FFTnet

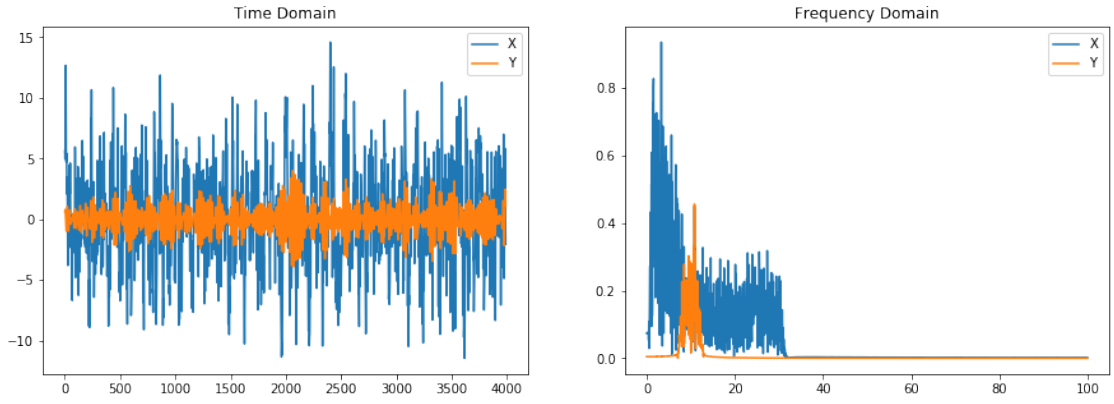
March 23, 2019

1 We can try to have a neural net 'Learn' the filters, that can be used to tranform X into Y. But as we'll see below, these filters will not be pure sine waves, but will have a lot of noise (with frequencies that are absent in the original signal

In [9]:
```
import numpy as np
import mne
import matplotlib.pyplot as plt

import scipy.io as sio
import tensorflow as tf
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)
from keras import backend as K
K.set_session(sess)
###################################

from keras.layers import Input, Dense, Conv2D, Conv1D, MaxPooling1D, UpSampling1D, Dropo
from keras.layers.advanced_activations import LeakyReLU
from keras import regularizers
# act = LeakyReLU(alpha=0.01)

from keras.models import Model, load_model
from keras.callbacks import TensorBoard, EarlyStopping, Callback
from keras import optimizers
from keras.initializers import Orthogonal as orth

def periodogram(fs, X):
    T = X.shape[0]/fs
    t = np.linspace(0.0, T, int(T/(1/fs)))
    yf = fft(X)
    xf = np.linspace(0.0, fs/2.0, int(len(yf)/2))
    yf = 2/(T*fs) * np.abs(yf[0:int(len(yf)/2)])
    return xf, yf
```

In [13]:
```
mat_contents = sio.loadmat('/home/amplifier/home/DATASETS/XY.mat')

X = mat_contents['X']
Y = mat_contents['Y']

X = X.reshape(1,32,3991,1)
Y = Y.reshape(1,32,3991,1)

fig = plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(np.squeeze(X[:,14,:,:]))
plt.plot(np.squeeze(Y[:,14,:,:]))
plt.title('Time Domain')
plt.legend(['X', 'Y'])

plt.subplot(1,2,2)
xf1, yf1 = periodogram(fs, X[:,14,:,:].flatten())
xf2, yf2 = periodogram(fs, Y[:,14,:,:].flatten())
plt.plot(xf1, yf1)
plt.plot(xf2, yf2)
plt.title('Frequency Domain')
plt.legend(['X', 'Y'])
```

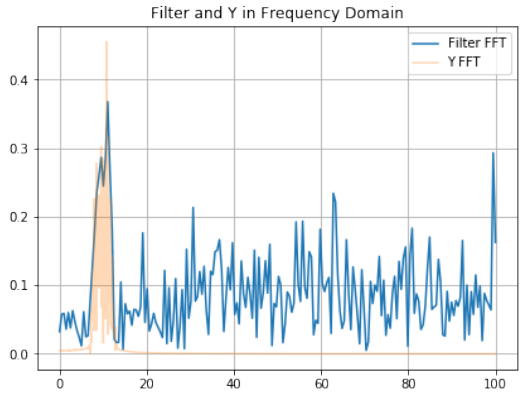
Out[13]:
```
<matplotlib.legend.Legend at 0x7f1d0a30fda0>
```



