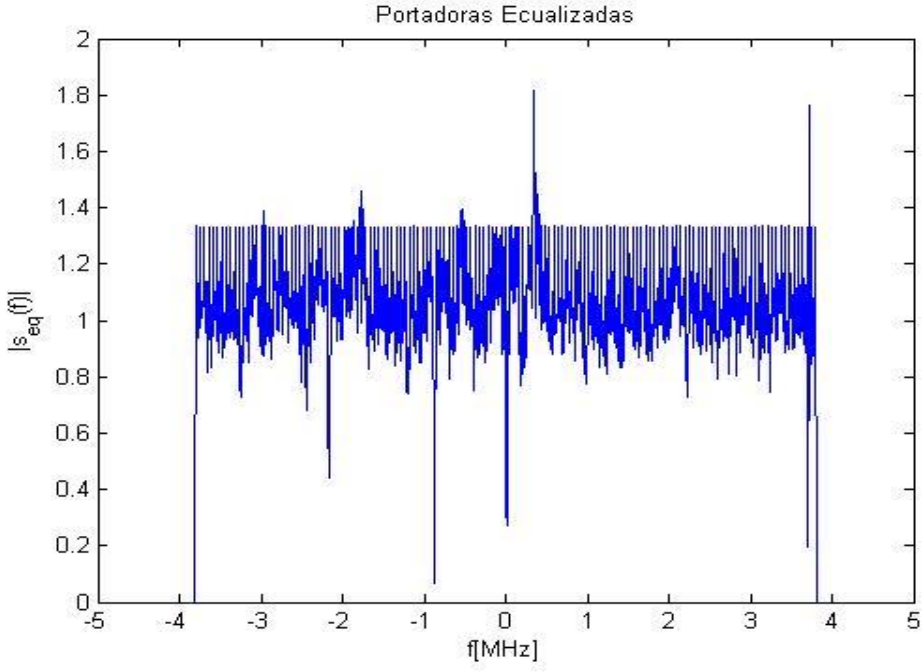



#### 1.1.2.25 Ecualización

La ecualización se realiza mediante una operación de división de los datos recibidos frente al canal estimado.

Para su comprobación se han representado las portadoras ecualizadas. Se puede observar como la respuesta ecualizada no es igual al espectro de la señal transmitida, pero se le parece. Además, se observa como hay lugares donde se ha ecualizado mejor y otros donde se ha ecualizado peor, esto se debe a errores en la estimación del canal.

Se puede observar como para algunas frecuencias la estimación no se ha realizado correctamente y por tanto la ecualización presenta algunos errores (algunas amplitudes muy bajas para algunas frecuencias).



#### 1.1.2.26 Quitar los pilotos

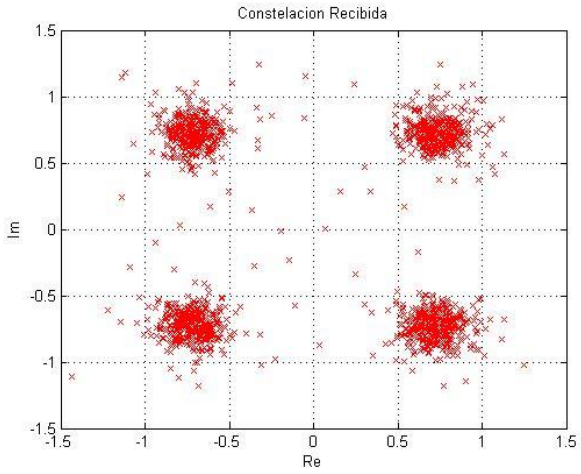
Para quitar los pilotos se sigue la misma estrategia que en su inserción, pero en orden inverso.

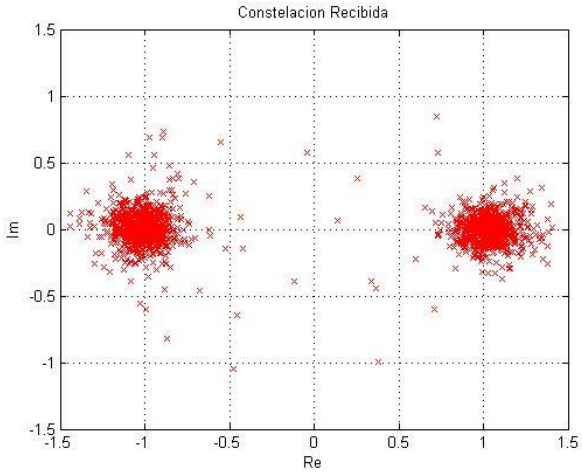
1. Se quita el piloto final.
2. Se transforman los datos recibidos en una matriz de 12 columnas quedando en la primera columna los pilotos y en las 11 siguientes los datos de cada ristra.
3. Se selecciona los datos obviando los pilotos.
4. Se transforma a vector columna de nuevo.

Por último, se renombra la variable de datos recibidos para el demapper.

#### 1.1.2.27 Dibujar los símbolos recibidos

Se representan los símbolos recibidos en la constelación del plano real - imaginario. Se puede observar la distribución de los valores recibidos frente al origen de cada uno de los posibles símbolos.




#### 1.1.2.28 Demapper

En el demapper o decisor decide en función de la constelación empleada y los datos recibidos.

- BPSK
  - Si bit < 0 entonces es un 1
  - Si bit > 1 entonces es un 0
- QPSK
  - Tiene 2 bits (b1 y b0) por símbolo en orden: b1b0
  - Si parte real negativa b0 = 1 si positiva b0 = 0
  - Si parte real negativa b1 = 1 si positiva b1 = 0

Luego se computa la BER mediante una XOR de los bits transmitidos y los bits recibidos.

También se pueden probar los resultados para la constelación BPSK.

# 2 VHDL

## 2.1. Introducción

En esta sección se detallan los distintos bloques VHDL codificados así como se justifican las decisiones de diseño tomadas.

## 2.2. Arquitectura del proyecto

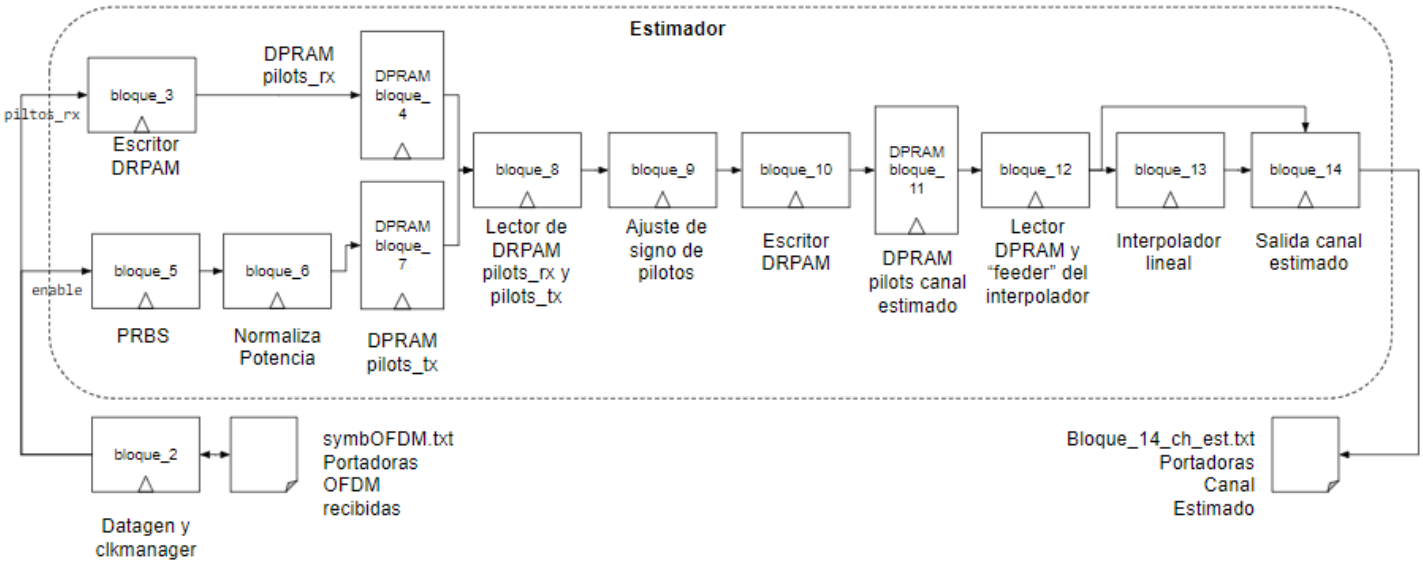
El proyecto se ha realizado por bloques acorde a la siguiente funcionalidad:

