Plano de aula

Pensei em tratarmos de um ECG já que temos pouco tempo, a base do código é simples e aplicação direto dos conhecimentos que iremos tratar na semana universitária.

O primeiro passo vai ser desenvolver uma função de downsample com o pessoal. Ela é bem simples tratando de vetores e tamanho de vetores (só explicar a teoria dos arranjos de filtros tratada pelo professor):

```
import numpy as np


def downsample(x, n = [], M = 2):
    if len(n) == 0:
        n = np.arange(0, len(x))
    k = [n[i] == 0 for i in range(0, len(n))]
    k = np.where(k)[0][0]
    x_right = x[k :]
    y_right = x_right[: len(x_right) : M]
    x_left = x[: k + 1]
    x_left = x_left[::-1]
    y_left = x_left[: len(x_left) : M]
    y_left = y_left[::-1]
    y_left = y_left[: len(y_left) - 1]
    n_ = np.arange(-len(y_left), len(y_right))
    y = np.concatenate((y_left, y_right))
    return y, n_
```

Feita a função, podemos tratar agora das interações dos filtros para a nossa decomposição, lembrando dos conhecimentos sobre eles das funções que tratamos em sala (essa parte dá uma boa revisada com a galera em como é feita a reconstrução e como o teorema tem que ser aplicado para garantir que o upsample retorne a entrada):

```
# filter_iterator.py

# map <leader><leader> :wall<cr>:!python %<cr>
```

```python
from copy import deepcopy
from matplotlib import pyplot as plt
import numpy as np
from qmf_filters_validator import qmf_filters_validator
from scipy.io import loadmat
from upsample import upsample


def font_configuration():
    try:
        plt.rcParams.update({
            "text.usetex": True,
            "font.family": "serif",
            "font.serif": ["Palatino"],
            "font.size": 25,
        })
    except:
        plt.rcParams.update({
            "text.usetex": False,
            "font.family": "serif",
            "font.serif": ["Palatino"],
            "font.size": 16,
        })


def filter_iterator(x0, x1, A, d, levels = 4): # valid both for analysis and synthesis
    x = [0] * (levels + 1)
    x0_ = deepcopy(x0)
    x[levels] = x1
    for k in range(levels - 1, 0, -1):
        x[k] = np.convolve(upsample(x[k + 1], 2), x0)
        x0_ = np.convolve(upsample(x0_, 2), x0)
    x[0] = x0_
    T = np.arange(levels + 1, 0, -1)
    T[0] = levels
    multirate_factors = deepcopy(T)
    multirate_factors = 2 ** multirate_factors
    rescale_factors = deepcopy(T)
    rescale_factors = 1.0 / (A ** T)
    advance_values = deepcopy(T)
    advance_values = d * ((2 ** T) - 1)
    return x, multirate_factors, rescale_factors, advance_values


if __name__ == '__main__':
    M = loadmat('wfilters.mat')
    chosen_wfilter = 44
    h0 = M['h0']
    h1 = M['h1']
    g0 = M['g0']
    g1 = M['g1']
    h0 = h0[0, chosen_wfilter][0]
    h1 = h1[0, chosen_wfilter][0]
    g0 = g0[0, chosen_wfilter][0]
    g1 = g1[0, chosen_wfilter][0]
    _, _, _, A, d = qmf_filters_validator(h0, h1, g0, g1)
    h, multirate_factors, rescale_factors, advance_values = filter_iterator(h0, h1, A = A, d = d,
levels = 10)
    font_configuration()
    for k in range(len(h) - 1, -1, -1):
        plt.plot(h[k])
        plt.show()
    indices = "1"
    final_indice = "0"
    for k in range(len(h) - 1, -1, -1):
        H = np.fft.fft(h[k], 1000000)
        f = np.linspace(-0.5, 0.5, len(H))
        label = '${|H_{' + indices + '}(f)|}$'
        indices = "0" + indices
        if k == 1:
            indices = final_indice
        else:
            final_indice = "0" + final_indice
        plt.plot(f, np.abs(np.fft.fftshift(H)), label = label)
    plt.xlim((0, 0.5))
    plt.legend(fontsize = 25.0)
    plt.show()
    print(len(h))
```

