

Lecture 13, Multirate Signal Processing

MDCT

Last time we saw that **para-unitary** polyphase matrices are an easy way to obtain the inverse polyphase matrix for the synthesis (we just need to transpose the matrix and replace z by z^{-1}). But how do we obtain useful para-unitary polyphase matrices or filter banks? (more useful than a simple transform matrix)

We now go one step further in **increasing the length** of our polynomials in the polyphase matrix. We saw that zero'th order polynomials result in our usual transform matrices, like a DCT. But we already know their frequency responses are often not good enough. The next step is to look at polynomials of **first order**, where we have elements z^{-1} . This then leads to impulse response with a length of **2 blocks** (the first block corresponds to a delay of z^0 , the second block to z^{-1}). Hence we get the **impulse response length of $L=2N$** (with N the block length and the number of subbands). One wide-spread example are the so-called **MDCT** (Modified Discrete Cosine Transform) filter banks, which are so-called **cosine modulated filter banks**. As we saw (Lecture 6), modulation means the multiplication of a baseband prototype filter impulse response

with a periodic **modulation** function, here a cosine function. In this way, all the subband filters are obtained from one prototype filter $h(n)$, in this case for the **analysis filters**, as

$$h_k(n) = h(n) \cdot \sqrt{\frac{2}{N}} \cdot \cos\left(\frac{\pi}{N}\left(k + \frac{1}{2}\right)\left(n + \frac{N}{2} + \frac{1}{2}\right)\right) \quad (1)$$

for the subbands $k=0, \dots, N-1$, and time index $n=0, \dots, 2N-1$, meaning we have filters of length **$L=2N$** .

The prototype filter $h(n)$ is a low pass filter, and the $\cos()$ modulation function shifts the center frequency of the filter to the cosine function frequencies $(\pi/N \cdot (k+0.5))$, such that we evenly **cover the entire frequency range from 0 to π** . Imagine the lowpass having a pass band from $\frac{-\pi}{2N}$ to $\frac{\pi}{2N}$, then the first subband, for $k=0$, already results from modulation with frequency $\frac{\pi}{2N}$, and hence it goes on the positive side from 0 to π/N . In this way we obtain N filter of passband width π/N , which then cover the entire frequency range between 0 and π in the positive frequency range (the negative frequencies are the mirrored version).

Observe that we can view eq. (1), the

multiplication of $h(n)$ with the cos term, also as a window design method. Here the ideal filter would not be the sinc function, but an infinitely long cosine function. This corresponds to infinitely narrow bandpass filters at the cosine frequencies as ideal filters. After windowing they then become wider.

Because of those 2 different views, we see 2 different names in literature for $h(n)$:

- "**baseband prototype**"

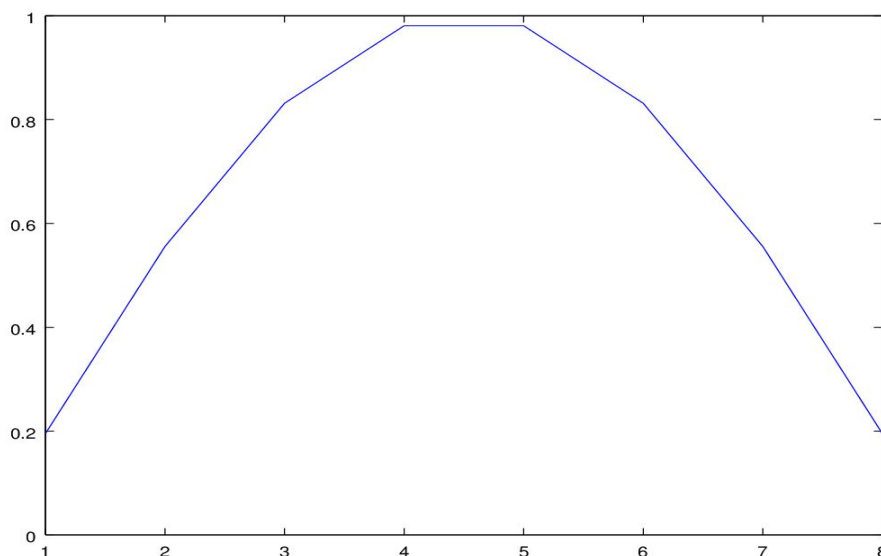
- "**window function**" (time-reversed baseband prototype).

Matlab/Octave Example Filters

Take the so-called **sine window** or **baseband prototype** for $N=4$ subbands,

```
 $h = \sin(\pi/8 * ((0:7) + 0.5));$ 
```

```
plot(h)
```

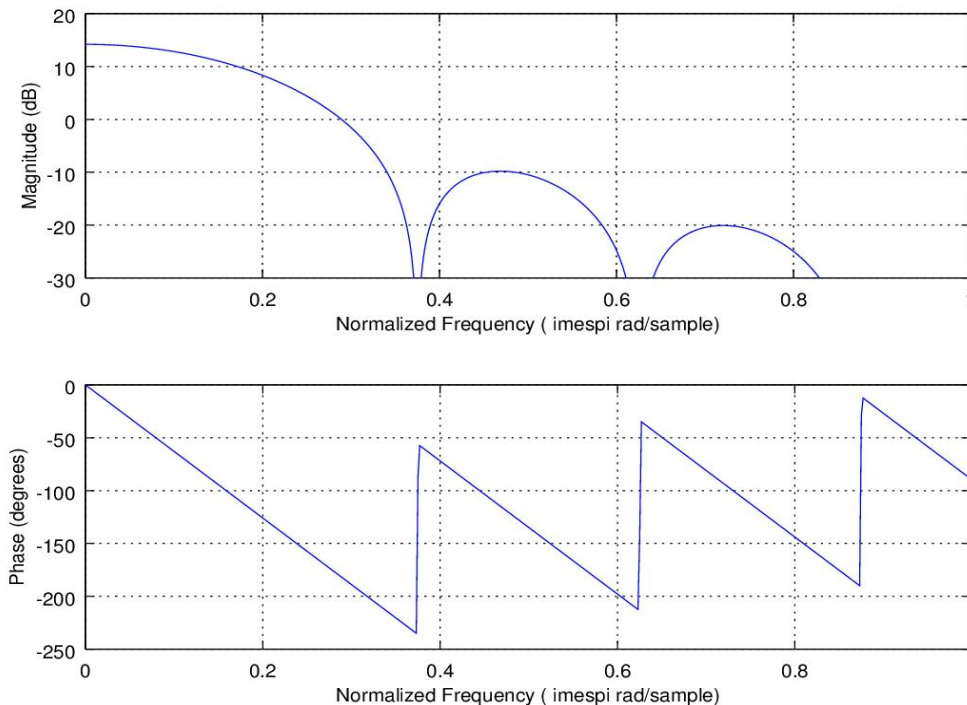


The corresponding frequency response is

```

freqz(h)
subplot(2,1,1)
axis([0 1 -30 20])

```



We see that this is indeed a low pass filter. It works as both, a window and a low pass prototype filter.

Now we construct the filter impulse response for subband $k=2$,

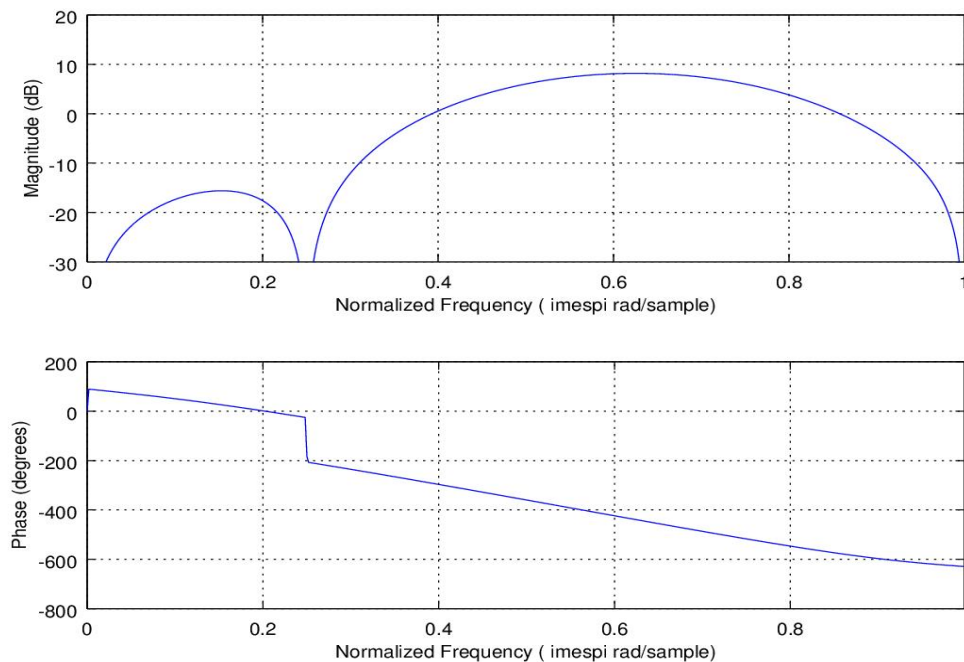
```
h2=h.* cos(pi/4*((0:7)+0.5+4/2)*(2.5));
```

and we get the following frequency response,

```
freqz(h2)
```

```
subplot(2,1,1)
```

```
axis([0 1 -30 20])
```



We see that our low pass filter is indeed shifted in frequency to become a bandpass filter.