- symbOFDM – Portadoras de entrada: Fichero de entrada con los símbolos OFDM recibidos de la simulación de MALAB
- bloque_2 – Datagen y clkmanager: Bloque que lee datos de un fichero y genera el reloj
- bloque_3 – Escritor DPRAM: Bloque para escribir los datos generados por el bloque anterior en una memoria RAM
- bloque_4 – DPRAM portadoras OFDM: DPRAM donde los símbolos ODFDM de las portadoras recibidas se guardan
- bloque_5 – PRBS: Bloque que implementa el PRBS
- bloque_6 – Normalizador en potencia: Bloque que genera pilotos con los resultados del bloque PRBS y los guarda en una DPRAM
- bloque_7 – DPRAM pilotos transmitidos: DPRAM con los pilotos generados por símbolo (están solo en las posiciones 1:12:1705)
- bloque_8 – Lector de DRPAM pilotos recibidos y pilotos transmitidos: Cuando la DPRAM de los pilotos recibidos y la de los pilotos generados del PRBS han terminado de escribirse, el bloque 8 las lee y devuelve los pilot_rx and pilot_tx. Aunque del pilot_tx solo devuevle si es positivo o negativo ya que los pilots_tx solo pueden ser +4/3 o -4/3
- bloque_9 – Ajuste de signo de pilotos: Divide pilot_rx por +/- 4/3 = +/-0.75 dependiendo del signo del pilot_tx generando así los pilotos estimados pilot_est
- bloque_10 – Escritor DRPAM: Guarda los pilotos del canal estimado en una DPRAM antes del interpolador para que luego éste pueda procesarlos adecuadamente
- bloque_11 – DPRAM pilotos de canal estimado: DPRAM para guardar los pilotos ecualizados
- bloque_12 – Lector de DPRAM y escritor en el interpolador: Va recorriendo la DPRAM anterior tomando los pilotos superior e inferior para cada iteración y se lo pasa al interpolador para que vaya

interpolando en orden

- bloque_13 – Interpolador lineal: Interpolador de los pilotos del canal. Devuelve una interpolación lineal de 11 valores entre el piloto superior y el inferior.
- bloque_14 – Salida del canal estimado: Bloque que toma los pilotos y los valores interpolados, los ordena y produce el canal estimado

A continuación, se muestra un esquema de los bloques codificados:




### 2.2.1 Fichero symbOFDM.txt

Fichero que contiene las portadoras recibidas generadas en MATLAB. Estas portadoras se han codificado en punto fijo para una palabra de 12 bits con 4 de parte decimal para la parte real y otros tantos para la imaginaria.

### 2.2.2 Bloque_2 – Datagen y clkmanager

Se compone del clkmanager y el dataread con los que genera la señal de reloj y reset y va leyendo los datos del fichero que expone al siguiente bloque:

```
component bloque_2
port(
    data_b2  : out std_logic_vector(23 downto 0);    -- [Re(23,12), Im(11,0)]
    valid_b2 : out std_logic;                        -- 1 if data_in is ready
    clk_b2   : out std_logic;
    rst_b2   : out std_logic
    );
end bloque_2;
```

### 2.2.3 Bloque_3 – Escritor DPRAM

Toma los datos de entrada del bloque 2 cuando se lo señalizan con valid y los escribe en la DPRAM del bloque 4 usando el write enable.

```
component bloque_3
port(
    data_in_b3   : in std_logic_vector(23 downto 0);    -- [Re(23,12), Im(11,0)]
    valid_in_b3  : in std_logic;                        -- 1 if data_in is ready
    clk_b3       : in std_logic;
    rst_b3       : in std_logic;
    data_out_b3  : out std_logic_vector(23 downto 0);
    addr_out_b3  : out std_logic_vector(10 downto 0);    -- 11b = 2^(11) = 2408
    write_en_b3  : out std_logic;
    write_fin_b3 : out std_logic
    );
end component;
```

### 2.2.4 Bloque_4 – DPRAM portadoras OFDM

DPRAM de 1705 palabras de longitud y 24 bits de anchura de palabra.

```
component bloque_4
port(
      clka  : in  std_logic;
      wea   : in  std_logic_vector(0 downto 0);
      addra : in  std_logic_vector(10 downto 0);
      dina  : in  std_logic_vector(23 downto 0);
      clkb  : in  std_logic;
      addrb : in  std_logic_vector(10 downto 0);
      doutb : out  std_logic_vector(23 downto 0)
      );
end component;
```

### 2.2.5 Bloque_5 – PRBS

Bloque que implementa el PRBS del estandar DVBT. Genera un dato cada vez que se recibe una nueva portadora, esto se señaliza con el valid del bloque 2

```
component bloque_5
port(
      clk    : in std_logic;   -- clock
      rst    : in std_logic;   -- reset
      Yout   : out std_logic;  -- randomized output
      valid  : out std_logic;  -- output signaling
      enable : in std_logic    -- on/off singal
      );
end component;
```

### 2.2.6 Bloque_6 – Normalizador en potencia

Cuando la DPRAM está escrita, entonces lee los datos y transforma el resultado del PRBS en pilotos normalizados en +/- 4/3 y los guarda en la DPRAM del bloque 7

```
component bloque_6
port(
      clk_b6       : in std_logic;
      rst_b6       : in std_logic;
```

```
      prbs_b6      : in std_logic;
      valid_b6     : in std_logic;
      data_out_b6  : out std_logic_vector(23 downto 0);
      addr_out_b6  : out std_logic_vector(10 downto 0);
      write_en_b6  : out std_logic;
      write_fin_b6 : out std_logic
      );
end component;
```

### 2.2.7 Bloque_7 – DPRAM pilotos transmitidos

DPRAM de 143 palabras de longitud y 24 bits por palabra para guardar los pilotos generados por el PRBS y el bloque 6

```
component bloque_7
port(
      clka  : in  std_logic;
      wea   : in  std_logic_vector(0 downto 0);
      addra : in  std_logic_vector(10 downto 0);
      dina  : in  std_logic_vector(23 downto 0);
      clkb  : in  std_logic;
      addrb : in  std_logic_vector(10 downto 0);
      doutb : out  std_logic_vector(23 downto 0)
      );
end component;
```

### 2.2.8 Bloque_8 – Lector de DRPAM pilotos recibidos y pilotos transmitidos

Cuando el PRBS ha terminado de generar los pilotos transmitidos y el bloque 3 de recibir el símbolo OFDM en recepción el bloque 8 lee ambos datos de su DPRAM correspondiente. Conforme va leyendo devuelve los pilotos transmitidos (generados por el PRBS) y los pilotos recibidos (en las posiciones de pilotos del símbolo OFDM).

Como los pilotos transmitidos solo pueden tomar dos valores 4/3 y -4/3, tan solo se señaliza si el piloto transmitido es positivo o negativo, es decir, solo se informa con el primer bit del piloto transmitido.

Nótese como ahora el pilot_rx es un record de un tipo complejo.

```
component bloque_8
port(
      clk         : in std_logic;
      rst         : in std_logic;
      addr_symb   : out std_logic_vector(10 downto 0);
      data_symb   : in std_logic_vector(23 downto 0);
      symb_ready  : in std_logic;
      addr_pilot  : out std_logic_vector(10 downto 0);
      data_pilot  : in std_logic_vector(23 downto 0);
      pilot_ready : in std_logic;
      pilot_rx    : out complex12;
      pilot_tx_signed : out std_logic;
      pilot_txrx_fin : out std_logic;
      valid : out std_logic
      );
end component;
```

### 2.2.9 Bloque_9 – Ajuste de signo de pilotos

Bloque que realiza la ecualización de los pilotos recibidos con respecto a los pilotos transmitidos para hallar los pilotos del canal estimado y así, mediante interpolación, poder estimar el canal.

