Optimization of Filter Banks

Goal: Obtain a filter bank from our structure or product $\boldsymbol{P}_a(z)=\boldsymbol{F}_a\cdot\boldsymbol{D}(z)\cdot\boldsymbol{G}(z)\ldots.$, which has "good" subband filters, i.e. a good or sufficient stopband attenuation and not much pass band attenuation. An example could be a desired stopband attenuation of -60dB and a pass band attenuation of less than -3dB.

Problem to solve: The coefficients which determine the frequency responses are the coefficients in our cascade or product, the coefficients in our matrices e_j^i , h , g_j^i ... They result in the frequency response in a non-linear way, hence the usual filter design approaches don't work here, previous approaches like the Remez-exchange algorithm. Those traditional filter design algorithms can also be seen as optimization algorithms for more or less linear problems (or quadratic if we take the mean squared error).

We can read out the resulting analysis baseband prototype impulse response $h_a(n)$ for instance by computing the sparse folding matrix the product

$$\boldsymbol{F}_a(z)=\boldsymbol{F}_a\cdot\boldsymbol{D}(z)\prod_{n=0}^{P-1}\boldsymbol{G}_n(z)\prod_{m=0}^{Q-1}\boldsymbol{H}_m(z)$$

(without the transform matrix $\boldsymbol{T}$), and then compare it with eq. (1) of slides 15,

$$\boldsymbol{F}_a(z)=$$

$$=\begin{bmatrix} . & 0 & -H_{2N-1}^{\downarrow 2N}(-z^2)\cdot z^{-1} & -H_{N-1}^{\downarrow 2N}(-z^2) & 0 & . \\ . & . & 0 & 0 & . & . \\ -H_{1.5N}^{\downarrow 2N}(-z^2)\cdot z^{-1} & 0 & . & . & 0 & -H_{N/2}^{\downarrow 2N}(-z^2) \\ -H_{1.5N-1}^{\downarrow 2N}(-z^2)\cdot z^{-1} & 0 & . & . & 0 & H_{N/2-1}^{\downarrow 2N}(-z^2) \\ 0 & . & 0 & . & . & 0 \\ . & 0 & -H_N^{\downarrow 2N}(-z^2)\cdot z^{-1} & H_0^{\downarrow 2N}(-z^2) & . & . \end{bmatrix}$$

$$H_n^{\downarrow 2N}(z):=\sum_{m=0}^{\infty}h_a(m2N+n)\cdot z^{-m}$$

In this way we obtain $h_a(n)$, and hence its frequency response, from our matrix coefficients $\boldsymbol{x}=[h_1, h_2, \ldots, e_j^i, \ldots, h_j^i, \ldots]$. We can do the same for the synthesis side, or we restrict the (sub-) matrices to have det(.)=-1, in which case the synthesis baseband impulse reponse is the same as for the analysis.

Another possibility to obtain the analysis baseband impulse response is to compute the polyphase matrix $\boldsymbol{P}_a(z)$. Its first column contains the **polyphase elements of the first subband filter in reverse order**. This can be

used to read out its impulse response $h_0(n)$. To obtain $h_a(n)$ we simply **divide** $h_0(n)$ by the **modulation function** for the first subband (for k=0).

Our approach:
Since we now have this non-linear dependency of our baseband impulse response from the matrix coefficients, we need to use “more powerful” optimization algorithms, which are made for just mostly convex error functions.
We first define an **error function**, and then we use an **optimization** algorithm to minimize this error function. This error function can be the sum of the magnitudes, or the squares, of the **differences between our obtained frequency response**, given our unknown variables at some point, **and our desired frequency response**. To give the stop band attenuation and the pass band ripples different weights, we can also **assign weights to the different frequency regions** for our error function.

So our starting point is a vector of all of our unknowns of our matrices,

$$\boldsymbol{x}=[h_{1,}h_2,\ldots,e^i_j,\ldots,h^i_{j\ldots}]=[x_{1,}x_{2\ldots}]$$

We now define a function which computes the

baseband impulse response out of these unknowns, by multiplying our matrices to obtain our final folding matrix or the polyphase matrix, and then read out the baseband impulse response.