So the baseband prototype or window $h(n)$ is all what we need for the design of our filter bank.

$h(n)$ is a lowpass, which we want to design now with the **goal** that we get **high stopband attenuation** and also obtain **perfect reconstruction**.

For the **synthesis filters**, we have the baseband prototype filter $g(n)$ which produces the subband filters,

$$g_k(n) = g(n) \cdot \sqrt{\frac{2}{N}} \cdot \cos\left(\frac{\pi}{N}\left(k + \frac{1}{2}\right)\left(n - \frac{N}{2} + \frac{1}{2}\right)\right)$$

How do we now design the prototype filters $h(n)$ and $g(n)$ such that we obtain **perfect reconstruction**? At this point it is not clear if it even works with the modulation constraint we just introduced. To see that, we first construct the polyphase matrices. First the analysis, we get its elements as (Lecture 12, eq. (2))

$$\begin{aligned} H_{n,k}(z) &= \sum_{m=0}^{L/N-1} h_k(mN+n) \cdot z^{-m} = \\ &= h_k(n) + h_k(N+n) \cdot z^{-1} \end{aligned}$$

Observe that the upper limit of our summation index was $L/N-1$, hence it is 1 here since $L=2N$ (we have 2 blocks, block 0 and block 1). This means we only have 2 summands, for $m=0$ and $m=1$.

For the synthesis polyphase matrix we get,

$$\begin{aligned} G_{n,k}(z) &= \sum_{m=0}^{L/N-1} g_k(mN+n) \cdot z^{-m} = \\ &= g_k(n) + g_k(N+n) \cdot z^{-1} \end{aligned}$$

In this way we obtain the analysis polyphase matrix as

$$\begin{aligned}
 H_{MDCT}(z) &= \begin{bmatrix} H_{N-1,0}(z) & H_{N-1,1}(z) & \dots & H_{N-1,N-1}(z) \\ H_{N-2,0}(z) & \dots & \dots & \vdots \\ \vdots & \dots & \ddots & \vdots \\ H_{0,0}(z) & \dots & \dots & H_{0,N-1}(z) \end{bmatrix} = \\
 &= \begin{bmatrix} h_0(N-1)+h_0(2N-1)\cdot z^{-1} & \dots & \dots & h_{N-1}(N-1)+h_{N-1}(2N-1)\cdot z^{-1} \\ h_0(N-2)+h_0(2N-2)\cdot z^{-1} & \dots & \dots & \vdots \\ \vdots & \dots & \ddots & \vdots \\ h_0(0)+h_0(N)\cdot z^{-1} & \dots & \dots & h_{N-1}(0)+h_{N-1}(N)\cdot z^{-1} \end{bmatrix} \\
 (2)
 \end{aligned}$$

Observe that we here only have polynomials of first order as elements of our polyphase matrix. Similar for the synthesis, we get,

$$\mathbf{G}_{MDCT}(z) = \begin{bmatrix} g_0(0)+g_0(N)\cdot z^{-1} & \dots & \dots & g_0(N-1)+g_0(2N-1)\cdot z^{-1} \\ g_1(0)+g_1(N)\cdot z^{-1} & \dots & \dots & \vdots \\ \vdots & \dots & \ddots & \vdots \\ g_{N-1}(0)+g_{N-1}(N)\cdot z^{-1} & \dots & \dots & g_{N-1}(N-1)+g_{N-1}(2N-1)\cdot z^{-1} \end{bmatrix}$$

Observe the corresponding time/phase indices for the analysis and synthesis polyphase matrix, which run the transpose way for the synthesis.

The problem is that we still don't know how to proceed with the design of our prototype filters. We basically could start constructing an

analysis polyphase matrix, and then **invert** it to obtain **perfect reconstruction**. But the inverse of a polynomial matrix is not so easy to compute, and also the inverse polyphase matrix might have filters which are not "good" (for instance no real passband or not sufficient stopband attenuation, or we get IIR filters instead of FIR filters).

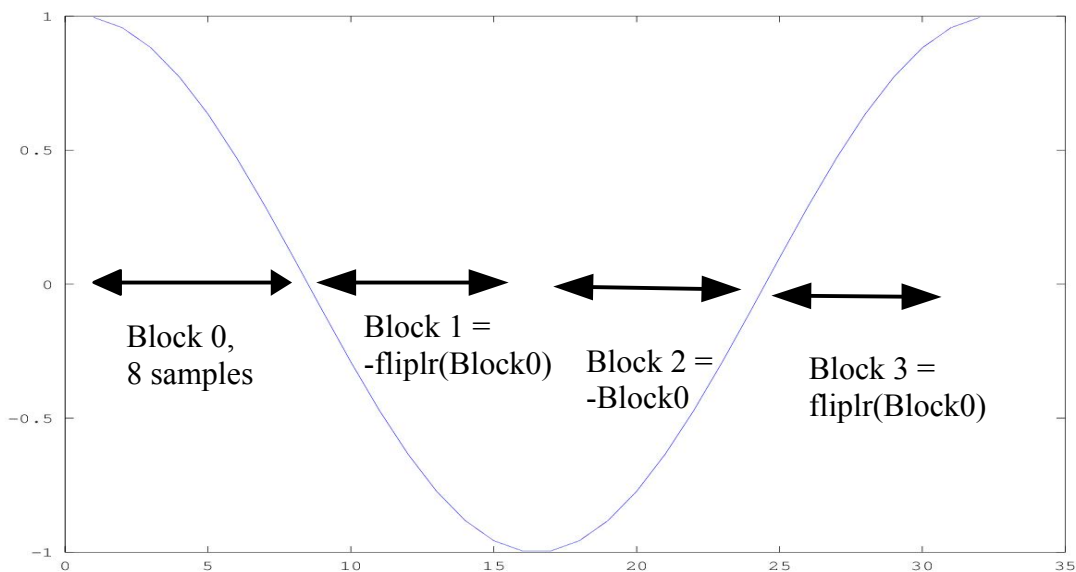
But here, a trick helps. Since we have a modulated filter bank, we have certain periodicities hidden in the impulse responses and hence on the polyphase matrix. It is possible to investigate those periodicities manually, and come up with a solution for perfect reconstruction, as Princen and Bradley did in the late 80's, who first described the MDCT, which they called **TDAC** then (Time Domain Aliasing Cancellation), because that was the approach they used. They analyzed all the aliasing components in the time domain and found one condition to cancel them in the time domain. This lead to the **sine window**. Later people, especially in the MPEG context, called it MDCT.