La ecualización o división se hace teniendo en cuenta que la operación a realizar es:

$$pilot_{est} = \frac{pilot_{rx}}{pilot_{tx}}$$

Pero como $pilot_{tx}$solo toma dos valores determinados:

$$pilot_{tx} = \{4/3\ , -4/3\ \}$$

Tomando esto en cuenta la operación anterior se reduce a dos tipos de operaciones, dependiendo del valor del piloto transmitido:

$$pilot_{est} = \frac{pilot_{rx}}{\{4/3\ , -4/3\ \}} = \ pilot_{rx} * \ \{0.75, -0.75\ \}$$

Así sabiendo si el piloto transmitido es negativo o positivo podemos transformar la operación de ecualización en una multiplicación por dos constantes. Dado que la multiplicación es mas simple de implementar en hardware que la división se ha usado este planteamiento para evitar el uso de una división.

```
component bloque_9
port(
      pilot_signed : in std_logic;
      pilot_rx     : in complex12;
      pilot_est    : out complex12
      );
end component;
```

### 2.2.10 Bloque_10 – Escritor DRPAM

Bloque que guarda los pilotos estimados en una DPRAM previa al interpolador.

```
component bloque_10
port(
      clk         : in std_logic;
      rst         : in std_logic;
      pilot_eq    : in complex12;
      pilot_eq_valid : in std_logic;
      pilot_addr  : out std_logic_vector(7 downto 0);
      pilot_data  : out std_logic_vector(23 downto 0);
      pilot_data_valid : out std_logic;
      pilot_write_fin : out std_logic
      );
end component;
```

### 2.2.11 Bloque_11 – DPRAM pilotos de canal estimado

DPRAM que guarda los pilotos estimados. Tiene una longitud de 143 palabras y 24 bits por palabra.

```
component bloque_11
port(
      clka  : in  std_logic;
```

```
      wea   : in  std_logic_vector(0 downto 0);
      addra : in  std_logic_vector(7 downto 0);
      dina  : in  std_logic_vector(23 downto 0);
      clkb  : in  std_logic;
      addrb : in  std_logic_vector(7downto 0);
      doutb : out  std_logic_vector(23 downto 0)
      );
end component;
```

### 2.2.12 Bloque_12 – Lector de DPRAM y escritor en el interpolador

Toma los pilotos estimados de la DPRAM y se los va insertando al interpolador en orden teniendo en cuenta que el interpolador toma el piloto inferior y superior para hacer la interpolación en cada iteración.

```
component bloque_12
port(
      clk       : in std_logic;
      rst       : in std_logic;
      ram_ready : in std_logic;
      data      : in std_logic_vector(23 downto 0);
      addr      : out std_logic_vector(7 downto 0);
      pilot_inf : out complex12;
      pilot_sup : out complex12;
      valid     : out std_logic;
      interp_ready : in std_logic
);
end component;
```

### 2.2.13 Bloque_13 – Interpolador lineal

Interpolador lineal de la práctica de la asignatura. Toma dos datos en el formato complejo definido en el paquete como un record e interpola linealmente, tanto para la parte real como para la imaginaria, entre el valor inferior y el superior.

El objetivo del bloque del interpolador es devolver 11 valores resultados de una interpolación lineal entre los pilotos inferior y superior de un canal de comunicaciones para su estimación. Además, hay que tener en cuenta que los pilotos son complejos y pueden ser positivos o negativos. En resumen, busca resolver la siguiente expresión:

$$estim(n) = pilot_{sup} \cdot \frac{n}{12} + pilot_{inf} \cdot \frac{12-n}{12} \qquad con\ n = 1,2,\dots 11$$

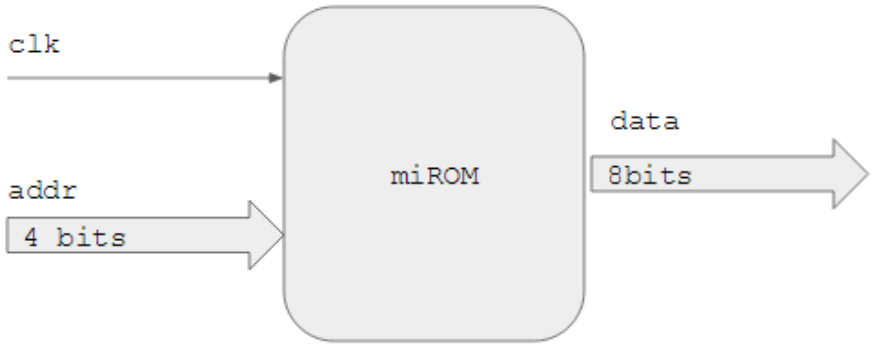
Por especificación, se ha determinado que los valores de estim los vaya devolviendo uno a uno durante un periodo determinado.

#### 2.2.13.1 Memoria ROM codificada

Ante la imposibilidad del uso del IP core generator en una de las versiones con las que se ha trabajado, en primera instancia, se ha buscado en la literatura sobre VHDL un código de ejemplo de una memoria ROM que se ha adaptado convenientemente para su uso en el proyecto. Esta memoria ROM es exclusiva del interpolador siendo las demás DPRAM generadas con el IP core generator.

La memoria ROM se define como un bloque que dada una dirección de entrada devuelve el valor del contenido de la dirección requerida en el siguiente flanco de reloj. Por tanto, es una memoria ROM síncrona.

Se han definido los siguientes puertos:




Se ha definido un tipo llamado ROM_16 declarado un array de vectores de 8 bits. Se ha declarado que ROM_16 sea un array de elementos de 8 bits porque es la longitud de los datos a almacenar.

```
type ROM_16 is array (0 to 15) of std_logic_vector(7 downto 0);
```

Se define una constante llamada memoria del tipo ROM_16 y se inicializa a los valores deseados. El razonamiento de porque se almacenan estos valores y no otros se detalla mas adelante en la descripción del interpolador que usa este bloque dentro de su arquitectura interna.

Se define un único proceso, el proceso síncrono, que dada una dirección devuelve el contenido guardado en la memoria para esa dirección en cada flanco de subida del reloj. Esto se realiza con la orden:

```
data <= memoria(to_integer(unsigned(addr)))
```

#### 2.2.13.1.1 Codificación con 3 bits

La primera prueba se ha realizado con 3 bits. Para la codificación de los coeficientes se asume que los tres bits son de la parte decimal del número a codificar. Se obtienen las siguientes características.

| Bit | 0 | 1 | 2 |
|---|---|---|---|
| Pesos | 1/2 | 1/4 | 1/6 |

| | **Decimal** | **Hexadecimal** | **Binario** |
|---|---|---|---|
| Valor máximo codificable | 0.8750 | 7 | 111 |
| Valor mínimo de paso codificable | 0.1250 | 1 | 001 |
| Error medio de punto fijo | 0.030303 | | |

Con estos parámetros se han obtenido los siguientes resultados:

```
C_sup       C_sup_fp      Hex         C_inf       C_inf_fp      Hex
0.0833      0.1250        1           0.9167      0.8750        7
```

```
0.1667      0.1250       1           0.8333      0.8750       7
0.2500      0.2500       2           0.7500      0.7500       6
0.3333      0.3750       3           0.6667      0.6250       5
0.4167      0.3750       3           0.5833      0.6250       5
0.5000      0.5000       4           0.5000      0.5000       4
0.5833      0.6250       5           0.4167      0.3750       3
0.6667      0.6250       5           0.3333      0.3750       3
0.7500      0.7500       6           0.2500      0.2500       2
0.8333      0.8750       7           0.1667      0.1250       1
0.9167      0.8750       7           0.0833      0.1250       1
Error medio de punto fijo: 0.030303
```

Donde el error medio se ha calculado como la media del valor absoluto de la diferencia entre los coeficientes en coma flotante y en punto fijo, esto en matlab queda como:

```
mean(abs(coef_sup - coef_sup_fp.data));
```

#### 2.2.13.1.2 Codificación con 4 bits

Se ha realizado el mismo cálculo para una codificación con 4 bits. Se obtienen las siguientes características.

| Bit | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Pesos | 1/2 | 1/4 | 1/6 | 1/16 |

| | **Decimal** | **Hexadecimal** | **Binario** |
|---|---|---|---|
| Valor máximo codificable | 0.9375 | f | 1111 |
| Valor mínimo de paso codificable | 0.0625 | 1 | 0001 |
| Error medio de punto fijo | 0.015152 | | |

Se han realizado los cálculos para estas características y se han obtenido los siguientes resultados.

```
C_sup       C_sup_fp     Hex         C_inf       C_inf_fp          Hex
0.0833      0.0625       1           0.9167      0.9375            f
0.1667      0.1875       3           0.8333      0.8125            d
0.2500      0.2500       4           0.7500      0.7500            c
0.3333      0.3125       5           0.6667      0.6875            b
0.4167      0.4375       7           0.5833      0.5625            9
0.5000      0.5000       8           0.5000      0.5000            8
0.5833      0.5625       9           0.4167      0.4375            7
0.6667      0.6875       b           0.3333      0.3125            5
0.7500      0.7500       c           0.2500      0.2500            4
0.8333      0.8125       d           0.1667      0.1875            3
0.9167      0.9375       f           0.0833      0.0625            1
Error medio de punto fijo: 0.015152
```

Se ha determinado que el error medio con 4 bits es aceptable en comparación con el error para 3 bits y se ha determinado por tanto usar 4 bits para la codificación de los coeficientes que se almacenan en la memoria ROM codificada.