This then yields the baseband prototype for our coefficient set x , which we call $h_x(n)$. This is now used to compute a **weighted frequency response, at k frequency points**, with weights w_i for each frequency point,

$$H_i = \sum_{n=0}^{L \cdot N - 1} h_x(n) e^{j \omega_i \cdot n} \cdot w_i, for\, i = 0, \ldots, k-1.$$

(see also Schuller, Smith: “New Framework for Modulated Perfect Reconstruction Filter Banks”, IEEE Transactions on Signal Processing, August 1996)

For instance we can use “freqz” to compute H_i (by default it computes 512 equally spaced frequency points), and then multiply it with the weights w_i .

We choose as many frequency points as necessary to sufficiently cover our frequency response. Usually it should be **several times the length of our filter**, to avoid sampling the frequency response accidentally near its zeros. The more important frequency points in terms of attenuation get the higher weights.

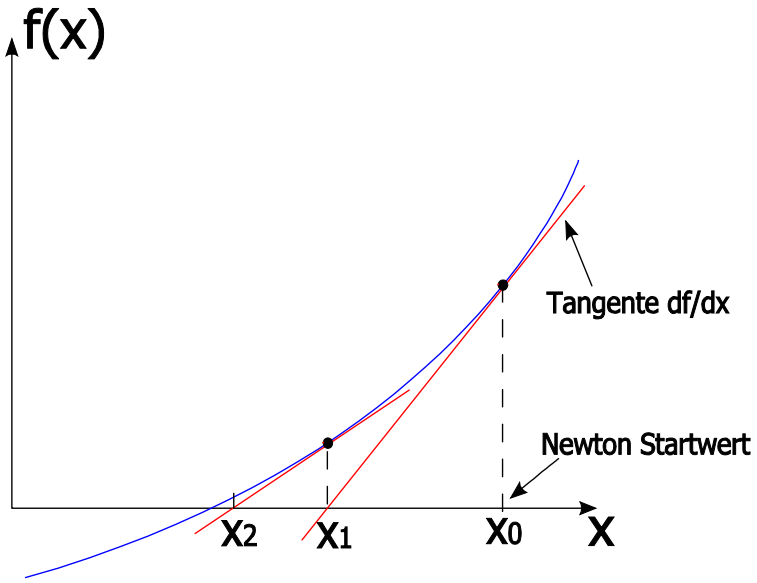
The same can also be done for the synthesis, but remember that this is only necessary of we don't have det(..)=-1, because for det(..)=-1, we obtain identical prototype impulse responses for analysis and synthesis, and we only need to optimize the analysis (Lecture 13 and 14).
We also need to have a vector containing the weighted desired frequency response, which would be 1 for the pass band (multiplied by the corresponding weights), and zeros in the stop band. We call this $\boldsymbol{d}$ (for desired).
Our error function of our vector of unknowns $\boldsymbol{x}$ is now

$$f(\boldsymbol{x})=\sum_{i=1}^{2k}|H_i(\boldsymbol{x})-d_i|^2=(\boldsymbol{H}-\boldsymbol{d})\cdot\overline{(\boldsymbol{H}-\boldsymbol{d})^T}$$

(row times column vector). Now we just need to minimize this function $f(\boldsymbol{x})$ with respect to our vector of unknows $\boldsymbol{x}$.

This leads us to optimization in general. The goal of optimization is to find the vector $\boldsymbol{x}$ which minimizes the error function $f(\boldsymbol{x})$.
We know: in a minimum, the functions derivative is zero, $f'(\boldsymbol{x}):=\frac{df(\boldsymbol{x})}{d\boldsymbol{x}}=\boldsymbol{0}$.

An approach to iteratively find the zero of a function is **Newtons method**. Take some function f(x), where x is not a vector but just a number, then we can find its zero as depicted in the following picture,



with the iteration

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, 2, \ldots$$

Observe: Sometimes we need to compute the derivative numerically, by evaluating the function a few times in the neighborhood, and eg. compute $\Delta f / \Delta x$, for a small Δx (there more involved methods for it).

Now we want to find the zero not of $f(x)$, but of $f'(x)$, hence we simply replace $f(x)$ by $f'(x)$

and obtain the following iteration,

$$x_{k+1}=x_k-\frac{f'\left(x_k\right)}{f''\left(x_k\right)}$$

To obtain a minimum (not a maximum) we need the condition $f''\left(x_k\right)>0$.