Matlab/Octave example for the **symmetries** of the cosine modulation function:
Take a **modulated analysis filter bank** with impulse responses

$$h_k(n) = h(n) \cdot \cos\left(\frac{\pi}{8} \cdot (k+0.5) \cdot (n+0.5)\right)$$

with a cosine modulation function like a DCT4. For subband $k=0$, for $N=8$ subbands, and for a length of 32, we get:

```
c=cos(pi/8*0.5*((0:31)+0.5))
plot(c)
```



This image visualizes the following symmetries.

```
c(1:8)
```

```
ans =
Columns 1 through 7:
0.995185 0.956940 0.881921 0.773010 0.634393 0.471397 0.290285
Column 8:
0.098017
```

The second block is identical to flipping and negating the first block:

```
c(9:16)
```

```
ans =
Columns 1 through 7:
-0.098017 -0.290285 -0.471397 -0.634393 -0.773010 -0.881921 -0.956940
Column 8:
-0.995185
```

```
-fliplr(c(1:8))
```

```
ans =
Columns 1 through 7:
-0.098017 -0.290285 -0.471397 -0.634393 -0.773010 -0.881921 -0.956940
```

```
Column 8:  
-0.995185
```

The third block is identical to negating the first block:

```
c(17:24)
```

```
ans =  
Columns 1 through 7:  
-0.995185 -0.956940 -0.881921 -0.773010 -0.634393 -0.471397 -0.290285  
Column 8:  
-0.098017
```

The last block is identical to flipping the first block:

```
c(25:32)
```

```
ans =  
Columns 1 through 7:  
0.098017 0.290285 0.471397 0.634393 0.773010 0.881921 0.956940  
Column 8:  
0.995185
```

And so on. We see the following rule: **Every second block is flipped, and after 2 blocks we get a sign change.**

This is not only true for the first subband, but also for all other subbands k .

To show how this is useful for a matrix implementation, the following **example**:

We have a prototype function $h(n)$ and $N=4$ subbands.

Then our modulated impulse response for subband $k=0$ with the basic DCT4 cosine modulation function is

$$h_0(n) = h(n) \cdot \cos\left(\frac{\pi}{4} \cdot 0.5 \cdot (n+0.5)\right)$$

(to show the basic principle, the time shift of

the MDCT was omitted). Now we can implement it with blocks of size 4. For a filter length of 8 (twice as long as the DCT4) we have 2 modulated blocks (with block number 0 and 1), with time reversal for the analysis, for h_0 :

$$h_0(0) := [h_0(3), h_0(2), h_0(1), h_0(0)] ,$$

$$h_0(1) := [h_0(7), h_0(6), h_0(5), h_0(4)]$$

Now we can write the modulation with length $2N$ with our cosine function with the help of a matrix formulation, which includes the flipping and sign change of the second cosine modulation block. For that we also define the first block of our cos modulation function as

$$\mathbf{T}_0 := \text{fliplr}([\cos(\frac{\pi}{4} \cdot 0.5 \cdot ((0:3)+0.5))])$$

Observe that \mathbf{T}_0^T is also the **first column** of our **DCT4** transform matrix.

Hence, for the first block of the modulated impulse response we simply get:

$$\begin{aligned} h_0(0) &= \begin{bmatrix} h(3) & 0 & .. & . \\ 0 & h(2) & .. & . \\ . & . & h(1) & . \\ . & . & . & h(0) \end{bmatrix} \cdot \mathbf{T}_0^T \\ &= [h(n) \cdot \cos(\frac{\pi}{4} \cdot 0.5 \cdot (n+0.5))]_{n=0, \dots, 3} \end{aligned}$$

Because the second modulation block (block 1) is time-flipped and sign changed compared to the first (block 0), we obtain our second

modulated block as. Instead of flipping and changing the sign of our modulation vector T_0 , we apply the flipping and sign change to the matrix of the prototype function,

$$h_0(1) = \begin{bmatrix} \cdot & 0 & \cdot & -h(7) \\ 0 & \cdot & -h(6) & \cdot \\ \cdot & -h(5) & \cdot & \cdot \\ -h(4) & \cdot & \cdot & \cdot \end{bmatrix} \cdot T_0^T$$

$$= [-h(4) \cdot \cos(\frac{\pi}{4} \cdot 0.5 \cdot (3+0.5)), -h(5) \cdot \cos(\frac{\pi}{4} \cdot 0.5 \cdot (2+0.5)), \dots]$$

Hence we can put both together in the z-domain as

$$h_0(0) + z^{-1} h_0(1) = \begin{bmatrix} h(3) & 0 & \cdot & -h(7)z^{-1} \\ 0 & h(2) & -h(6)z^{-1} & \cdot \\ \cdot & -h(5)z^{-1} & h(1) & \cdot \\ -h(4)z^{-1} & \cdot & \cdot & h(0) \end{bmatrix} \cdot T_0^T$$

observe that we can simply add the 2 previous matrices since they have the same size and we can factor out the vector of the modulation function T_0 , and multiply the second block with a z^{-1} to obtain the z-transform.

We see that we obtain a very compact representation for the modulation, where we **separated the prototype function and the modulation function** into 2 different matrices, and the matrix with the prototype function is a sparse matrix.

If we extend this example to filters of length 4 blocks, we get:

$$h_0(0) + z^{-1}h_0(1) + z^{-2}h_0(2) + z^{-3}h_0(3) =$$

$$= \begin{bmatrix} h(3) - z^{-2}h(11) & 0 & \dots & -h(7)z^{-1} + h(15)z^{-3} \\ 0 & h(2) - z^{-2}h(10) & -h(6)z^{-1} + h(14)z^{-3} & \cdot \\ \cdot & -h(5)z^{-1} + h(13)z^{-3} & h(1) - z^{-2}h(9) & \cdot \\ -h(4)z^{-1} + h(12)z^{-3} & \cdot & \cdot & h(0) - z^{-2}h(8) \end{bmatrix} \cdot \mathbf{T}_0^T$$

We can see that this is a **cross-shaped** matrix.

Observe: Every polynomial in this resulting matrix is a **downsampled** version of the prototype $h(n)$ by a factor of **2N**! (in our case $2N=8$). In addition, every second resulting value after this downsampling is also sign changed (we can also obtain this sign change by replacing z by $-z$ in the z -transform for the $2N$ downsampled versions).

So this matrix times our column vector \mathbf{T}_0^T above results in the polyphase representation of our subband filter $h_0(n)$ for the subband $k=0$. So this is the first of 4 filters we have in our modulated filter bank.

We obtain all 4 filters if we just use the complete DCT4 transform matrix \mathbf{T} , with

$$T_{n,k} = \cos\left(\frac{\pi}{N} \cdot (n+0.5) \cdot (k+0.5)\right), \quad n, k = 0, \dots, N-1,$$

instead of just the first column, \mathbf{T}_0^T .

For instance, with $N=4$ we get for \mathbf{T} ,

T =

$$\begin{bmatrix} 0.98079 & 0.83147 & 0.55557 & 0.19509 \\ 0.83147 & -0.19509 & -0.98079 & -0.55557 \\ 0.55557 & -0.98079 & 0.19509 & 0.83147 \\ 0.19509 & -0.55557 & 0.83147 & -0.98079 \end{bmatrix}$$

Fortunately, we obtain the same symmetries as for the first subband also for the higher subbands.

Summary:

If we have an analysis filter bank with impulse responses

$$h_k(n) = h(n) \cdot \cos\left(\frac{\pi}{4} \cdot (k+0.5) \cdot (n+0.5)\right)$$

its analysis **polyphase matrix** $H(z)$ can be written as a **multiplication of a sparse cross shaped matrix** with a DCT4 transform matrix

T as

$$H(z) = \begin{bmatrix} h(3) - z^{-2}h(11) & 0 & \dots & -h(7)z^{-1} + h(15)z^{-3} \\ 0 & h(2) - z^{-2}h(10) & -h(6)z^{-1} + h(14)z^{-3} & \cdot \\ \cdot & -h(5)z^{-1} + h(13)z^{-3} & h(1) - z^{-2}h(9) & \cdot \\ -h(4)z^{-1} + h(12)z^{-3} & \cdot & \cdot & h(0) - z^{-2}h(8) \end{bmatrix} \cdot T$$

For odd numbers of subbands N we would obtain one value or polynomial in the center of the matrix.