Los coeficientes se almacenan con la siguiente estructura: 4 primeros bits, coeficiente superior, 4 últimos bits, coeficiente inferior.

#### 2.2.13.2 Características del interpolador

El interpolador implementado tiene las siguientes características:

- Latencia = 3. Tarda tres ciclos de reloj en entregar el primer dato.
- Rendimiento (troughtput) = 1/2. Entrega un dato nuevo cada dos ciclos de reloj.

#### 2.2.13.3 Entidad

El bloque del interpolador viene dado por los puertos:

- clk: puerto de entrada del reloj.
- rst: puerto de entrada de la señal del reset.
- sup: puerto de entrada para el piloto superior de la interpolación. Es de tipo complex12.
- inf: puerto de entrada para el piloto inferior de la interpolación. Es de tipo complex12.
- valid: puerto de entrada que indica que las señales de entrada tienen datos válidos para ser procesados.
- estim: puerto de salida con el resultado de una de la interpolación (en un instante determinado). Es de tipo complex12.
- estim_valid: puerto de salida que indica que la señal de salida tiene un dato válido para ser procesado.

#### 2.2.13.4 Arquitectura

##### 2.2.13.4.1 Señales y componentes

Se han definido varias señales:

- estado: señal tipo FSM_estado que determina el estado instantáneo de la máquina de estados que rige el comportamiento del interpolador. Se ha declarado también la señal p_estado (próximo estado) para asegurar el sincronismo.
- saddr: señal dirección de 4 bits para la memoria ROM. Se ha declarado también la señal p_saddr para asegurar el sincronismo.
- sdata: señal de 8 bits que recoge los datos de salida de la ROM.
- ssup: señal compleja de 12 bits que guarda el valor del piloto superior para la interpolación. Se ha declarado también la señal p_ssup (próximo estado) para asegurar el sincronismo.
- sinf: señal compleja de 12 bits que guarda el valor del piloto inferior para la interpolación. Se ha declarado también la señal p_sinf (próximo estado) para asegurar el sincronismo.
- p_estim: señal compleja de salida que guarda los datos que transferirá al puerto de salida estime, antes de la señal de sincronismo,
- sesitm: señal que toma el mismo valor que el puerto de salida estim, se podría haber usado el propio puerto de salida, pero como por especificación no puede ser un puerto inout se ha decidido declarar esta señal auxiliar.

Además, se ha definido una memoria ROM, descrita en el fichero miROM.vhd como componente del interpolador. Esta memoria guardará los coeficientes de la interpolación. Los puertos de esta memoria estarán

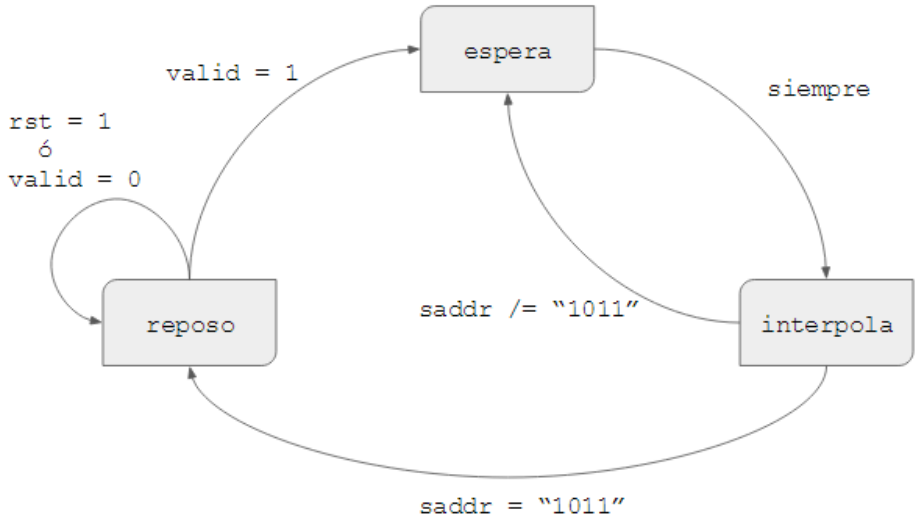
conectados las siguientes señales definidas como sigue (puerto > señal):

```
clk => clk
addr => saddr
data => sdata
```

La descripción de la arquitectura de esta memoria ya ha sido realizada en otro apartado anterior.

#### 2.2.13.4.2 Proceso combinacional: La máquina de estados

El funcionamiento del interpolador se rige por una máquina de estados cuyos estados quedan implementados en una variable del tipo FSM_estado declarada en mi_paquete.vhd. La estructura de la máquina viene dada en la siguiente figura.




Los estados realizan las siguientes funciones:

- Reposo: es el estado por defecto, la máquina se encontrará en reposo mientras no haya estímulos válidos en los puertos de entrada.
    - Si valid es 1 se guardan los puertos de entradas en las señales "p_xxxx" correspondiente a cada puerto. Se guardan en las señales "próxima señal" para que al siguiente ciclo de reloj estén en las señales correspondientes de forma sincronizada. Además, se configura el siguiente estado como espera.
    - Si valid no es 1 la máquina se queda como está, las señales "próxima señal" se actualizan con el valor actual de las señales originales asegurando así que nada cambie. La señal p_estado se actualiza con el estado de reposo, no se ha actualizado a si misma porque es posible que se acceda al estado de reposo desde un reset y la variable estado no esté configurada.
    - Se configura la señal estim_valid a cero porque en este estado aún no hay datos procesados en la salida.
    - Se actualiza el valor de p_estim, como la señal estim.
- Espera: es un estado que espera un ciclo de reloj. Este ciclo de reloj es justo el ciclo que tarda la memoria ROM desde que se le pone una dirección en su puerto de entrada hasta que saca el dato en su puerto de salida. Además, como este es un estado de transición al que también se llega después del estado interpola, se configura la señal estim_valid a 1, ya que tras la interpolación realizada en el estado

interpola los datos están listos para ser entregados. Esto se cumple excepto para la primera transición a este estado, es decir, a la transición reposo – espera.

- Se configura interpola como el próximo estado, esta condición se cumple siempre y la variable estado se actualizará en el próximo ciclo de reloj.
- El resto de señales se auto asignan a sus señales "próxima señal" para asegurar que no cambien.
- Sino es la primera transición (es decir, si venimos del estado interpola) configuramos estim_valid a 1. Esta condición se cumple cuando la señal de dirección de la ROM es distinta a la dirección inicial "0000", es decir, sino estamos esperando el primer coeficiente.

- Interpola: cuando se llega a este estado tenemos datos válidos y listos para ser procesados en las señales de piloto superior, inferior y coeficientes (ssup, sinf y coef respectivamente). Este estado se encarga de interpolar (con la función descrita en mi_paquete.vhd), de bajar la bandera estim_valid (ya que se están procesando los datos y no estarán listos hasta el siguiente ciclo de reloj) y de transicionar al estado de reposo en caso de que se hayan interpolado los 11 valores correspondientes a los 11 coeficientes almacenados en la ROM.
  - Se guarda el resultado de la interpolación en la señal de próximo estado de la señal estim.
  - Si el valor interpolado era el último, el número 11 se pasa al estado de reposo. Esto se codifica con la condición de que la señal de dirección del coeficiente utilizado sea "1011".
  - Si el valor interpolado no es el último se aumenta una dirección en la variable de próximo estado de dirección de la ROM (para obtener el próximo coeficiente), y se pasa al estado espera (para que le de tiempo a la ROM a sacar el coeficiente).
- Others: en caso de que ante la presencia de una anomalía la señal estado tome un valor fuera de los asignados se configura todo a cero y se configura el próximo estado como estado de reposo.

#### 2.2.13.4.3 Proceso síncrono

El proceso síncrono se encarga de:

- Actualizar el valor de las señales en cada ciclo de reloj con el valor de las señales próxima señal ("p_xxxx") configurado en cada uno de los estados de la máquina de estados.
- Configurar los valores iniciales cuando se da un reset.

### 2.2.13.5 Codificación de coeficientes del interpolador

Los coeficientes de la interpolación vienen dados por la fórmula:

$$Coef_{sup} = \frac{n}{12} \qquad Coef_{inf} = \frac{12-n}{12} \qquad con\ n = 1{,}2 \dots 11$$

Como en el proyecto se ha convenido trabajar en punto fijo se ha realizado una aproximación de estos coeficientes a punto fijo de forma que se de un compromiso entre el número de bits usados para la codificación de estos coeficientes y el error cometido. Para ello se ha usado matlab con el toolbox de punto fijo. Los resultados expuestos a continuación se han sacado de diferentes realizaciones del script coef_calc_fixed_point.m que se adjunta junto con el proyecto.