In [21]:
```
ker_width = 400
inpt = Input(shape=(32, 3991, 1))
l = Conv2D(1, (1,ker_width),
           use_bias=None,
           padding="same",
           activation=None)(inpt)
model = Model(inpt,l)
model.compile(loss='binary_crossentropy', optimizer='adam')
```

```
model.summary()

train_history = model.fit(X, Y, epochs=1000, verbose=0)
wts = np.array(model.layers[1].get_weights()).flatten()

fig = plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(wts)
plt.title('Filter in Time Domain')
plt.grid()

plt.subplot(1,2,2)
plt.title('Filter and Y in Frequency Domain')
xf1, yf1 = periodogram(fs, wts)
plt.plot(xf1, 20*yf1)
xf2, yf2 = periodogram(fs, Y[:,14,:,:].flatten())
plt.plot(xf2, yf2, alpha=1.3)
plt.legend(['Filter FFT', 'Y FFT'])
plt.grid()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_8 (InputLayer)         (None, 32, 3991, 1)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 32, 3991, 1)       400
=================================================================
Total params: 400
Trainable params: 400
Non-trainable params: 0
_________________________________________________________________
```



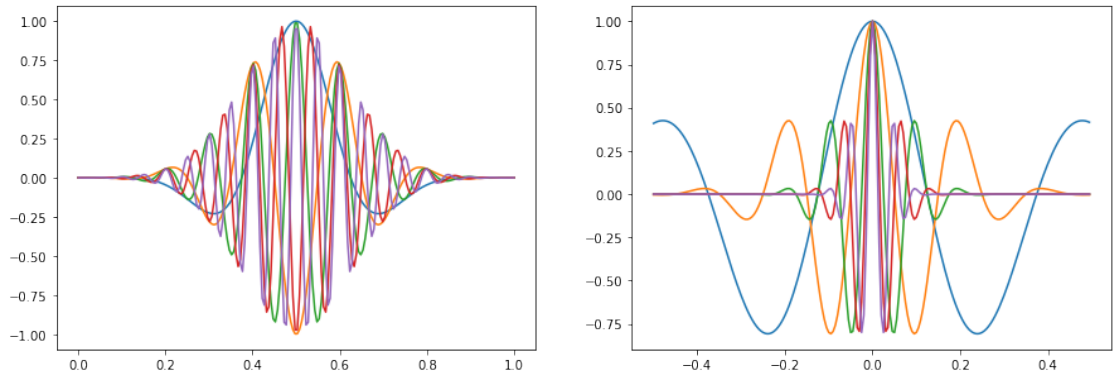
2 Alternatively, we can have the model learn NOT the filters directly (when these filters start to include irrelevant frequencies that don't 'resonate' with the frequencies in the input signal, because these aren't there), but have it learn the filter parameter (i.e. frequency of the sinusoid, or Morlet wavelet). In that case we can get pure (and easily interpretable) filters. The results of this filtering can be combined downstream and/or fed into a FC layer for regression or classification. Keras, as far as I know, can't do it, but PyTorch can:

In [29]:
```
import torch
from torch.autograd import Variable
import torch.optim as optim
import torch.nn.functional as F
from scipy import signal
from scipy.fftpack import fft
import scipy.io as sio
import numpy as np
import matplotlib.pyplot as plt

def gaussian(x, mu, sig):
    return np.exp(-np.power(x - mu, 2.) / (2 * np.power(sig, 2.)))

mat_contents = sio.loadmat('/home/amplifier/home/DATASETS/XY.mat')
X = mat_contents['X']
Y = mat_contents['Y']

X = X[:,0:1000]
Y = Y[:,0:1000]
```