Sparse Matrices and the MDCT

Remember, the MDCT analysis impulse responses are

$$h_k(n) = h(n) \cdot \sqrt{\frac{2}{N}} \cdot \cos\left(\frac{\pi}{N} \left(k + \frac{1}{2}\right) \left(n + \frac{N}{2} + \frac{1}{2}\right)\right) \quad (2b)$$

and the filter length here is limited to $2N$, meaning $n=0, \dots, 2N-1$. Since the MDCT has a very similar modulation function as the DCT4, just with a time shift of $N/2$ in it, we suspect that we can also factor it into a sparse matrix and the DCT4 transform matrix, as

$$\mathbf{H}_{MDCT}(z) = \mathbf{F}_a(z) \cdot \mathbf{T}$$

with some sparse matrix $\mathbf{F}_a(z)$. If this assumption is true, we can obtain the sparse matrix by bringing the transform matrix in above formula on the other side,

$$\mathbf{F}_a(z) = \mathbf{H}_{MDCT}(z) \cdot \mathbf{T}^{-1}$$

We can simply start with constructing the complete polyphase matrix $\mathbf{H}_{MDCT}(z)$ using eq. (1), (2), plugging in the definition of $h_k(n)$ (2b), and then compute $\mathbf{F}_a(z)$ as follows,

$$\begin{aligned} & \mathbf{H}_{MDCT}(z) \cdot \mathbf{T}^{-1} = \\ & = \begin{bmatrix} \cdot & \cdot & -z^{-1} \cdot h(2N-1) & -h(N-1) & \cdot & 0 \\ 0 & \ddots & 0 & \cdot & \ddots & \cdot \\ -z^{-1} \cdot h(1.5N) & \cdot & 0 & \cdot & \cdot & -h(0.5N) \\ -z^{-1} \cdot h(1.5N-1) & \cdot & \cdot & 0 & \cdot & h(0.5N-1) \\ 0 & \ddots & \cdot & \cdot & \ddots & \cdot \\ 0 & \cdot & -z^{-1} \cdot h(N) & h(0) & \cdot & 0 \end{bmatrix} \end{aligned} \quad (3)$$

This is for the simple case of a **filter length of**

just $2N$, which is the length of a MDCT.
For odd numbers of subbands N we would get one value or polynomial in the top and bottom row centers and the left- and right-most columns.

Python Computation

We can try out this factorization for instance with the symbolic Math package “**sympy**” for **Python**.

Observe that exponentiation in Python is symbolized with two stars, “**”. We write a file, and name it “Famatrix.py”:

```
#Example for the extraction of the Fa Matrix from
the MDCT polyphase matrix
from sympy import *
from scipy import *

z=symbols('z')
N=4

#baseband prototype filter h(n):
h=symbols('h:8');
print( "h=")
print(h)

#MDCT Polyphase matrix H. Since each column
contains the time-reversed impulse response,
#we need the "N-1-n" instead of the "n":
#start with a NxN matrix of zeros:
H=Matrix(zeros((N,N)));
#range(0,N) produces indices from 0 to N-1.
```



```

#We compute H using eq. (1) and (2):
for n in range(0,N):
    for k in range(0,N):
        H[n,k]=h[N-1-n]*cos(pi/N*(N-1-
n+N/2+0.5)*(k+0.5))+z**(-1)*h[2*N-1-
n]*cos(pi/N*(2*N-1-n+N/2+0.5)*(k+0.5))

#Transform matrix T for the DCT4:
T=Matrix(zeros((N,N)));
for n in range(0,N):
    for k in range(0,N):
        T[n,k]=cos(pi/N*(n+0.5)*(k+0.5));

#Compute the sparse Fa matrix:
Fa= H*(T**(-1))

#Print the H matrix with 1 digit after the decimal
point and replacement of very small number by 0:
print( "H=")
print( H.evalf(1,chop=True))

#Print the Fa matrix with 1 digit after the
decimal point and replacement of very small number
by 0:
print( "Fa=")
print( Fa.evalf(1,chop=True))

```

Then we go to a terminal window and type:

python Famatrix.py

and we obtain:

```

h=
(h0, h1, h2, h3, h4, h5, h6, h7)

#The coefficients in the following polyphase matrix H come from the cosine
modulation function:
H=
Matrix(
[[-0.6*h3 - 0.8*h7/z, 1.0*h3 + 0.2*h7/z, -0.2*h3 + 1.0*h7/z, -0.8*h3 + 0.6*h7/z],
[-0.2*h2 - 1.0*h6/z, 0.6*h2 - 0.8*h6/z, -0.8*h2 - 0.6*h6/z, 1.0*h2 - 0.2*h6/z],
[0.2*h1 - 1.0*h5/z, -0.6*h1 - 0.8*h5/z, 0.8*h1 - 0.6*h5/z, -1.0*h1 - 0.2*h5/z],

```

```
[0.6*h0 - 0.8*h4/z, -1.0*h0 + 0.2*h4/z, 0.2*h0 + 1.0*h4/z, 0.8*h0 + 0.6*h4/z]]
Fa=
[ 0, 3.0e-16*h3 - 1.0*h7/z, -1.0*h3 + 2.0e-15*h7/z, 0]
[-4.0e-17*h2 - 1.0*h6/z, 0, 0, -1.0*h2 - 1.0e-15*h6/z]
[ 1.0e-17*h1 - 1.0*h5/z, 0, 0, 1.0*h1 - 2.0e-15*h5/z]
[ 0, 1.0e-16*h0 - 1.0*h4/z, 1.0*h0 + 1.0e-15*h4/z, 0]
```

Here we see that it is the same as eq. (3), apart from rounding errors, and that we get a diamond matrix shape. This **diamond shape** results from the **time-shift of N/2** in the cos modulation function. It has the effect of shifting the left and the right halves of the above cross shaped matrix (and multiplying it with z^{-1}). The result of shifting or exchanging those 2 halves is this diamond shaped matrix we can see here.

This matrix is now surprisingly simple, it only contains the $2N$ samples of our baseband prototype impulse response. **Most entries are zero**, and we have **delays only on the left hand side**. This was the purpose of the summand $N/2$ on the cos modulation function. The particular diamond shape of the matrix results from the symmetries of the cosine modulation, and the shift of $N/2$ on the modulation function for the time index.

The Delay Matrix

The next step is to separate out the delays, for which we use a delay matrix,

$$\mathbf{D}(z) = \begin{bmatrix} z^{-1} & 0 & . & . & . & 0 \\ 0 & \ddots & 0 & . & . & . \\ . & 0 & z^{-1} & 0 & . & . \\ . & . & 0 & 1 & 0 & . \\ . & 0 & . & . & \ddots & . \\ 0 & . & . & . & 0 & 1 \end{bmatrix}$$

For a faster (non-sympy) implementation in Python we can again write this as a polynomial with matrix coefficients,

$$\mathbf{D}(z) = \begin{bmatrix} 0 & 0 & . & . & . & 0 \\ 0 & \ddots & 0 & . & . & . \\ . & 0 & 0 & 0 & . & . \\ . & . & 0 & 1 & 0 & . \\ . & 0 & . & . & \ddots & . \\ 0 & . & . & . & 0 & 1 \end{bmatrix} \cdot z^0 + \begin{bmatrix} 1 & 0 & . & . & . & 0 \\ 0 & \ddots & 0 & . & . & . \\ . & 0 & 1 & 0 & . & . \\ . & . & 0 & 0 & 0 & . \\ . & 0 & . & . & \ddots & . \\ 0 & . & . & . & 0 & 0 \end{bmatrix} \cdot z^{-1}$$

and store these matrix coefficients on a 3-dimensional tensor,

```
Dp[:, :, 0] = np.diag(np.hstack((np.zeros(N/2), np.ones(N/2))))
Dp[:, :, 1] = np.diag(np.hstack((np.ones(N/2), np.zeros(N/2))))
```

We can factor out this delay matrix, and what remains is a matrix which only contains the coefficients of our baseband prototype filter, which we call "Folding Matrix" or "Filter Matrix"

\mathbf{F}_a ,

$$\mathbf{F}_a = \begin{bmatrix} \cdot & \cdot & -h(2N-1) & -h(N-1) & \cdot & 0 \\ 0 & \ddots & 0 & \cdot & \ddots & \cdot \\ -h(1.5N) & \cdot & 0 & \cdot & \cdot & -h(0.5N) \\ -h(1.5N-1) & \cdot & \cdot & 0 & \cdot & h(0.5N-1) \\ 0 & \ddots & \cdot & \cdot & \ddots & \cdot \\ 0 & \cdot & -h(N) & h(0) & \cdot & 0 \end{bmatrix} \quad (4)$$

so we get $\mathbf{F}_a(z) = \mathbf{F}_a \cdot \mathbf{D}(z)$.