### 2.2.13.6 Representación de los datos

Dado que se ha tomado la decisión de implementar los coeficientes con 4 bits se adopta por tanto la siguiente estructura de palabra para los pilotos.

| Bits | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Pesos | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 1/2 | 1/4 | 1/8 | 1/16 |

Con esta estructura en punto fijo tenemos las siguientes características.

| | **Decimal** | **Hexadecimal** | **Binario** |
|---|---|---|---|
| Valor máximo codificable | 255.9375 | fff | 111111111111 |
| Valor mínimo de paso codificable | 0.0625 | 001 | 000000000001 |

Dado el valor de la amplitud de los pilotos recibidos en la simulación del receptor OFDM se observa que esta codificación es mas que suficiente para su representación.

### 2.2.14 Bloque_14 – Salida del canal estimado

Canal que ordena adecuadamente los datos que salen del interpolador (los valores interpolados) y los pilotos superior e inferior de cada tramo. De este modo, los valores que saca el estimador entendido como bloque a nivel de sistema, son efectivamente, los valores de las portadoras del canal estimado.

## 2.3. Paquete de funciones

Se ha definido un paquete VHDL que almacena los tipos de datos y funciones utilizadas en diferentes partes de la realización del proyecto. Este paquete se encuentra definido en el fichero mi_paquete.vhd que se usará dentro de la librería por defecto work.

### 2.5.1 Tipos de datos

Se han definido varios tipos de datos, tipos de datos complejos de distinta longitud y un tipo de dato con una serie de estados definido para su uso en una máquina de estados.

#### 2.5.1.1 Tipos de datos complejos

Los números complejos se han implementado como un tipo de dato con el mismo número de bits para la parte real como para la imaginaria.

Con la orden type y record se han definido varios tipos de datos con la siguiente sintaxis:

```
complexXX
```

Donde XX determina el número de bits de la parte real e imaginaria para ese tipo.

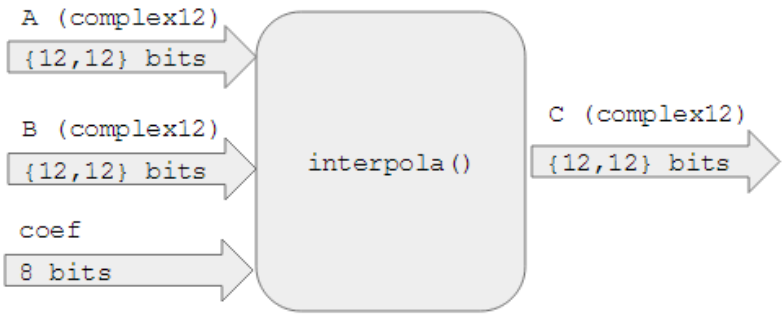
#### 2.5.1.2 Tipo de estados de FSM

Por simplicidad en el código y en su depuración se ha definido un tipo de dato que codifica los distintos estados de una FSM que se usará más adelante. Se han definido 3 estados: reposo, espera e interpola.

### 2.5.2 Función interpolar

Se ha declarado una única función que empaqueta la implementación del proceso combinacional que realiza una

operación de interpolación entre dos valores para unos coeficientes dados. Esta función se puede entender con la siguiente figura.




Las variables implicadas son:

- A: valor inferior de la interpolación.
- B: valor superior de la interpolación.
- coef: coeficientes para el valor superior (4 primeros bits) y el inferior (4 últimos).
- C: valor interpolado.

Un ejemplo de su uso sería:

```
C <= interpolar(A, B, coef);
```

La definición de la función viene dada por la arquitectura que se extrae de la siguiente expresión matemática:

$$C = B \cdot coef_{sup} + A \cdot coef_{inf}$$

Donde el coeficiente que multiplica a B son los 4 primeros bits de coef y el coeficiente que multiplica a A son los 4 últimos bits de coef.

De esta expresión se extrae que para cada componente de A y de B (son imaginarios) hay que multiplicar y sumar. Además, hay que tener en cuenta que las longitudes son diferentes y que los coeficientes son positivos pero que los parámetros A y B no tienen por qué serlo. Se ha procedido en distintos pasos.

#### 2.5.2.1 La multiplicación

Se busca multiplicar cada componente de A y B por sus coeficientes correspondientes. Para ello se ha procedido con la siguiente orden:

```
smultA.re := std_logic_vector((signed('0'&coefA) * signed(A.re)));
```

Donde:

- Se ha realizado la misma operación para cada componente de A y B por separado (A.re, A.im, B.re y B.im).
- Se ha realizado la operación con la orden signed() ya que los valores A y B pueden ser negativos y para evitar que los coeficientes se interpreten como valores negativos (en complemento a 2 si empiezan por 1 se considera negativo)
- Se ha concatenado un bit configurado a 0 delante de los coeficientes de modo que la función signed() entienda los coeficientes como positivos.

- Las variables smultA y smultB almacenan los resultados de las operaciones de cada componente de A y B respectivamente.
- La variable smult es compleja y tiene 17 bits que resultan de la suma de los bits de coef mas los bits de una de las componentes de A o B mas el bit concatenado para evitar que los coeficientes se tomen como negativos.

#### 2.5.2.2 La suma

Se han sumado los resultados de la multiplicación como sigue:

```
ssum.re := std_logic_vector(signed(smultA.re) + signed(smultB.re));
```

Donde:

- Se han sumado las componentes de A y B por separado (dos operaciones, una para la parte real y otra para la imaginaria).
- La variable ssum es compleja de 17 bits por lo expuesto anteriormente, se desprecia el bit de acarreo.

#### 2.5.2.3 El truncado

Se ha realizado un truncado como sigue:

```
C.re := ssum.re(15 downto 4);
```

Donde:

- Se ha truncado tanto para la parte real como para la imaginaria.
- Se han truncado los 4 últimos bits resultados de la operación de multiplicación por ser los menos significativos y tener una salida limitada a 12 bits para cada componente.
- Se ha truncado el primer bit resultado de la concatenación del 0 al principio de los coeficientes para la multiplicación.
- La variable C es la variable que se devuelve con el return.

## 2.4. Posibles mejoras

Tras la realización del proyecto se han detectado ciertas mejoras de optimización que se podrían haber aplicado:

- El bloque_8 que lee los pilotos recibidos y el transmitido y se los pasa al bloque 9 para que le ajuste el signo, podría haber tomado como entrada directamente el bloque 5, PRBS, de modo que se evitaría el escritor en el DPRAM y la DRAM en sí (bloques 6 y 7)
- Existen varios bloques redundantes, como los escritores o lectores de DPRAM

# 3 VERIFICACIÓN CRUZADA Y SIMULACIONES

## 3.1. Introducción

En este apartado se exponen el proceso de verificación cruzada así como las simulaciones realizadas. Los ficheros implicados en este apartado son:

```
.
├── documentos
├── MATLAB
├── verificacion
│   ├── verificacion.m
│   ├── ficheros_MATLAB
│   │   ├── gen_data_tx_dvbt_mult_symb.m
│   │   ├── H_est_wkspc.mat
│   │   ├── matlab_H_est.txt
│   │   ├── matlab_pilots_est.txt
│   │   ├── matlab_prbs.txt
│   │   ├── matlab_rx_pilots.txt
│   │   ├── matlab_symbOFDM.txt
│   │   ├── matlab_tx_pilots.txt
│   │   └── pilots_est_wkspc.mat
│   └── ficheros_VHDL
│       ├── bloque_14_ch_est.txt
│       ├── bloque_2_rx_symb.txt
│       ├── bloque_3_rx_symb.txt
│       ├── bloque_5_prbs.txt
│       ├── bloque_8_rx_pilots.txt
│       ├── bloque_8_tx_pilots.txt
│       └── bloque_9_pilots_est.txt
└── VHDL
    ├── estimador_verification.vhd
    └── wave_conf_files
        └── estimador_verification_all_signals.wcfg
```

## 3.2. Verificación Cruzada

La verificación cruzada se ha realizado en dos bloques. La verificación del funcionamiento de los bloques como entidades propias con la ayuda del framework de cálculo de punto fijo de MATLAB y la verificación del funcionamiento a nivel de sistema del estimador.