Example for finding a zero of a function:
Compute $\sqrt{2}$ numerically by finding the zero of the function

$$f\left(x\right)=x^2-2$$

We know the solution is $\sqrt{2}$. Now apply Newtons Method to find this solution numerically. The first derivative is

$$f'\left(x\right)=2x$$

The Newton iteration is

$$x_{k+1}=x_k-\frac{f\left(x_k\right)}{f'\left(x_k\right)}=x_k-\frac{x_k^2-2}{2x_k}$$

In ipython and with a starting value of 1 this becomes:

```
ipython3 -pylab
x=1.0
x=x-(x**2-2)/(2*x)
#x =  1.5000
x=x-(x**2-2)/(2*x)
#x =  1.4166666666666667
x=x-(x**2-2)/(2*x)
#x =  1.4142156862745099
```

(True value: 1.414213562373095 1)
We see: after only 3 iterations we obtain $\sqrt{2}$ with 6 digits accuracy!

Example: find the **minimum** of $\cos(x)$. We know a minimum is at $x=\pi$, so this is also a way to numerically determine π .
We have

$$f(x)=\cos(x)$$
$$f'(x)=-\sin(x)$$
$$f''(x)=-\cos(x)$$

Newton update:

$$x_{new}=x_{old}-f'(x_{old})/f''(x_{old})=x_{old}-\sin(x_{old})/\cos(x_{old})$$

In ipyhton we simply repeatedly use the expression:

```
x=x-sin(x)/cos(x)
```

We start with x=3 and proceed with the Newton update,

```
x=3;
x=x-sin(x)/cos(x)
#x =  3.142546543074278
x=x-sin(x)/cos(x)
#x =  3.141592653300477
```

We see that this iteration indeed converges to π ! It computes π =3.141592653589793 with an accuracy of 10 digits after only 2 iterations!

For a **multi-dimensional** function, where the argument $\boldsymbol{x}$ is a vector, the first derivative is a vector called “Gradient”, with symbol Nabla ∇ , because we need the derivative with respect to each element of the argument vector $\boldsymbol{x}$,

$$\nabla f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

(where n is the number of unknowns in the argiment vector $\boldsymbol{x}$). For the second derivative, we need to take each element of the gradient vector and again take the derivative to each element of the argument vector. Hence we obtain a matrix, the **Hesse Matrix**, as matrix of second derivatives,

$$H_f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix}$$

Observe that this Hesse Matrix is symmetric.
Using these definitions we can generalize our Newton algorithm to the multi-dimensional case.
The one-dimensional iteration

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

turns into the multi-dimensional iteration

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} - H_f^{-1}(\boldsymbol{x_k}) \nabla f(\boldsymbol{x_k})$$

For a minimum, $H_f(\boldsymbol{x})$ must be positive definite (all eigenvalues are positive).
The problem here is that for the Hesse matrix we need to compute n^2 second derivatives, which can be computationally too complex, and then we need to invert this matrix. Hence we make the simplifying assumption, that the Hesse matrix can be written as a **diagonal matrix with constant values on the diagonal**.
Observe that this is mostly is mostly a very crude approximation, but since we have an iteration with many small updates it can still work,

$$\boldsymbol{H_f}(\boldsymbol{x_k}) = \frac{1}{\alpha} \cdot \boldsymbol{I}$$

The best value of α depends on how good it approximates the Hesse matrix.
Hence our iteration $\boldsymbol{x_{k+1}} = \boldsymbol{x_k} - \boldsymbol{H_f^{-1}}(\boldsymbol{x_k}) \cdot \nabla f(\boldsymbol{x_k})$ turns into

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} - \alpha \cdot \nabla f(\boldsymbol{x_k}) \quad (1)$$

which is much simpler to compute. This is also called “Steepest Descent”, because the gradient tell us the direction of the steepest descent, or “Gradient Descent”.
We see that the update of $\boldsymbol{x}$ consists only of the gradient $\nabla f(\boldsymbol{x_k})$ scaled by the factor α . In each step, we reduce the value of $f(\boldsymbol{x})$ by moving $\boldsymbol{x}$ in the direction of the gradient. If we make α larger, we obtain larger update steps and hence quicker convergence to the minimum, but it may oscillate around the minimum. For smaller α the steps become smaller, but it will converge more precisely to the minimum.
Because of the update direction along the gradient, this method is also called “**Gradient Descent**”.