Python Sympy Example

Start Python by typing "python" in a terminal window. The output in normal letters:

```
from sympy import *
Fa=Matrix([[0, 1, 4, 0],[2, 0, 0, 3],[3,
0, 0, -2],[0, 4, -1, 0]])
Fa
Matrix([
[0, 1, 4, 0],
[2, 0, 0, 3],
[3, 0, 0, -2],
[0, 4, -1, 0]])

z=symbols('z')
D=Matrix([[z**(-1), 0, 0, 0],[0, z**(-
1), 0, 0],[0, 0, 1, 0],[0, 0, 0, 1]])
```

D

```
Matrix([  
[1/z,  0, 0, 0],  
[ 0, 1/z, 0, 0],  
[ 0,  0, 1, 0],  
[ 0,  0, 0, 1]])
```

$F a^* D$

```
Matrix([  
[ 0, 1/z, 4, 0],  
[2/z,  0, 0, 3],  
[3/z,  0, 0, -2],  
[ 0, 4/z, -1, 0]])
```

Here now we have a matrix which only consists of real (or complex) valued elements, which we can now enter into a numerical math package, like MATLAB or Octave or Python. Observe that our time index n for the prototype function runs from the bottom up in this matrix.

Faster numerical Python Implementation

We use the function `polmatmult` from last time,

```
python
import numpy as np
from polmatmult import *
N=4

#Fa matrix:
Fa=np.zeros((N,N,1))
Fa[:, :, 0]=([[0, 1, 4, 0], [2, 0, 0, 3],
[3, 0, 0, -2], [0, 4, -1, 0]])

#Delay matrix:
Dp=np.zeros((N,N,2))
Dp[:, :, 0]=np.diag(np.hstack((np.zeros(N/2), np.ones(N/2))))
Dp[:, :, 1]=np.diag(np.hstack((np.ones(N/2), np.zeros(N/2))))

#Their product:
FaD=polmatmult(Fa, Dp)

#Answer:
FaD[:, :, 0]
array([[ 0.,  0.,  4.,  0.],
       [ 0.,  0.,  0.,  3.],
       [ 0.,  0.,  0., -2.],
       [ 0.,  0., -1.,  0.]])
FaD[:, :, 1]
array([[ 0.,  1.,  0.,  0.],
       [ 2.,  0.,  0.,  0.],
       [ 3.,  0.,  0.,  0.],
       [ 0.,  4.,  0.,  0.]])
```

Here we see: the first matrix is the coefficient matrix for z^0 , and the second is the coefficient matrix of z^{-1} . We see this result is identical to the calculation with sympy.

The Python Folding Matrix Function

We can implement a function `Famatrix` which generates the Folding matrix F_a from a general coefficient array `h`,

```
def Famatrix(h):
    """produces a diamond shaped folding
    matrix Fa from the coefficients h
    (h is a row matrix)
    """

    N = len(h)/2;
    print("Famatrix N=", N)
    #fliplr:
    h=h[::-1]
    Fa=np.zeros((N,N,1))
    Fa[0:N/2,0:N/2,0]=-
np.fliplr(np.diag(h[0:N/2]))
    Fa[N/2:N,0:N/2,0]=-np.diag(h[N/2:N])
    Fa[0:N/2,N/2:N,0]=-np.diag(h[N:
(N+N/2)])

    Fa[N/2:N,N/2:N,0]=np.fliplr(np.diag(h[(N
+N/2):2*N]))
```

```
return Fa
```

We store it in file Fafoldingmatrix.py. We test it:

```
python
from Fafoldingmatrix import *
h=np.array([1,2,3,4,5,6,7,8])
Fa=Famatrix(h)
Fa[:, :, 0]
array([[ 0., -8., -4.,  0.],
       [-7.,  0.,  0., -3.],
       [-6.,  0.,  0.,  2.],
       [ 0., -5.,  1.,  0.]])
```

We see that this is indeed the shape shown in eq. (4).

The Factorization

This means that we now have an easy way to write or construct our analysis polyphase matrix,

$$\mathbf{H}_{MDCT}(z) = \mathbf{F}_a \cdot \mathbf{D}(z) \cdot \mathbf{T}$$

Observe that these 3 matrices are now much simpler and also efficiently implementable. \mathbf{F}_a only contains real or complex numbers and is a sparse matrix, which can be efficiently implemented, also $\mathbf{D}(z)$ (only delays), and \mathbf{T} , the DCT4 transform, can also be efficiently implemented, for instance using our DCT4

function from lecture 2, which we apply to each block.

For the **synthesis** polyphase matrix we get

$$\begin{aligned}\mathbf{G}_{MDCT}(z) &= \mathbf{T}^{-1} \cdot \mathbf{F}_s(z) = \\ &= \mathbf{T}^{-1} \cdot \mathbf{z}^{-1} \cdot \mathbf{D}^{-1}(z) \cdot \mathbf{F}_s.\end{aligned}$$

We can apply the same trick, now with the **synthesis polyphase matrix**. We assume that we also can write it with a sparse matrix, $\mathbf{F}_s(z)$, now with the inverse transform matrix in the beginning to invert the analysis transform matrix,

solving for the sparse matrix $\mathbf{F}_s(z)$ yields

$$\mathbf{T} \cdot \mathbf{G}_{MDCT}(z) = \mathbf{F}_s(z)$$

which results in

$$\begin{aligned}\mathbf{F}_s(z) &= \mathbf{T} \cdot \mathbf{G}_{MDCT}(z) = \\ &= \begin{bmatrix} . & . & g(0.5N-1) & g(0.5N) & . & 0 \\ 0 & \ddots & 0 & . & \ddots & . \\ g(0) & . & 0 & . & . & g(N-1) \\ z^{-1} \cdot g(N) & . & . & 0 & . & -z^{-1} \cdot g(2N-1) \\ 0 & \ddots & . & . & \ddots & . \\ 0 & . & z^{-1} \cdot g(1.5N-1) & -z^{-1} \cdot g(1.5N) & . & 0 \end{bmatrix}\end{aligned}$$

which we can obtain using Python, with
`python Fsmatrix.py`

We again get the diamond shape, but a

different ordering of the coefficients of the baseband prototype $g(n)$.

The time or phase coefficient n of our baseband prototype $g(n)$ runs from left to right (which also corresponds to the transposing of matrices). Observe that here the delays are on the lower half of our matrix. We can again factor out these delays with a matrix, which we multiply from the left hand side, and which has the delay on the lower half of its diagonal. We can again write this needed matrix using our just defined matrix $\mathbf{D}(z)$, by using its inverse. Remember that the inverse of a diagonal matrix is obtained by inverting each element of its diagonal separately.

$$\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}^{-1} = \begin{bmatrix} 1/a & 0 \\ 0 & 1/b \end{bmatrix}$$

Hence we get:

$$\mathbf{D}^{-1}(z) = \begin{bmatrix} z^{-1} & 0 & \dots \\ 0 & \ddots & 0 \\ 0 & \dots & 1 \end{bmatrix}^{-1} = \begin{bmatrix} z & 0 & \dots \\ 0 & \ddots & 0 \\ 0 & \dots & 1 \end{bmatrix}$$

This means we replace the z^{-1} by z in the Delay matrix. But this would result in a non-causal system (z denotes the future block). To make it causal, and also to obtain the delays on the lower half of the diagonal, we need to multiply the inverse matrix by z^{-1} :

$$\mathbf{D}^{-1}(z) \cdot z^{-1} = \begin{bmatrix} 1 & 0 & \cdots & . & . & . \\ 0 & \ddots & 0 & . & . & . \\ 0 & \cdots & 1 & . & . & . \\ 0 & . & . & z^{-1} & . & . \\ . & . & . & . & \ddots & . \\ . & . & . & . & . & z^{-1} \end{bmatrix}$$

This now has the delays z^{-1} on the lower part of its diagonal, as needed.