El proceso de verificación es como sigue:

- Simular los datos en MATLAB
- Simular los datos en VHDL
- Comparar los datos bloque a nivel de bloque
- Comparar los datos a nivel de sistema

### 3.1.1 Simulación MATLAB

Bajo la ruta de `./verifiacacion/ficheros_MATLAB/` se encuentran los siguientes ficheros:

- gen_data_tx_dvbt_mult_symb.m: implementación sin gráficas que genera datos en MATLAB, los pasa punto fijo y los escribe en los ficheros de a continuación:
    - H_est_wkspc.mat: workspace de ayuda para el `./verificacion/verifiacion.m`
    - matlab_H_est.txt: canal estimado por la simluación MATLAB
    - matlab_pilots_est.txt: pilotos estimados por la simluación MATLAB
    - matlab_prbs.txt: valores del PRBS sacados por la simluación MATLAB
    - matlab_rx_pilots.txt: pilotos recibidos por la simluación MATLAB
    - matlab_symbOFDM.txt: simbolos OFDM de entrada de la simluación MATLAB
    - matlab_tx_pilots.txt: pilotos transmitidos por la simluación MATLAB

Para obtener los datos de la simulación de MATLAB tan solo hay que:

- Ejecutar `gen_data_tx_dvbt_mult_symb.m` para obtener los valores simulados por MATLAB

En este fichero se ha hecho uso del framework de punto fijo de MATLAB para que los resultados obtenidos coincidan con los arrojados por el VHDL.

#### 3.1.1.1 Obtención pilotos canal estimado

A continuación se ofrece un ejemplo del uso del framework de punto fijo de MATLAB para la obtención de los pilotos del canal estimado y escribirlos en un fichero para la verificación cruzada.

```
tx_pilots_inv = 1./tx_pilots;

rxpfi_re = fi(real(rx_pilots), 1, 12, 4);           % Parte Real
txpfi_inv_re = fi(real(tx_pilots_inv), 1, 12, 4);   % Parte Imag
rxpfi_im = fi(imag(rx_pilots), 1, 12, 4);           % Parte Real
txpfi_inv_im = fi(imag(tx_pilots_inv), 1, 12, 4);   % Parte Imag

estpfi_re = rxpfi_re.* txpfi_inv_re;    % Multiplicacion Parte Real
estpfi_im = rxpfi_im.* txpfi_inv_re;    % Multiplicacion Parte Imag

q = fixed.Quantizer(1,12,4);     % Cuantizador de punto fijo

estpfi_q_re = quantize(q, estpfi_re);
estpfi_q_im = quantize(q, estpfi_im);

% Volcando pilotos rx a fichero
% Fichero a volcar la info
file_name = 'matlab_pilots_est.txt';
```

```
    fileID = fopen(file_name,'w');

    for k = 1:length(estpfi_re)
        re_fi = estpfi_q_re(k);
        im_fi = estpfi_q_im(k);
        % Volcando en hexadecimal
        fprintf(fileID,'%s%s\n', re_fi.hex, im_fi.hex);

    end
```

### 3.1.2 Simulación VHDL

Una vez obtenidos los valores objetivos simulados a nivel de sistema y de bloque se busca comprobar si la implementación se ajusta a lo simulado. Para ello se ejecuta el fichero `./VHDL/estimador_verification.vhd`

Se puede usar los ficheros incluidos en la siguiente ruta para visualizar mejor las señales

```
./VHDL/wave_conf_files/estimador_verification_all_signals.wcfg
```

Tras 56 microsegundos de ejecución del simulador se generan los siguientes ficheros de texto bajo la ruta de `./verificacion/ficheros_VHDL/`:

- bloque_2_rx_symb.txt: simbolos generados por el datagen
- bloque_3_rx_symb.txt: simbolos recibidos por el datagen que se escriben en la DPRAM
- bloque_5_prbs.txt: valores del PRBS
- bloque_8_rx_pilots.txt: pilotods recibidos leídos de la DPRAM
- bloque_8_tx_pilots.txt: pilotos transmitidos leídos de la DPRAM
- bloque_9_pilots_est.txt: pilotos del canal estimados calculados en el bloque 9
- bloque_14_ch_est.txt: canal estimado calculado tras el interpolador

Estos ficheros están codificados con 6 caracteres hexadecimal, 3 para la parte real, seguidos de otros 3 de parte imaginaria, al igual que los ficheros generados por la simulación de MATLAB. Ahora tan solo queda compararlos.

### 3.1.3 Verificación por bloques

Para la verificación por bloques no hay mas que comparar los ficheros correspondientes a un bloque generado por MATLAB con su fichero correspondiente en VHDL. Las relaciones son las que siguen.

| MATLAB | VHDL |
|---|---|
| matlab_symbOFDM.txt | bloque_2_rx_symb.txt |
| matlab_symbOFDM.txt | bloque_3_rx_symb.txt |
| matlab_prbs.txt | bloque_5_prbs.txt |

| | |
|---|---|
| matlab_rx_pilots.txt | bloque_8_rx_pilots.txt |
| matlab_tx_pilots.txt | bloque_8_tx_pilots.txt |
| matlab_pilots_est.txt | bloque_9_pilots_est.txt |
| matlab_H_est.txt | bloque_14_ch_est.txt |

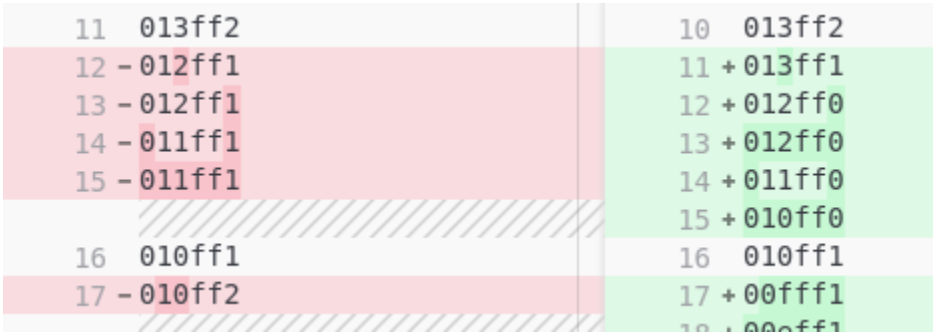
Las comparaciones se deben hacer con `diff -i` (ignore case) ya que MATLAB escribe los caracteres hexadecimales en minúsculas y VHDL en mayúsculas. También se puede hacer uso de herramientas gráficas para mayor visibilidad.

Para comparar los resultados, desde una terminal Linux, tan solo habría que ir hasta la ruta `./verificacion/` y ejecutar:

```
diff -i ficheros_MATLAB/matlab_symbOFDM.txt   ficheros_VHDL/bloque_2_rx_symb.txt
diff -i ficheros_MATLAB/matlab_symbOFDM.txt   ficheros_VHDL/bloque_3_rx_symb.txt
diff -i ficheros_MATLAB/matlab_prbs.txt       ficheros_VHDL/bloque_5_prbs.txt
diff -i ficheros_MATLAB/matlab_rx_pilots.txt  ficheros_VHDL/bloque_8_rx_pilots.txt
diff -i ficheros_MATLAB/matlab_tx_pilots.txt  ficheros_VHDL/bloque_8_tx_pilots.txt
diff -i ficheros_MATLAB/matlab_pilots_est.txt ficheros_VHDL/bloque_9_pilots_est.txt
diff -i ficheros_MATLAB/matlab_H_est.txt      ficheros_VHDL/bloque_14_ch_est.txt
```

De la ejecución de ello se observa cómo todos los ficheros son iguales (diff no arroja nada a la terminal) exceptuando la última comparación. La última comparación es entre el canal estimado por MATLAB y el canal estimado por VHDL y tan solo difiere en un bit de diferencia en la parte real y en la imaginaria codificadas como: 3 caracteres hexadecimal de parte real y 3 caracteres hexadecimal de parte imaginaria.

Se ha hecho uso de una herramienta de comparación gráfica para ilustrar estas diferencias encontradas:




Por otra parte, la implementación de los distintos bloques con el framework de punto fijo se pueden encontrar en el fichero de generación de los datos de simulación de MATLAB `gen_data_tx_dvbt_mult_symb.m`

Si se desea verificar algún bloque en particular se puede acceder a una mejor visualización usando los ficheros de `./VHDL/estimador_verification_bXX.wcfg` donde XX es el número del bloque.