In [30]:
```
fs = 200
f = Variable(torch.tensor([2.0, 5.0, 10.0, 15.0, 20.0]), requires_grad=True)
print(f)
pi = torch.tensor(np.pi)
T = 1
t = torch.tensor(np.linspace(0.0, T, int(T/(1/fs)), dtype='float32'))


x_values = np.linspace(-4, 4, 200)
g = torch.tensor(gaussian(x_values, 0, 1)).float()

k = []
for i in range(5):
    k.append(g * torch.cos(2.0*pi*f[i]*t))

fig = plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
```

```
        for i in range(5):
            plt.plot(t.detach().numpy(), k[i].detach().numpy())

        t = np.linspace(-T/2, T/2, T * fs, endpoint=False) # sampled @ 200 Hz for 0.5 s
        I = []
        k_m = []
        for j in range(5):
            i, q, e = signal.gausspulse(t, f[j].detach().numpy(), retquad=True, retenv=True)
            I.append(i)
            k_m.append(torch.tensor(i).float())

        plt.subplot(1,2,2)
        for i in range(5):
            plt.plot(t, I[i])
```

```
tensor([ 2.,  5., 10., 15., 20.], requires_grad=True)
```

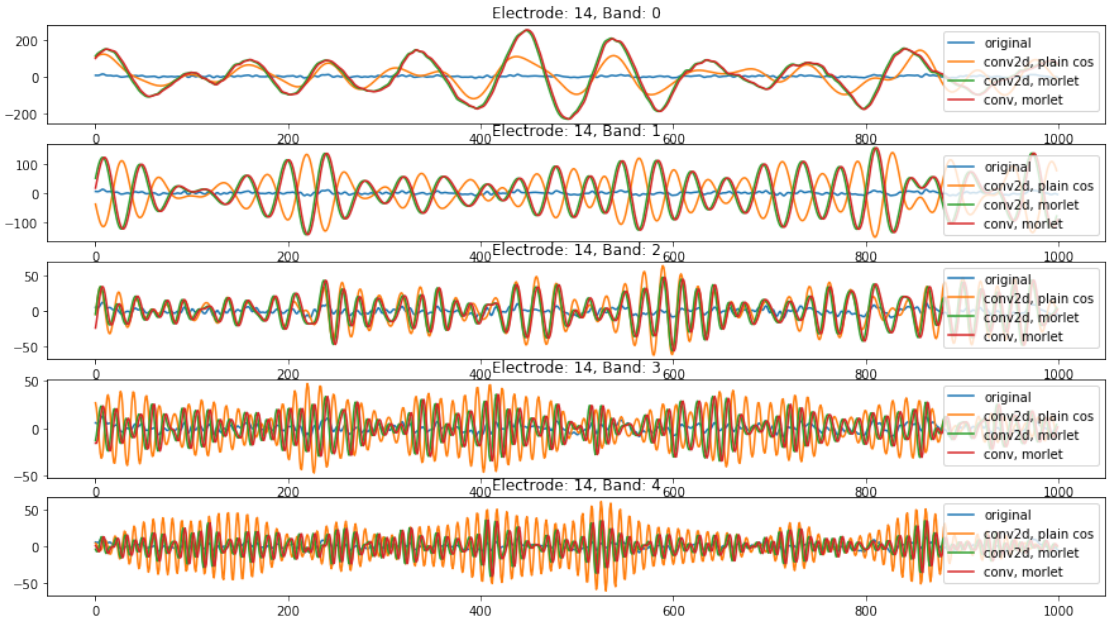


In [31]:
```
def forward(inputs, f, pi):
    fs = 200
    T = 1
    t = torch.tensor(np.linspace(0.0, T, int(T/(1/fs)), dtype='float32'))
    x_values = np.linspace(-4, 4, 200)
    g = torch.tensor(gaussian(x_values, 0, 1)).float()
    k = []
    for i in range(5):
        k.append(g * torch.cos(2.0*pi*f[i]*t))
    filters = torch.stack(k).view(5,1,1,200)
    x = F.conv2d(inputs, filters, padding=(0,padding))
    return x, filters
```