Using this result, we obtain the synthesis "folding matrix" \mathbf{F}_s from above,

$$\mathbf{F}_s = \begin{bmatrix} . & . & g(0.5N-1) & g(0.5N) & . & 0 \\ 0 & \ddots & 0 & . & \ddots & . \\ g(0) & . & 0 & . & . & g(N-1) \\ g(N) & . & . & 0 & . & -g(2N-1) \\ 0 & \ddots & . & . & \ddots & . \\ 0 & . & g(1.5N-1) & -g(1.5N) & . & 0 \end{bmatrix} \quad (5)$$

In this way we can rewrite the synthesis polyphase matrix as

$$\mathbf{G}_{MDCT}(z) = \mathbf{T}^{-1} \cdot z^{-1} \cdot \mathbf{D}^{-1}(z) \cdot \mathbf{F}_s$$

Perfect Reconstruction

The direct concatenation of the analysis and synthesis filter banks, without any processing in between, leads to the product of their

polyphase matrices):

$$\begin{aligned} \mathbf{H}_{MDCT}(z) \cdot \mathbf{G}_{MDCT}(z) &= \mathbf{F}_a \cdot \mathbf{D}(z) \cdot \mathbf{T} \cdot \mathbf{T}^{-1} \cdot z^{-1} \cdot \mathbf{D}^{-1}(z) \cdot \mathbf{F}_s = \\ &= \mathbf{F}_a \cdot z^{-1} \cdot \mathbf{F}_s \end{aligned}$$

should result in a pure delay (remember: our polphase representation is in the donwsampled domain, since all polyphase elements are downsampled sequences at different phase; hence a multiplication with z^{-1} corresponds to a delay of 1 sample in the downsampled domain, which corresponds to a delay of 1 block in our original signal domain).

The pure delay is the case if we choose

$$\mathbf{F}_s = \mathbf{F}_a^{-1}$$

The we can compare this result with eq. (5) to obtain the synthesis prototype function $g(n)$ for perfect reconstruction!

Example in Python: Again assume we have the block length and number of subbands $N=4$, and our baseband prototype impulse response is $h(n)=[1,2,3,4,4,3,2,1]$. Using eq. (4) we get the analysis Folding matrix \mathbf{F}_a (as above)

```

from sympy import *
Fa=Matrix([[0, -1, -4, 0], [-2, 0, 0, -3],
[-3, 0, 0, 2], [0, -4, 1, 0]])
Fa
Matrix([
[0, -1, -4, 0],
[-2, 0, 0, -3],
[-3, 0, 0, 2],
[0, -4, 1, 0]])

```

and its inverse becomes the Folding matrix for the synthesis, to ensure perfect reconstruction, is $F_a^{(-1)}$:

```

Fa**(-1)
Matrix([
[ 0, -2/13, -3/13,  0],
[-1/17,  0,  0, -4/17],
[-4/17,  0,  0, 1/17],
[ 0, -3/13, 2/13,  0]])

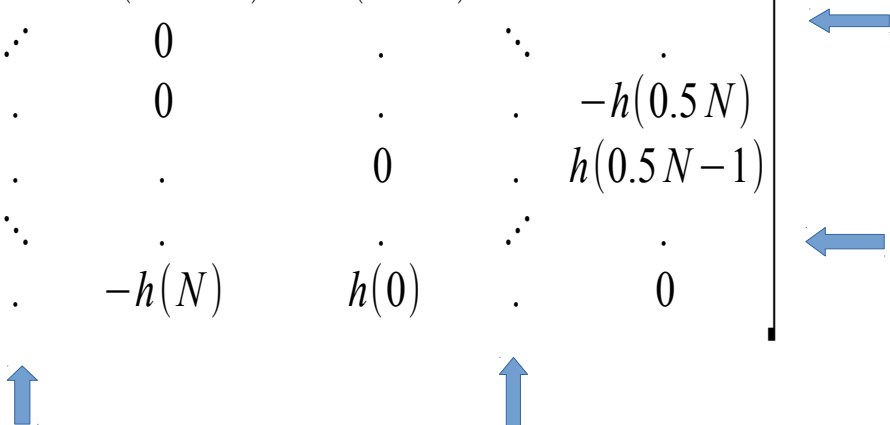
```

This is now equal to the synthesis folding matrix F_s , and comparing it with its eq. (5) we can read out the resulting synthesis baseband impulse response as:

$$g(n) = \left[\frac{1}{17}, \frac{2}{13}, \frac{3}{13}, \frac{4}{17}, \frac{4}{17}, \frac{3}{13}, \frac{2}{13}, \frac{1}{17} \right]$$

we can see that its values not quite identical to the analysis prototype $h(n)$. The numerator is the same, but the denominators change.

To find out more about the inverse in an analytical way, the matrices F_a and F_s in (4) and (5) can be treated as nested **2x2 sub-matrices**. We take the 2 non-zero entries of a given row n , and the corresponding non-zero entries of the “mirrored” row $N-1-n$, and these 4 non-zero entries become our submatrix.

$$F_a = \begin{bmatrix} \cdot & \cdot & -h(2N-1) & -h(N-1) & \cdot & 0 \\ 0 & \ddots & 0 & \cdot & \ddots & \cdot \\ -h(1.5N) & \cdot & 0 & \cdot & \cdot & -h(0.5N) \\ -h(1.5N-1) & \cdot & \cdot & 0 & \cdot & h(0.5N-1) \\ 0 & \ddots & \cdot & \cdot & \ddots & \cdot \\ 0 & \cdot & -h(N) & h(0) & \cdot & 0 \end{bmatrix}$$


These submatrices have the following form. Because they are nested into each other they don't interact with the other 2x2 submatrices in the multiplication). Hence we only need to consider 2x2 matrices at position n ,

$$\begin{bmatrix} -h(2N-1-n) & -h(N-1-n) \\ -h(N+n) & h(n) \end{bmatrix}$$

where $n=0, \dots, N/2-1$, such that we obtain

$N/2$ submatrices. For instance, the submatrix for $n=0$ would contain the elements of F_a of the first row and the last row.

For bigger n , the submatrices contain the non-zero elements of the corresponding rows in between (for $n=1$ the second row and the row one before the last row). In this way we reduced our bigger matrix into several smaller, simpler matrices.

The inverse of these sub-matrices is easily obtained in closed form,

$$\frac{1}{h(2N-1-n)h(n)+h(N-1-n)h(N+n)} \begin{bmatrix} -h(n) & -h(N-1-n) \\ -h(N+n) & h(2N-1-n) \end{bmatrix}$$

Observe that the denominator is the **negative determinant** ($-\det(\cdot)$) of the sub matrix.

The corresponding sub-matrices for the synthesis are,

$$\begin{bmatrix} g(n) & g(N-1-n) \\ g(N+n) & -g(2N-1-n) \end{bmatrix}$$

and by setting them equal we get a solution for perfect reconstruction,

$$\begin{bmatrix} g(n) & g(N-1-n) \\ g(N+n) & -g(2N-1-n) \end{bmatrix} = \frac{1}{h(2N-1-n)h(n)+h(N-1-n)h(N+n)} \begin{bmatrix} -h(n) & -h(N-1-n) \\ -h(N+n) & h(2N-1-n) \end{bmatrix}$$

Here we can see that if we choose the determinant to be -1, (the **denominator to be**

1), then we get **identical analysis and synthesis prototype filters** (up to the sign, and not necessarily para-unitary polyphase matrices). If we choose $\det(..)=-1$, then the comparison (of the submatrices) shows that

$$g(n) = -h(n)$$

for $n=0, \dots, 2N-1$. This means the **analysis and synthesis window must be identical** if $\det(..)=-1$ for perfect reconstruction! (This must be true for every such sub-matrix)

In conclusion: if we design our window $h(n)$ such that $\det(..)=-1$, then we automatically obtain perfect reconstruction if we choose

$$g(n) = -h(n)$$

Observe that this is a **very powerful result**. It tells us how to obtain perfect reconstruction, including the **cancellation of all alias components**, even though we didn't even look at them!

Question: Is this now also an orthogonal, or rather **para-unitary** matrix? Let's see: For a para-unitary polyphase matrix $H(z)$ (meaning $H^{-1}(z) = H^T(z^{-1})$), we already have an orthogonal transform matrix T (which in this case of a real valued matrix also means orthogonality, $T^{-1} = T^T$). Hence **T is also para-unitary**. For real or complex valued

matrices, orthogonality and para-unitarity is the same.