### 3.1.4 Verficación a nivel de sistema

Tras haber ejecutado previamente tanto el fichero de generación de datos en MATLAB como haber simulado el estimador VHDL en Xilinx : `./verifiacion/ficheros_MATLAB/gen_data_tx_dvbt_mult_symb.m` y `./VHDL/estimador_verification.vhd`

Se ha preparado un fichero que ayuda a comparar los resultados gráficamente, así como da una intuición del

error en términos absolutos y cuadráticos. El fichero se encuentra en:

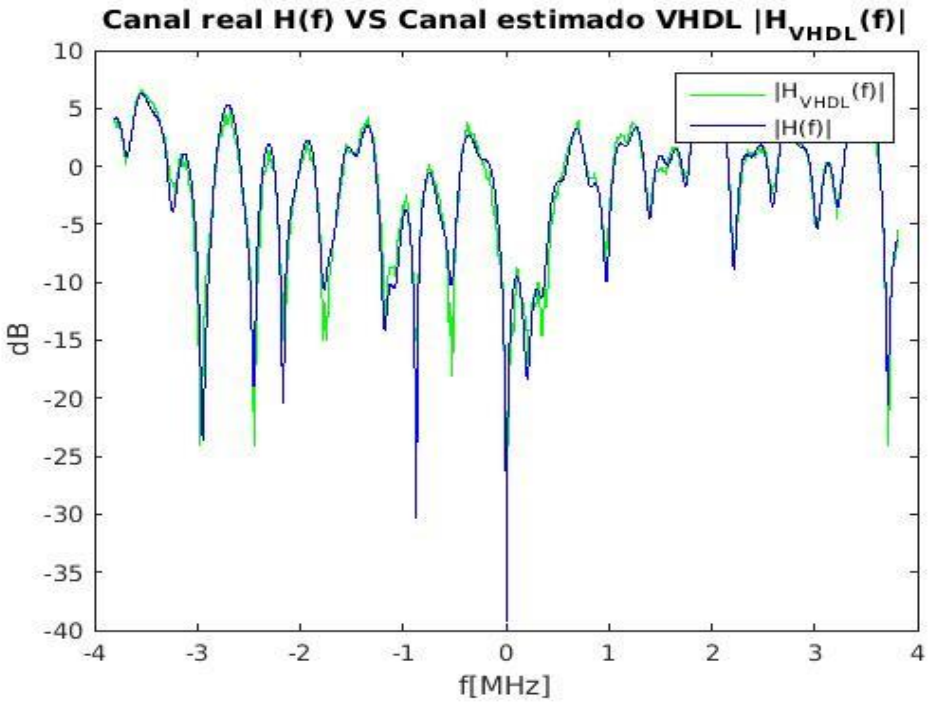
```
./verificacion/verificacion.m
```

Las comparaciones se establecen entre:

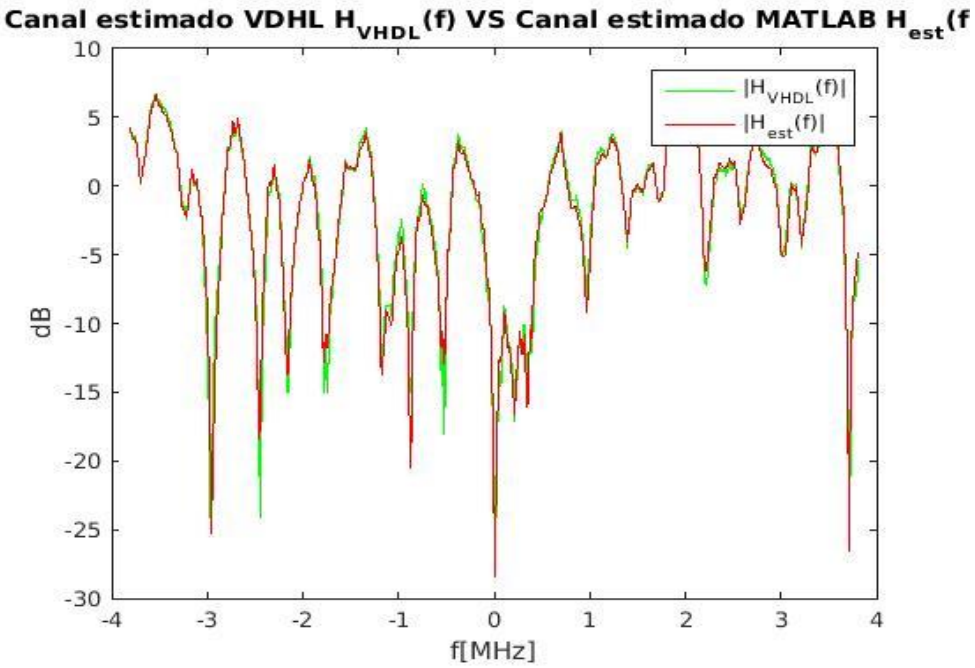
- Canal real
- Canal estimado MATLAB
- Canal estimado VHDL

Tal y como se muestra a continuación.

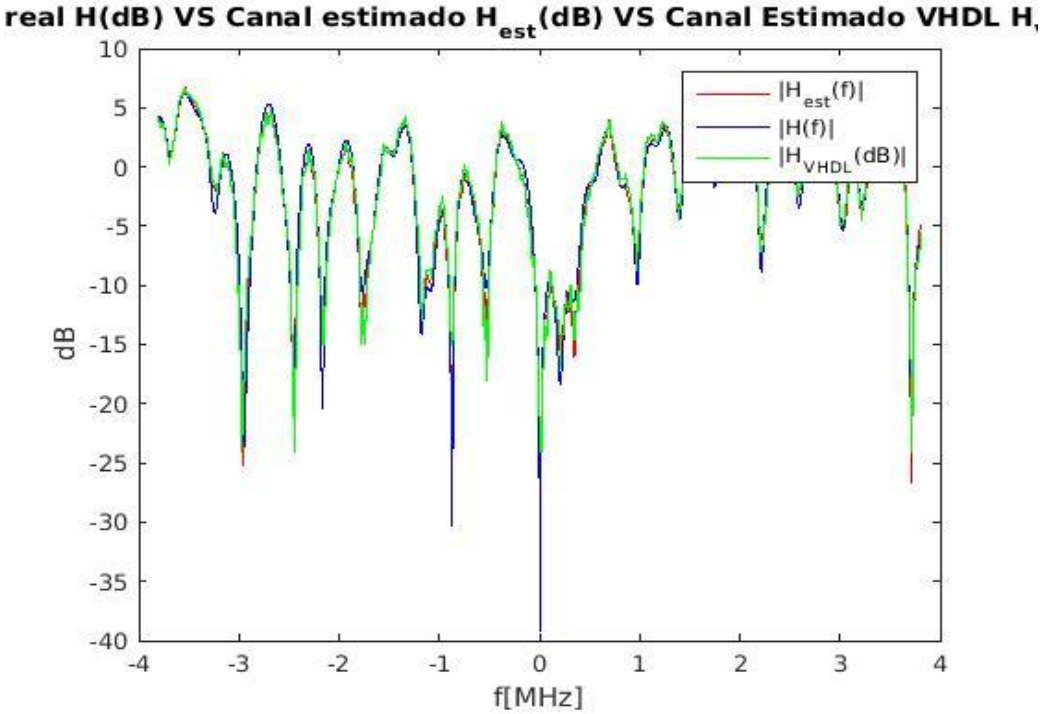
#### 3.1.4.1 Comparación canal estimado VHDL con canal estimado real



### 3.1.4.2 Comparación canal estimado VHDL y canal estimado MATLAB




### 3.1.4.3 Comparación canal estimado VHDL, canal estimado MATLAB y canal real




### 3.1.4.4 Comparación de errores

Finalmente se ha calculado los errores cuadráticos medio entre los tres canales expuestos, así como el error absoluto de la parte real e imaginaria.