Com isso terminamos a parte de decomposição e podemos rodar o código abaixo com o sinal de ECG1 e os filtros de Daubechies5. (Lembre-se que você tem duas opções de abordagem, a primeira é fazer todo esse código em 1 grande, a segunda é botar tudo em uma pasta e ficar chamando como se fossem funções independentes no código. Como no código do professor ele fez a segunda opção, vou colocar assim, porém caso queira trocar é só comentar a parte do código que importa a função no início):

```
# multivel_multirate_decomposition.py
# map <leader><leader> :wall<cr>:!python %<cr>


from downsample import downsample
from filter_iterator import filter_iterator
from matplotlib import pyplot as plt
import numpy as np
from scipy.io import loadmat


def unite(x):
    N = 0
    for k in range(0, len(x)):
        N += len(x[k])
    y = np.zeros(shape = (N,))
    n = 0
    for k in range(0, len(x)):
        y[n : n + len(x[k])] = x[k]
        n += len(x[k])
    return y


def multivel_multirate_decomposition(x, h0, h1, A, d, levels = 4):
    h, multirate_factors, _, _ = filter_iterator(h0, h1, A, d, levels = levels)
    x_decomp = [0] * len(h)
    for k in range(0, len(h)):
        x_ = np.convolve(h[k], x)
        x_, _ = downsample(x_, M = multirate_factors[k])
        x_decomp[k] = x_
    x_hat = unite(x_decomp)
    return x_hat, x_decomp


def font_configuration():
    try:
        plt.rcParams.update({
            "text.usetex": True,
            "font.family": "serif",
            "font.serif": ["Palatino"],
            "font.size": 26,
        })
    except:
        plt.rcParams.update({
            "text.usetex": False,
            "font.family": "serif",
            "font.serif": ["Palatino"],
        "font.size": 16,
      })

if __name__ == '__main__':
    font_configuration()
    M = loadmat('ECG_1.mat')
    x = M['x']
    x = x[:, 0]
    x = x - np.mean(x)
    fs = M['fs']
    t = np.zeros(shape = x.shape)
    t[0 : len(x)] = np.arange(0, len(x)) * 1.0 / fs
    fig = plt.figure()
    ax = fig.add_subplot(111)
    xrange = np.max(t) - np.min(t)
    yrange = np.max(x) - np.min(x)
    # ax.set_aspect(5.0 * xrange / yrange / 8.0)
    plt.tight_layout()
    plt.plot(t, x)
    plt.xlabel('Time ${t}$ (seconds)')
    plt.ylabel('${x_c(t)}$', rotation = 0.0, labelpad = 20)
    plt.grid()
    plt.show()
    M = loadmat('wfilters.mat')
    chosen_wfilter = 4
    h0 = M['h0']
    h1 = M['h1']
    g0 = M['g0']
    g1 = M['g1']
    h0 = h0[0, chosen_wfilter][0]
    h1 = h1[0, chosen_wfilter][0]
    g0 = g0[0, chosen_wfilter][0]
    g1 = g1[0, chosen_wfilter][0]
    # h0 = np.array([1, 1])
    # h1 = np.array([1, -1])
    # g0 = np.array([1, 1])
    # g1 = np.array([-1, 1])
    x_hat, x_decomp = multivel_multirate_decomposition(x, h0, h1, levels = 4)
    print(x_hat)
    n = np.arange(0, len(x_hat))
    fig = plt.figure()
    ax = fig.add_subplot(111)
    xrange = np.max(n) - np.min(n)
    yrange = np.max(x_hat) - np.min(x_hat)
    # ax.set_aspect(5.0 * xrange / yrange / 8.0)
    plt.tight_layout()
    plt.plot(x_hat)
    plt.xlabel('Índice da transformada k')
    plt.ylabel('${\hat{x}[k]}$', rotation = 0.0, labelpad = 20)
    plt.grid()
    plt.show()
    print(len(x))
    print(len(x_hat))
```