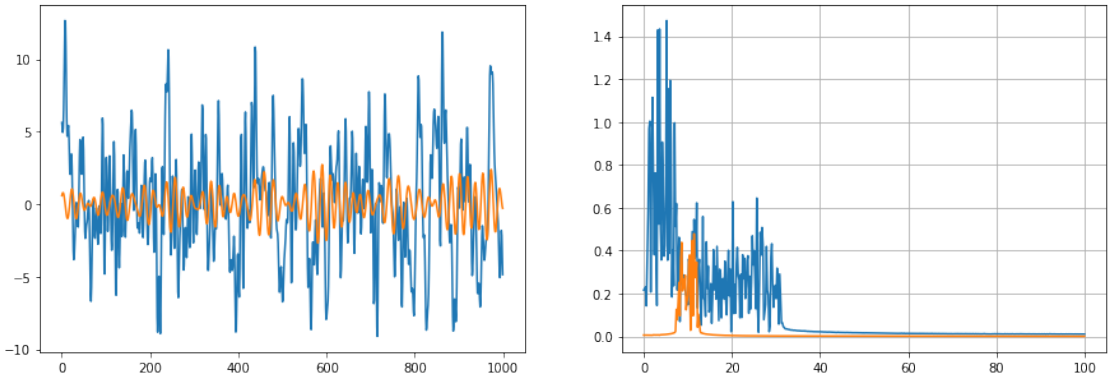
In [32]:
```
signal_len = X.shape[1]
n_eeg_chans = X.shape[0]
ker_width = 200
```

```
        padding = 99
        n_filters = 5
        electrode = 14

        inputs = torch.tensor(X).view(1, 1, n_eeg_chans, signal_len) # batch,        in_channel

        # GET FILTERED SIGNAL AND FILTERS BASED ON SET FREQUENCIES AND USING MY "MORLET" THAT S
        x, filters = forward(inputs, f, pi)

        # GET FILTERED SIGNAL AND FILTERS BASED ON SET FREQUENCIES AND USING MORLET THAT DOESN'
        filters_m = torch.stack(k_m).view(5,1,1,200)
        x_m = F.conv1d(inputs, filters_m, padding=(0,padding))

        # GET FILTERED SIGNAL USING NUMPY'S CONV. JUST TO SEE THAT IT WORKS THE SAME AS TORCH.M
        x_c = []
        for i in range(n_filters):
            x_c.append(np.convolve(X[electrode,:], filters_m[i,0,0,:].detach().numpy().flatten(
        x_c = np.vstack(x_c)

        print('inputs.shape =\t', inputs.shape)
        print('filters.shape =\t' , filters.shape)
        print('filters_m.shape =\t' , filters_m.shape)
        print('x.shape =\t', x.shape)
        print('x_m.shape =\t', x_m.shape)
        print('x_c.shape =\t', x_c.shape)

        fig = plt.figure(figsize=(15,10))
        for band in range(0, filters.shape[0]):
            plt.subplot(filters.shape[0]+1, 1, band+1)
            plt.plot(X[electrode,:])
            plt.plot(x[0,band,electrode,:].detach().numpy())
            plt.plot(x_m[0,band,electrode,:].detach().numpy())
            plt.plot(x_c[band])
            plt.legend(['original', 'conv2d, plain cos', 'conv2d, morlet', 'conv, morlet'])
            plt.title('Electrode: ' + str(electrode) + ', Band: ' + str(band))
```

```
inputs.shape =         torch.Size([1, 1, 32, 1000])
filters.shape =         torch.Size([5, 1, 1, 200])
filters_m.shape =         torch.Size([5, 1, 1, 200])
x.shape =         torch.Size([1, 5, 32, 999])
x_m.shape =         torch.Size([1, 5, 32, 999])
x_c.shape =         (5, 1000)
```



In [33]:
```
fs = 200.0
xf1, yf1 = periodogram(fs, X[electrode,:])
xf2, yf2 = periodogram(fs, Y[electrode,:])

fig = plt.figure(figsize=(15,5))

plt.subplot(1,2,1)
plt.plot(X[electrode,:])
plt.plot(Y[electrode,:])

plt.subplot(1,2,2)
plt.plot(xf1, yf1)
plt.plot(xf2, yf2)
plt.grid()
```