Now we check the delay matrix. For $\mathbf{D}(z)$ the transpose is identical to the original since it is a diagonal matrix.

Remember, we had

$$\mathbf{D}(z) = \begin{bmatrix} z^{-1} & 0 & . & . & . & 0 \\ 0 & \ddots & 0 & . & . & . \\ . & 0 & z^{-1} & 0 & . & . \\ . & . & 0 & 1 & 0 & . \\ . & 0 & . & . & \ddots & . \\ 0 & . & . & . & 0 & 1 \end{bmatrix}$$

If we replace z by z^{-1} (and take the transpose) we get

$$\mathbf{D}^T(z^{-1}) = \begin{bmatrix} z & 0 & \dots & 0 & 0 \\ 0 & \ddots & 0 & 0 & 0 \\ 0 & \dots & z & 0 & 0 \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

If we multiply those two matrices, we get the identity matrix,

$$\mathbf{D}(z) \cdot \mathbf{D}(z^{-1}) = \mathbf{I}$$

This means that also the delay matrix $\mathbf{D}(z)$ is **para-unitary**.

Now we check the Folding matrix. If we now design \mathbf{F}_a such that it is orthonormal (with

$F_a^{-1} = F_a^T$), then the entire polyphase matrix $H(z) = F_a \cdot D(z) \cdot T$ will become para-unitary (using $(A \cdot B)^T = B^T \cdot A^T$):

$$H^T(z^{-1}) = (F_a \cdot D(z^{-1}) \cdot T)^T = T^T \cdot D(z^{-1}) \cdot F_a^T$$

Since our transform matrix and folding matrix are orthonormal, we have $T^T = T^{-1}$ and

$F_a^T = F_a^{-1}$, we see that our analysis polyphase matrix is indeed para-unitary:

$$H^T(z^{-1}) = T^{-1} \cdot D^{-1}(z) \cdot F_a^{-1} = H^{-1}(z) .$$

This means **if** we ensure that **every matrix** of our product is **para-unitary**, the **product** is also **para-unitary**.

So what does this mean for our analysis folding matrix F_a ? We now know that in our special case of an orthonormal folding matrix, we have the property $F_a^{-1} = F_a^T$, which we can also apply to our submatrices. How do we get our matrix F_a orthonormal?

We take our general inverse, and set it equal to the transposed matrix:

$$\begin{aligned} & \begin{bmatrix} h(2N-1-n) & h(N-1-n) \\ h(N+n) & -h(n) \end{bmatrix}^{-1} = \\ & = \frac{1}{h(2N-1-n)h(n) + h(N-1-n)h(N+n)} \begin{bmatrix} h(n) & h(N-1-n) \\ h(N+n) & -h(2N-1-n) \end{bmatrix} \end{aligned}$$

$$\stackrel{!}{=} \begin{bmatrix} h(2N-1-n) & h(N-1-n) \\ h(N+n) & -h(n) \end{bmatrix}^T$$

Assume we have $\det(..)=-1$, then we get the result:

$$\begin{bmatrix} h(n) & h(N-1-n) \\ h(N+n) & -h(2N-1-n) \end{bmatrix} = \begin{bmatrix} h(2N-1-n) & h(N+n) \\ h(N-1-n) & -h(n) \end{bmatrix}$$

We can see that the left side is the result of the inversion, and the right hand side the result of transposing.

Looking at the 2 matrices, with $n=0,..,N-1$, we see that

$$h(n)=h(2N-1-n)$$

This means we have a **symmetric window, symmetric around its center!** It looks the same forward and backwards. If this is the case, then also **F_a is para-unitary.**

In summary, this means:

-If we have the determinant of our 2x2 submatrices to be **$\det(..)=-1$** (as part of the design process), then the baseband impulse responses for analysis and synthesis are identical:

$$h(n)=-g(n)$$

-If we also would like to have orthogonality or **para-unitarity**, we need **symmetric**

baseband impulse responses:

$$h(n) = -g(n)$$

$$h(n) = h(2N-1-n)$$

These are 2 important properties, which we obtained for our baseband impulse responses!

A simple example for this case ($\det(..) = -1$ and orthogonality) is the sine window,

$$h(n) = g(n) = \sin\left(\frac{\pi}{2N}(n+0.5)\right)$$

This case also leads to **para-unitary polyphase matrices** because it is a symmetric window (symmetric around its center).

The sine window is often used in the MDCT filter bank, because it is easy to design. We know it leads to perfect reconstruction and still has a reasonable frequency response. For instance the raised cosine window would even have a better frequency response, but it would not lead to perfect reconstruction if it also used for the synthesis (it does not fulfill the $\det(..) = -1$ condition). For perfect reconstruction we need to compute a different synthesis prototype.

So **para-unitarity** is good for an **easy design** for perfect reconstruction

Remark: For para-unitary polyphase matrices Parseval's Theorem for the energy conservation holds (in the limit of long sequences), meaning

the total energy in the signal in the time domain is equal to the total energy in all subbands. This can be used, for instance, for the estimation of the energy of the quantization error in the reconstructed signal (total energy of the quantization error in the subbands is equal to the energy of the quantization error in the reconstructed signal). This is important for instance for designing quantizers.

Observe that we **don't necessarily need the symmetry** of our baseband impulse response. If we only have $\det(..)=-1$, we still have $h(n)=g(n)$, but they **don't need to be symmetric**. In this case, Parseval's Theorem does not need to hold (it can still hold approximately).

Remark: Observe the factor z^{-1} , which we obtained in the inversion of $\mathbf{D}(z)$ to obtain causal filters. This means we get a **delay of one block**, and it is the source of the **algorithmic or system delay** n_d of our analysis and synthesis filter bank (if the synthesis follows directly after the analysis filter bank).

In general the system delay is the blocking delay of $N-1$ samples to assemble the signal into blocks of length N , plus the delay needed

to make our matrices causal.

In the case of the MDCT we get a blocking delay of $N-1$ plus the delay from our Delay matrix, which results in a total delay of $2N-1$ samples.

In the MDCT case our filter length is $L=2N$, hence our total or system delay can also be written as $n_d=L-1$.

We see that the delay is **coupled to the filter length**. This is true in general for orthogonal or **para-unitary** filter banks, also for longer filters we obtain $n_d=L-1$. This is one of the **drawbacks of orthogonal filter banks**. To obtain lower delay, we need to take non-orthogonal or **non-pa-unitary** matrices! So **non-pa-unitary matrices** or filter banks can have some **important advantages**, but they are **more difficult to design**.

MDCT Python Implementation, Analysis

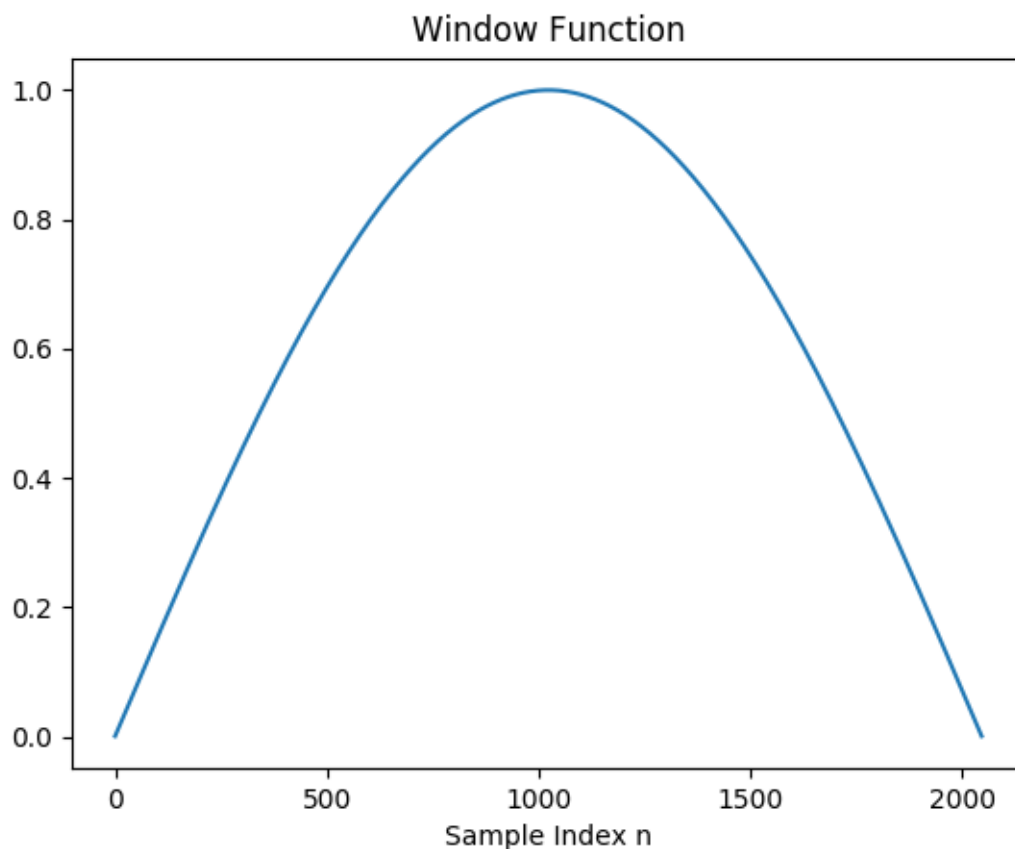
The following shows an MDCT implementation with a sine window and $n=1024$ subbands,

```
ipython --pylab
import scipy.io.wavfile as wav
from x2polyphase import *
from polmatmult import *
```

```

from Fafoldingmatrix import *
from DCT4 import *
rate, x = wav.read('04_topchart.wav')
#Number of subbands:
N=1024
#MDCT sine window:
h=np.sin(np.pi/(2*N)*(np.arange(2*N)+0.5))
#Plot window function:
plot(h)
title('Window Function')
xlabel('Sample Index n')