Tras la ejecución del fichero, en la línea de comando de MATLAB se obtiene:

```
Error Cuadratico Medio
MSE(H_est_vhdl, H_est_matlab)  = 0.002520
MSE(H_est_vhdl, H_real)        = 0.008145
MSE(H_est_matlab, H_real)      = 0.005032

Error Absoluto parte real e imaginaria
Error(H_real, H_vhdl)  :  real = 0.070396,  imag = 0.077267
Error(H_real, H_matlab):  real = 0.053706,  imag = 0.052728
Error(H_vhdl, H_matlab):  real = 0.047017,  imag = 0.044873
```

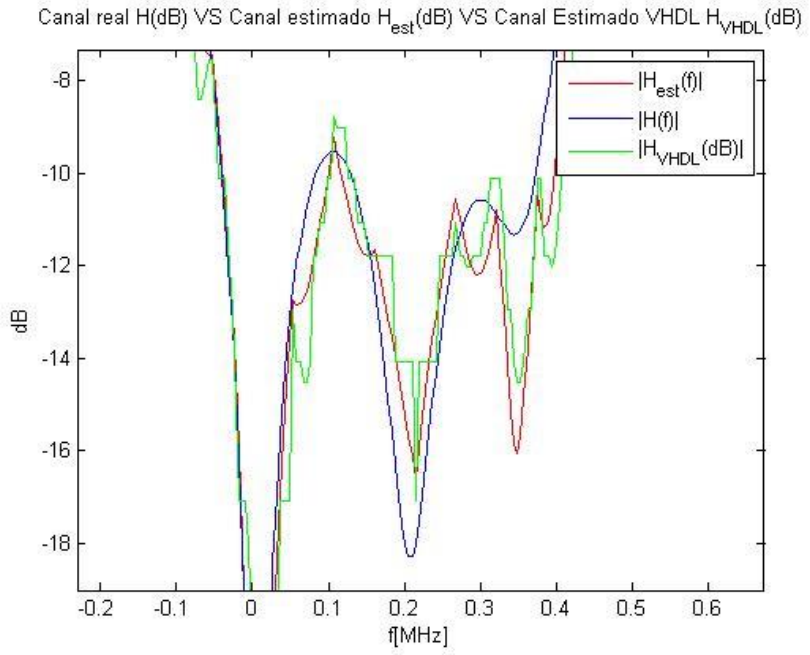
Se observa que:

- El error de la estimación lineal de punto flotante en MATLAB es menor que el de VHDL
- El error entre el canal estimado MATLAB y VHDL es menor que cualquiera de ellos con el canal real ofreciendo así una intuición de su similitud
- El error absoluto entre VHDL y el canal real se diferencia en un poco mas del valor máximo de resolución de la estructura elegida en punto fijo. Se han elegido 4 bits de parte decimal en punto fijo:

$$\frac{1}{2^4} = 0.0625 \approx 0.07$$

#### 3.1.4.5 Otros errores

Se ha notado que en la comparación entre el canal estimado de MATLAB y el canal estimado real hay ciertos errores un tanto extraños. En ciertas partes, la estimación del canal no es buena o no se ajusta tanto como debería a la del canal estimado de MATLAB.




En principio se pensó en que esos cambios tan bruscos en la ecualización se debían a que era el salto mínimo de cuantización (el bit menos significativo) sin embargo se ha observado como para otros trozos de la curva el

ajuste es bueno (como para el segundo valle de la figura).

Se han comparado los ficheros de salida del estimador VHDL y el estimador MATLAB y comparado sus diferencias.

```
019FFA - 019ffa
018FF9 - 019ff9
017FF8 - 018ff9
017FF7 - 017ff8
016FF6 - 017ff7
015FF5 - 016ff6
015FF5 - 016ff5
015FF4 - 015ff5
014FF3 - 014ff4
013FF2 - 014ff3
013FF1 - 013ff2
012FF0 - 012ff1
```

Por lo tanto se sospecha que habiendo una diferencia tan pequeña el error puede estar en la conversión de estos valores a punto flotante en MATLAB para su representación.

Debido a la falta de tiempo no se ha podido profundizar mas en este error.

## 3.3. Simulaciones

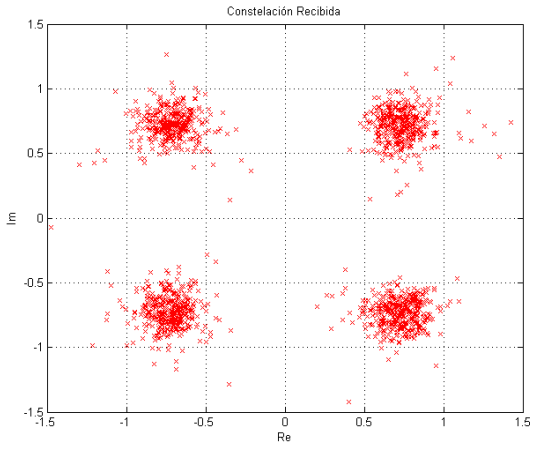
Durante la realización del código que implementa el transmisor, el canal y el receptor OFDM se han realizado multitud de pruebas para asegurar que los resultados tienen sentido y sean correctos.

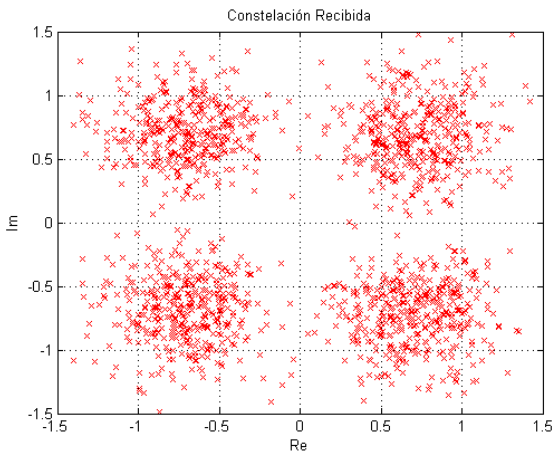
### 3.1.5 El ruido

Se ha probado la influencia del ruido en el sistema. Se ha probado su influencia sin considerar el fading y variando la SNR.

| SNR (dB) | 20 | 10 | 9 |
|---|---|---|---|
| BER | 0.00000 | 0.000640 | 0.003201 |

La constelación recibida para 20 (izquierda) y 10 (derecha).



Se comprueba como el valor de la SNR esta fuertemente ligado a la correcta recepción en presencia de ruido y ausencia de fadding. Tan solo con aumentar la SNR bajamos la BER exponencialmente.

### 3.1.6 El fading

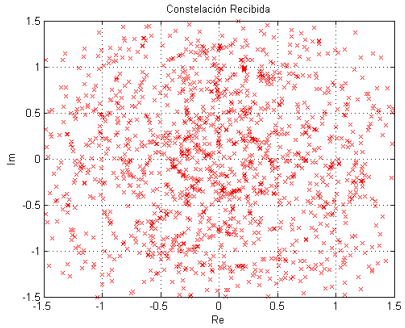
Se ha probado la influencia del fading obviando el ruido AWGN (se ha eliminado para analizar tan solo la influencia del fading) para distintos valores de SNR y se ha probado la necesidad del uso de un estimador de canal.

#### 3.1.6.1 Fading sin estimador de canal

Se ha comentado todo el estimador de canal y se han obtenido los siguientes resultados.

| | |
|---|---|
| SNR (dB) | 20 |
| BER | 0.532010 |

Se observa como para la misma SNR que antes se analizaba obtenemos un resultado desastroso. Esto queda confirmado por la constelación recibida.



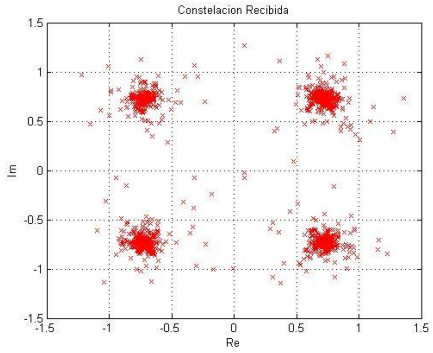

Nota: solo se ha obtenido un resultado porque el fading es fijo (solo se ha generado un canal) y se ha quitado la influencia del ruido AWGN, por tanto, aunque al aumentar la SNR debería disminuir la BER (pero de forma muy poco apreciable) no disminuye y por tanto solo se ha probado para un valor de SNR.

#### 3.1.6.2 Fading con estimador de canal

Si añadimos el bloque del estimador del canal obtenemos.

| | |
|---|---|
| SNR (dB) | 20 |
| BER | 0.004161 |

Se obtienen una constelación con algún valor disperso, pero en general centrada en el origen del símbolo.




### 3.1.7 Ruido y fading

Con ruido y fading ambos fenómenos se superponen con los siguientes resultados que son peores que en caso del ruido solo (como cabría esperar).

| SNR (dB) | 20 | 13 | 10 |
|---|---|---|---|
| BER | 0.004161 | 0. 004802 | 0.003201 |

### 3.1.8 Constelación BPSK

La constelación BPSK tiene la mitad de eficiencia espectral que la QPSK ya que por cada símbolo solo transmitimos un bit cuando para la QPSK se transmiten dos. Sin embargo esta constelación presenta mayor robustez frente al ruido.

A continuación, se muestran los distintos resultados arrojados por el simulador de las características de esta constelación frente al ruido y al fading para distintas SNR.

| SNR (dB) | 20 | 13 | 8 |
|---|---|---|---|
| BER | 0. 003201 | 0. 003841 | 0. 004481 |

Se infiere entonces que esta constelación es mas fuerte frente al ruido. Constelación recibida para 10dB (izquierda) y 20dB (derecha).