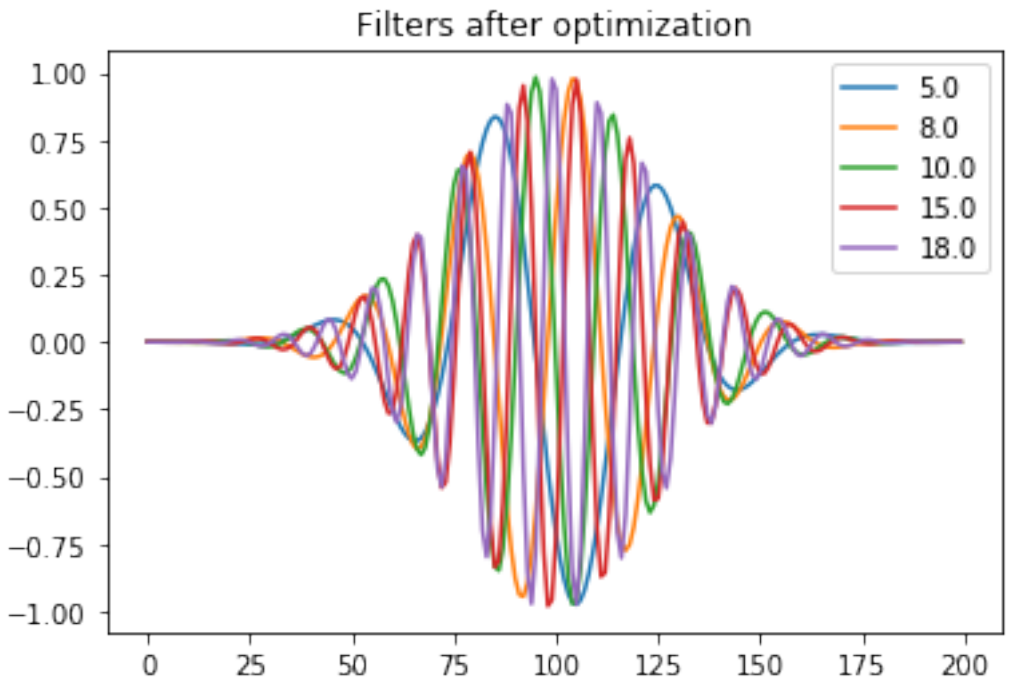


```
In [45]: optimizer = optim.Adam([f]) # tell the optimizer which var we want optimized
         criterion = torch.nn.MSELoss() # set loss type

         x, filters = forward(inputs, f, pi) # forward the model to get filtered signal and filt

         predictions = torch.sum(x, 1).view(32,-1)
         targets = torch.tensor(Y[:,0:999]).float()
         loss = criterion(predictions, targets) # compute loss

         print('predictions: \trequires_grad?\t', predictions.requires_grad)
         print('targets: \trequires_grad?\t', targets.requires_grad)
         print('loss: \t\trequires_grad?\t', loss.requires_grad)

predictions:         requires_grad?         True
targets:         requires_grad?         False
loss:                requires_grad?         True


In [44]: for epoch in range(5000):
             x, filters = forward(inputs, f, pi)
             predictions = torch.sum(x, 1).view(32,-1)
             loss = criterion(predictions, targets)
             loss.backward(retain_graph=True)
             optimizer.step()    # update the variable being optimized (f)
             optimizer.zero_grad()
             if epoch % 100 == 0:
                 print('loss:\t', np.round(loss.item(), 1), '\tfreqs:\t', f)

loss:         8164.0         freqs:         tensor([ 2.0010,  4.9990, 10.0010, 15.0010, 19.9990]
loss:         8115.3         freqs:         tensor([ 2.0558,  4.9871, 10.0521, 15.0697, 19.9084]
loss:         8080.7         freqs:         tensor([ 2.1268,  5.0512, 10.0597, 15.0460, 19.8152]
loss:         8014.2         freqs:         tensor([ 2.2251,  5.1467, 10.0657, 14.9872, 19.6710]
loss:         7952.5         freqs:         tensor([ 2.3136,  5.2366, 10.0706, 14.9217, 19.5093]
loss:         7910.7         freqs:         tensor([ 2.3792,  5.3039, 10.0713, 14.8631, 19.3543]
loss:         7882.6         freqs:         tensor([ 2.4243,  5.3497, 10.0687, 14.8162, 19.2097]
loss:         7860.5         freqs:         tensor([ 2.4543,  5.3798, 10.0673, 14.7886, 19.0714]
loss:         7832.6         freqs:         tensor([ 2.4749,  5.4010, 10.0841, 14.8265, 18.9190]
loss:         7376.0         freqs:         tensor([ 2.5026,  5.4328, 10.2078, 15.2148, 18.6048]
loss:         6906.4         freqs:         tensor([ 2.5501,  5.4818, 10.2731, 15.4376, 18.3694]
loss:         6872.7         freqs:         tensor([ 2.5943,  5.5257, 10.2933, 15.4099, 18.3027]
loss:         6847.4         freqs:         tensor([ 2.6335,  5.5639, 10.3042, 15.3609, 18.2511]
loss:         6826.1         freqs:         tensor([ 2.6700,  5.5989, 10.3108, 15.3145, 18.2019]
loss:         6808.4         freqs:         tensor([ 2.7057,  5.6325, 10.3155, 15.2720, 18.1561]
loss:         6793.9         freqs:         tensor([ 2.7426,  5.6670, 10.3198, 15.2338, 18.1144]
loss:         6782.0         freqs:         tensor([ 2.7830,  5.7045, 10.3256, 15.2007, 18.0775]
loss:         6771.5         freqs:         tensor([ 2.8301,  5.7483, 10.3348, 15.1731, 18.0460]
```

```