Example:
Find the 2-dimensional minimum of the function
$f(x_{1,}x_2) = \cos(x_1) - \sin(x_2)$
Its gradient is
$\nabla f(x_{1,}x_2) = [-\sin(x_1), -\cos(x_2)]$

In ipython we choose $\alpha = 1$ and a starting point of $[x_{1,}x_2] = [2,2]$,

```
alpha=1;
#start:
x=array([2,2])
x= x -alpha*array([-sin(x[0]), -cos(x[1])])
#x =
#array([2.90929743, 1.58385316])
x= x -alpha*array([-sin(x[0]), -cos(x[1])])
#x =
#array([3.13950913, 1.5707967 ])
x= x -alpha*array([-sin(x[0]), -cos(x[1])])
#x =
#array([3.14159265, 1.57079633])
```

We see: the minimum is obtained for $x_1 = 3.14159265$ (exact value: $\pi = 3.141592653589793$) and $x_2 = 1.57079633$ (exact value: $\pi/2 = 1.570796326794896$). Observe that we needed **3 iterations** to obtain 9 digit accuracy in this 2-dimensional case. In the 1-dimensional case with the **Newton** iteration we needed only **2 iterations** for the same accuracy.

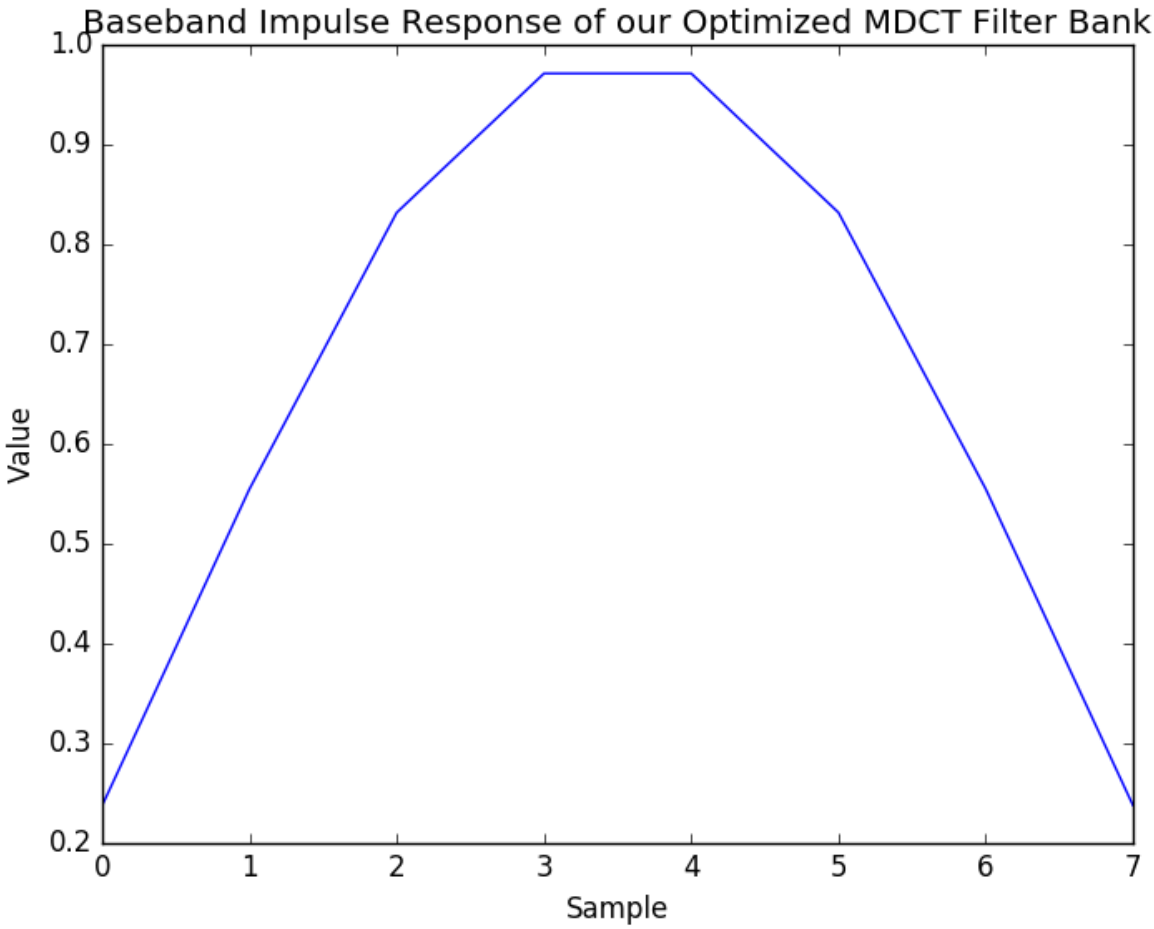
Python Example for the Optimization of an MDCT Filter Bank