```



```

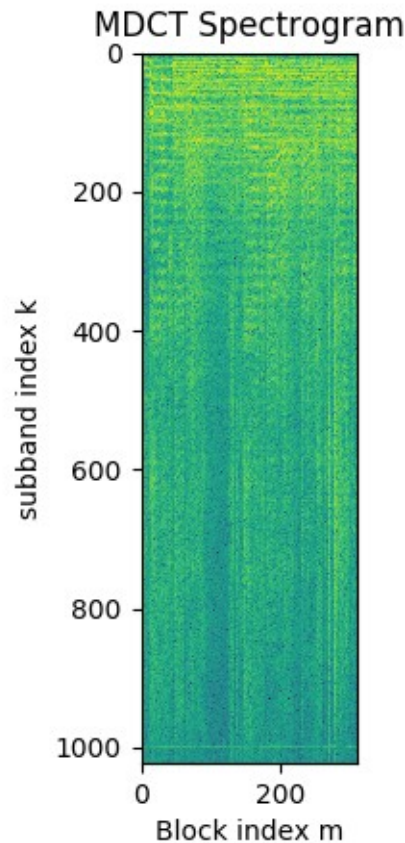
#Folding Matrix:

```

```

Fa=Famatrix(h)
#Delay Matrix D(z):
Dp=zeros((N,N,2))
Dp[:, :, 0]=diag(hstack((zeros(N/2), ones(N/2))))
Dp[:, :, 1]=diag(hstack((ones(N/2), zeros(N/2))))
#Fa*D(z):
Faz=polmatmult(Fa, Dp)
xp=x2polyphase(x, N)
yp=polmatmult(xp, Faz)
#Number of blocks:
L=yp.shape[2]
#Apply DCT4 transform to the rows:
for m in range(L):
    yp[0, :, m]=DCT4(yp[0, :, m])
#Resulting spectrogram image:
imshow(log(abs((yp[0, :, :]))))
title('MDCT Spectrogram')
ylabel('subband index k')
xlabel('Block index m')

```

Observe: The highest frequencies are above, at the small subband indices k , and we have only 313 blocks horizontally, due to critical sampling. Compare it with the STFT spectrogram, which was not critically sampled.

MDCT Synthesis Filter Bank

From this critically sampled version we can still go back to the original time domain signal, using the synthesis MDCT filter bank for perfect reconstruction (continued from above)

```
from polyphase2x import *
#Compute the inverse folding matrix for the
Synthesis:
```

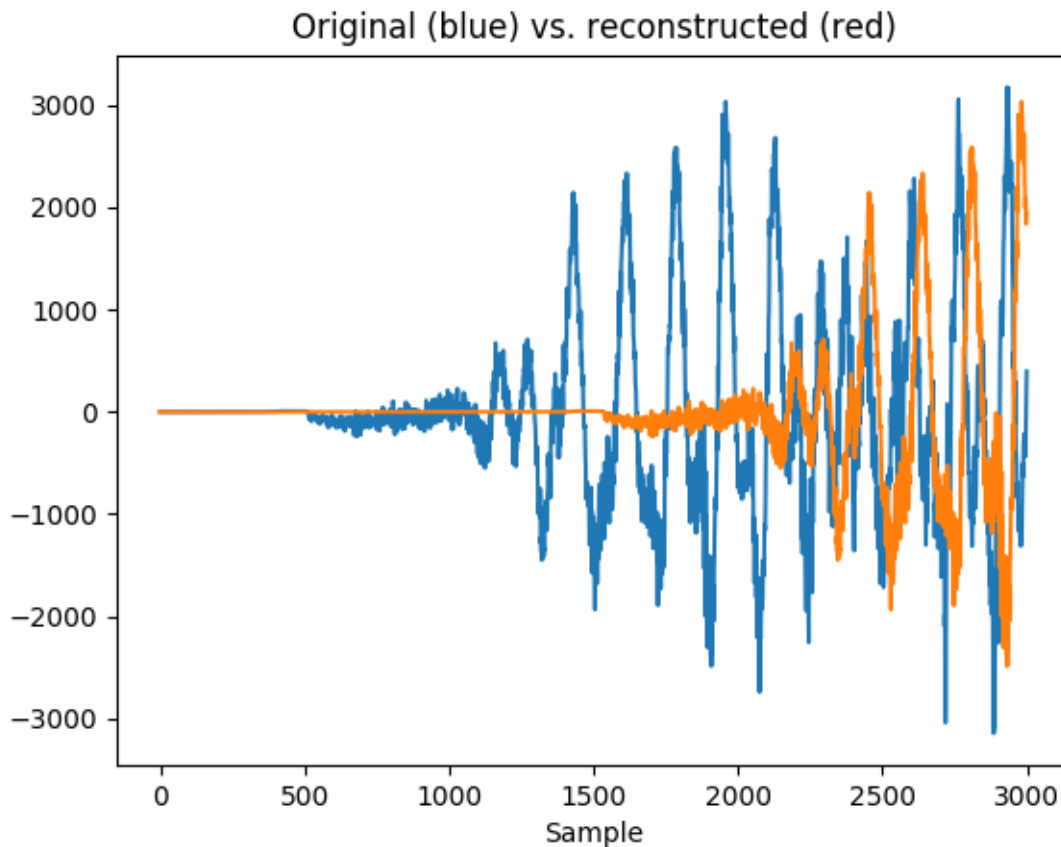
```

Fs=zeros(Fa.shape)
Fs[:, :, 0]=inv(Fa[:, :, 0])
#apply the inverse transform (which for the DCT4 is
identical to the forward transform):
L=yp.shape[2]
#Apply DCT4 (inverse) transform to the rows:
for m in range(L):
    yp[0, :, m]=DCT4(yp[0, :, m])

#Inverse Delay Matrix with delay:
Dpi=zeros((N,N,2))
Dpi[:, :, 1]=diag(hstack((zeros(N/2), ones(N/2))))
Dpi[:, :, 0]=diag(hstack((ones(N/2), zeros(N/2))))

#multiply with inverse delay matrix with delay:
yp=polmatmult(yp,Dpi)
#Multiply with synthesis folding matrix Fs:
xrekp=polmatmult(yp,Fs)
xrek=polyphase2x(xrekp)
xrekp.shape
#Out: (1, 1024, 314)
xrek.shape
#Out: (1, 321536)
plot(x[1000:4000])
plot(xrek[0,1000:4000])
title('Original (blue) vs. reconstructed (red)')

```



Observe the delay of 1024 samples between the original and the reconstructed signal (which is the system delay without the blocking delay, since the signal is already in memory).