loss:         6761.1         freqs:         tensor([ 2.8877,  5.8017, 10.3492, 15.1516, 18.0204]
loss:         6749.1         freqs:         tensor([ 2.9583,  5.8678, 10.3703, 15.1369, 18.0014]
loss:         6734.8         freqs:         tensor([ 3.0403,  5.9454, 10.3977, 15.1295, 17.9897]
loss:         6720.1         freqs:         tensor([ 3.1258,  6.0274, 10.4280, 15.1294, 17.9854]
loss:         6707.5         freqs:         tensor([ 3.2056,  6.1049, 10.4568, 15.1350, 17.9879]
loss:         6697.8         freqs:         tensor([ 3.2749,  6.1730, 10.4809, 15.1445, 17.9953]
loss:         6690.7         freqs:         tensor([ 3.3346,  6.2321, 10.4989, 15.1556, 18.0054]
loss:         6684.9         freqs:         tensor([ 3.3884,  6.2858, 10.5065, 15.1659, 18.0159]
loss:         6678.1         freqs:         tensor([ 3.4466,  6.3460, 10.4606, 15.1683, 18.0226]
loss:         5824.6         freqs:         tensor([ 3.7322,  6.6993,  9.8938, 15.0856, 17.9837]
loss:         5418.9         freqs:         tensor([ 3.9839,  6.8959,  9.8263, 14.9938, 17.8926]
loss:         5294.1         freqs:         tensor([ 4.1002,  7.0098,  9.9298, 14.9244, 17.8160]
loss:         5218.9         freqs:         tensor([ 4.1961,  7.1026, 10.0085, 14.8854, 17.7663]
loss:         5176.6         freqs:         tensor([ 4.2719,  7.1757, 10.0691, 14.8732, 17.7452]
loss:         5151.1         freqs:         tensor([ 4.3319,  7.2338, 10.1180, 14.8797, 17.7459]
loss:         5133.8         freqs:         tensor([ 4.3809,  7.2815, 10.1592, 14.8973, 17.7599]
loss:         5121.0         freqs:         tensor([ 4.4222,  7.3218, 10.1951, 14.9204, 17.7810]
loss:         5111.0         freqs:         tensor([ 4.4581,  7.3570, 10.2270, 14.9454, 17.8049]
loss:         5103.1         freqs:         tensor([ 4.4901,  7.3884, 10.2558, 14.9704, 17.8291]
loss:         5096.8         freqs:         tensor([ 4.5189,  7.4166, 10.2821, 14.9942, 17.8523]
loss:         5091.7         freqs:         tensor([ 4.5451,  7.4423, 10.3062, 15.0163, 17.8741]
loss:         5087.7         freqs:         tensor([ 4.5691,  7.4657, 10.3284, 15.0367, 17.8940]
loss:         5084.4         freqs:         tensor([ 4.5910,  7.4871, 10.3487, 15.0553, 17.9122]
loss:         5081.9         freqs:         tensor([ 4.6111,  7.5067, 10.3675, 15.0722, 17.9288]
loss:         5079.8         freqs:         tensor([ 4.6295,  7.5246, 10.3847, 15.0876, 17.9438]
loss:         5078.2         freqs:         tensor([ 4.6464,  7.5409, 10.4005, 15.1015, 17.9574]
loss:         5076.9         freqs:         tensor([ 4.6618,  7.5559, 10.4150, 15.1142, 17.9697]
loss:         5075.9         freqs:         tensor([ 4.6758,  7.5695, 10.4283, 15.1257, 17.9808]
loss:         5075.1         freqs:         tensor([ 4.6886,  7.5819, 10.4405, 15.1361, 17.9909]
loss:         5074.5         freqs:         tensor([ 4.7003,  7.5932, 10.4515, 15.1455, 18.0000]
loss:         5074.0         freqs:         tensor([ 4.7109,  7.6035, 10.4616, 15.1540, 18.0082]
loss:         5073.6         freqs:         tensor([ 4.7205,  7.6128, 10.4708, 15.1617, 18.0156]
```