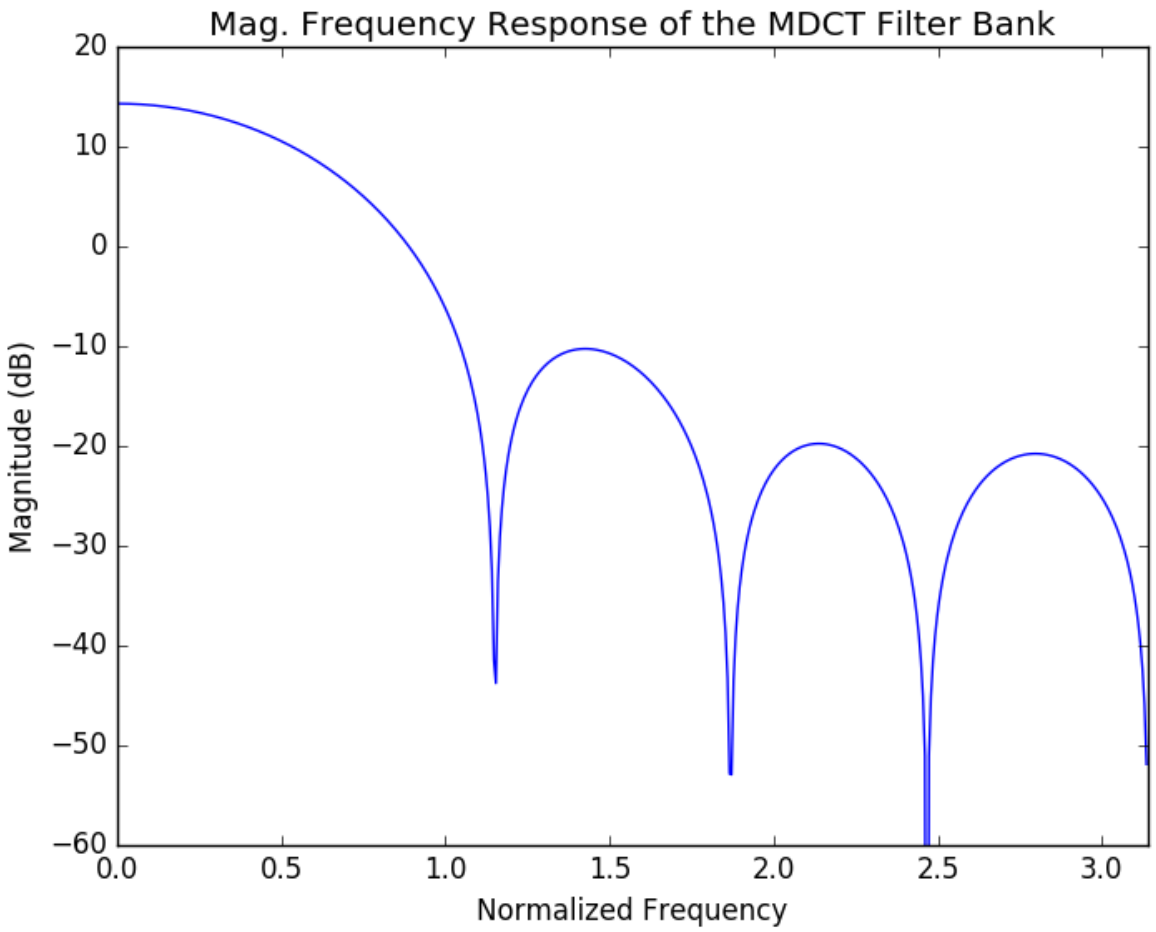
The function “optimfuncMDCT.py” contains an optimization of the baseband prototype, with

det(..)=-1, for a given number of subbands, set in the main part of the program, for instance N=8.
Start in the terminal shell with

```
python optimfuncMDCT.py
```

It starts with random coefficients, and prints their resulting error value. Then it optimizes the coefficients using “scipy.optimize.minimize” (it needs scipy version 0.19.1 or later), and after finishing the optimization prints out the resulting error value, and plots the resulting baseband prototype and its magnitude frequency response. For N=4 it is fairly fast. Observe that N=8 subbands already takes some time, N=16 takes a lot longer.
The resulting plots for N=4 are


Baseband Impulse Response of our Optimized MDCT Filter Bank
Value
Sample
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
1
2
3
4
5
6
7



Observe that the resulting baseband impulse response is similar to the sine window, but not exactly the same, it is “more optimal” in our optimization functions sense.