Agora para a parte de reconstrução, precisamos fazer uma função que valida os filtros escolhidos para ver se, ao utilizá-los, a reconstrução será correta e a entrada e a saída são iguais. (essa parte vai demandar tempo de teoria pra verificar os teoremas):

```
# map <leader><leader> :wall<cr>:!python %<cr>


from copy import deepcopy
import numpy as np
from scipy.io import loadmat
from signal_sum import signal_sum
```

```
def filter_change_z_signal(h):
    h_ = deepcopy(h)
    h_[1 : :2] = -h_[1 : :2]
    return h_

def locations_null_coefficients(v, tol):
    v_null = [np.abs(v[i]) <= tol for i in range(0, len(v))]
    k = np.where(v_null)[0]
    return k

def qmf_filters_validator(h0, h1, g0, g1, tol = 1e-6):
    # Check the aliasing term
    # We need 1/2 * (H0(-z)G0(z) + H1(-z)G1(z)) = 0.
    h0_ = filter_change_z_signal(h0)
    h1_ = filter_change_z_signal(h1)
    h0_g0 = np.convolve(h0_, g0)
    h1_g1 = np.convolve(h1_, g1)
    aliasing_term = 0.5 * signal_sum(h0_g0, h1_g1)
    k = locations_null_coefficients(aliasing_term, tol)
    if len(k) < len(aliasing_term):
        aliasing_term_null = False
    else:
        aliasing_term_null = True
    # Check the LTI term
    # We need 1/2 * (H0(z)G0(z) + H1(z)G1(z)) = Az^(-d).
    h0_g0 = np.convolve(h0, g0)
    h1_g1 = np.convolve(h1, g1)
    lti_term = 0.5 * signal_sum(h0_g0, h1_g1)
    k = locations_null_coefficients(lti_term, tol)
    d = np.nan
    A = np.nan
    if len(k) == len(lti_term) - 1:
        lti_term_valid = True
        n = np.arange(0, len(lti_term))
        d = np.setdiff1d(n, k)[0]
        print("Atraso:")
        print(d)
        A = lti_term[d]
    else:
        lti_term_valid = False
    valid_filters = lti_term_valid and aliasing_term_null
    return valid_filters, lti_term_valid, aliasing_term_null, A, d


if __name__ == '__main__':
    h0 = np.array([1, 1])
    h1 = np.array([1, -1])
    g0 = np.array([1, 1])
    g1 = np.array([-1, 1])
    valid_filters, lti_term_valid, aliasing_term_null, A, d = \
    qmf_filters_validator(h0, h1, g0, g1)
    print(valid_filters)
    print(lti_term_valid)
    print(aliasing_term_null)
    print(A)
    print(d)
    M = loadmat('wfilters.mat')
    print(M.keys())
    chosen_wfilter = 44
    print(M['filters_families'][0,chosen_wfilter])
    h0 = M['h0']
    h1 = M['h1']
    g0 = M['g0']
    g1 = M['g1']
    h0 = h0[0, chosen_wfilter][0]
    h1 = h1[0, chosen_wfilter][0]
    g0 = g0[0, chosen_wfilter][0]
    g1 = g1[0, chosen_wfilter][0]
    valid_filters, lti_term_valid, aliasing_term_null, A, d = \
    qmf_filters_validator(h0, h1, g0, g1, tol = 1e-4)
    print(valid_filters)
    print(lti_term_valid)
    print(aliasing_term_null)
    print(A)
    print(d)
```

Aqui o signal_sum mencionado:

```
import numpy as np


def signal_sum(x, y):
    if len(y) <= len(x):
        ya = np.zeros(shape = x.shape)
        ya[: len(y)] = y
        return x + ya
    else:
        return signal_sum(y, x)
```

Com o validador feito, prossiga com a função de upsample:

```
import numpy as np


def upsample(x, M = 2, remove_leading_zeros = True):
    y = np.zeros(shape = (x.shape[0] * M, ))
    y[0:y.shape[0]:M] = x
    if remove_leading_zeros:
        y = y[: len(y) - (M - 1)]
    return y
```

Agora para finalizar o código para reconstrução que chama todas as funções trabalhadas nesse arquivo:

```
# multivel_multirate_decomposition.py
# map <leader><leader> :wall<cr>:!python %<cr>

from copy import deepcopy
from extract_filters import extract_filters
from filter_iterator import filter_iterator
from multivel_multirate_decomposition import multivel_multirate_decomposition
from matplotlib import pyplot as plt
import numpy as np
from plot_signal import plot_signal
from qmf_filters_validator import qmf_filters_validator
from scipy.io import loadmat
from signal_sum import signal_sum
from upsample import upsample

def multivel_multirate_reconstruction(x_decomp, g0, g1, A, d):
    g, multirate_factors, rescale_factors, advance_values = filter_iterator(g0, g1, A, d, levels = len(x_decomp) - 1)
    xd = deepcopy(x_decomp)
    xr = np.zeros(shape = [1, ])
    for k in range(0, len(xd)):
        xd[k] = upsample(xd[k], M = multirate_factors[k])
        xd[k] = np.convolve(g[k], xd[k])
        xd[k] *= rescale_factors[k]
        x_ = xd[k]
        xd[k] = x_[advance_values[k]:]
        xr = signal_sum(xr, xd[k])
    return xr


def font_configuration():
    try:
        plt.rcParams.update({
            "text.usetex": True,
            "font.family": "serif",
            "font.serif": ["Palatino"],
            "font.size": 26,
        })
    except:
        plt.rcParams.update({
            "text.usetex": False,
            "font.family": "serif",
            "font.serif": ["Palatino"],
            "font.size": 16,
        })


if __name__ == '__main__':
    M = loadmat('ECG_1.mat')
    x = M['x']
    x = x[:, 0]
    x = x
    fs = M['fs']
    plot_signal(x, fs, xlabel = 'Tempo t (segundos)', ylabel = 'x_c(t)')
    h0, h1, g0, g1 = extract_filters(4)
    _, _, _, A, d = qmf_filters_validator(h0, h1, g0, g1)
    x_hat, x_decomp = multivel_multirate_decomposition(x, h0, h1, A, d, levels = 5)
    plot_signal(x_hat, 1, xlabel = 'Índice da transformada k', ylabel = 'x_hat[k]')
    xr = multivel_multirate_reconstruction(x_decomp, g0, g1, A, d)
    plot_signal(xr, fs)
```

```
err = xr[:len(x)] - x

plot_signal(err, fs)

print(len(x))

print(len(xr))

print(len(h0))

print(len(h1))

print(len(g0))

print(len(g1))
```