In [65]:
```
fs = 200
pi = torch.tensor(np.pi)
_, filters = forward(inputs, f, pi)

for i in range(5):
    filt = filters[i,:,:,:].detach().numpy().flatten()
    plt.plot(filt)
plt.legend(f.detach().numpy().round(decimals=0).tolist())
plt.title('Filters after optimization')
```

Out[65]:
```
Text(0.5, 1.0, 'Filters after optimization')
```



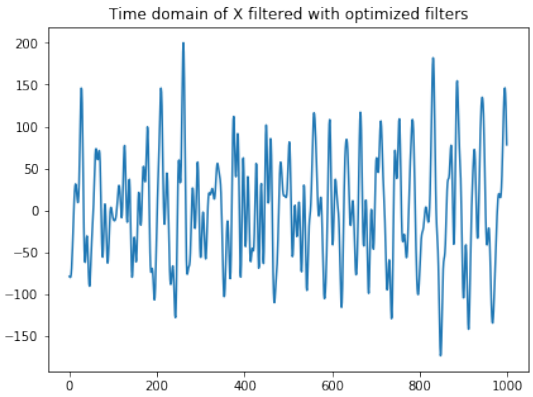
In [73]:
```
Y.shape
```

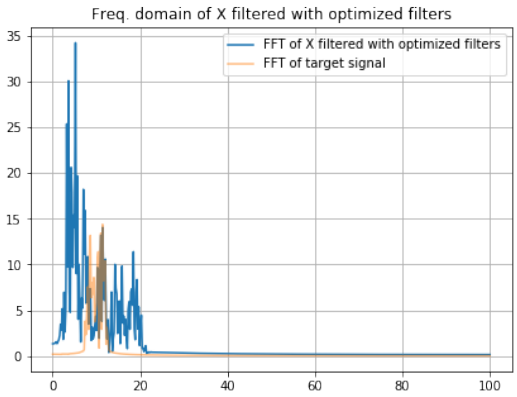
Out[73]:
```
(32, 1000)
```

In [ ]:
```
```

In [78]:
```
x_ = x[:,:,electrode,:].sum(1).detach().numpy().flatten()

fig = plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.plot(x_)
plt.title('Time domain of X filtered with optimized filters')

plt.subplot(1,2,2)
xf1, yf1 = periodogram(fs, x_)
plt.plot(xf1, yf1)
xf2, yf2 = periodogram(fs, Y[electrode,:].flatten())
plt.plot(xf2, 30*yf2, alpha=0.5)
plt.grid()
plt.title('Freq. domain of X filtered with optimized filters')
plt.legend(['FFT of X filtered with optimized filters', 'FFT of target signal'])
```

Out[78]:
```
<matplotlib.legend.Legend at 0x7f1bec0f66a0>
```





3 As we can see, this approach can't fit the target signal very well, but perhaps, if we add weights to each kernel (perhaps an FC layer) we would be able to achieve optimal filters

In [ ]: