Лабораторная работа №4

Построение вейвлетов Добеши и их графиков

Вариант 4 - 1
Выполнил Казачинский Глеб, 3 курс 6 группа

Постановка задачи

Дан тип ортономированного вейвлета Добеши с нулевыми моментами. Используя процедуру спектральной факторизации, вычислить коэффициенты двухмасштабного соотношения (ДМС) $\{h_n\}$. Построить график построенной масштабирующей функции и вейвлета, используя указанный в варианте способ.

Для осуществления спектральной факторизации разрешается использовать ПО для символьных вычислений (Mathematica, Maple, Matlab и т. п.)

Содержание отчета:

- Подробное описание процесса получения коэффициентов ДМС с приведением кода на используемом для этого языке.
- Сравнение полученных коэффициентов с приведенными в [Добеши].
- В силу неединственности спектральной факторизации также приветствуется получение одного-двух других вариантов ДМС (а также графиков соответствующих МФ и вейвлета), отличных от классических.
- Графики масштабирующей функции и вейвлета.
- Описание алгоритма, с помощью которого были построены графики.
- Код всех программ.

- Подробное описание процесса получения коэффициентов ДМС с приведением кода на используемом для этого языке.

Приведём код программы, основной функцией которой является *get_daubechies_coef(N, a_N),* принимающая на вход N и a_N(a_N сложно вычислять для любого N в программе, приходится считать a_N отдельно, в вольфраме) и отдающая на выход коэффициенты ДМС.

```
import numpy as np
import scipy.special as scp
import numpy.polynomial.polynomial as poly
from numpy import linalg as LA


def Q(N):
    polynom_coefs = np.zeros(N)
    for k in range(N):
        polynom_coefs[k] = scp.binom(N - 1 + k, k)
    return poly.Polynomial(polynom_coefs)


def U(N, Q):
    result_poly = 0
    for i in range(N):
        result_poly += Q.coef[i] * poly.Polynomial([1 / 2, -1 / 2]) ** i
    return result_poly


def z_k(U_roots, N, type):
    ret_value = []
    for k in range(len(U_roots)):
        ret_value.append(U_roots[k] + np.sqrt(U_roots[k] ** 2 - 1))
        ret_value.append(U_roots[k] - np.sqrt(U_roots[k] ** 2 - 1))
    print(ret_value)
    count = 0
    z_k = []
    z_k_blacklist = []
    if type == 'complex':
        for i in range(len(ret_value)):
            if not np.isin(np.abs(z_k_blacklist), np.abs(ret_value[i])) and ret_value[i].real < 1 and ret_value[
                    i].imag < 1:
                z_k.append(ret_value[i])
```

```
                z_k_blacklist.append(ret_value[i])
                count += 1
            if count == N:
                break
    if type == 'real':
        for i in range(len(ret_value)):
            if ret_value[i].imag == 0 and np.abs(ret_value[i]) <= 1:
                z_k.append(ret_value[i])
                count += 1
            if count == N:
                break
    return z_k


def B_1(N_1, z_k):
    r_k = z_k
    ret_value = poly.Polynomial([1])
    for k in range(1, N_1 + 1):
        ret_value *= (poly.Polynomial([- r_k[k - 1], 1])) / np.sqrt(np.abs(r_k[k - 1]))
    return ret_value


def B_2(N_2, z_k):
    cos_alpha_k = map(lambda x: x.real, z_k)  # TODO
    ret_value = poly.Polynomial([1])
    for k in range(1, N_2 + 1):
        ret_value *= poly.Polynomial([1, - 2 * cos_alpha_k[k - 1], 1])
    return ret_value


def B_3(N_3, z_k):
    ret_value = poly.Polynomial([1])
    for k in range(1, N_3 + 1):
        ret_value *= (poly.Polynomial([np.abs(z_k[k - 1]) ** 2, - 2 * z_k[k - 1].real, 1]) / np.abs(z_k[k - 1]))
    return ret_value


def B(a_N, B_1=1, B_2=1, B_3=1):
    return np.sqrt(np.abs(a_N) / 2) * B_1 * B_2 * B_3


def M_0(N, B):
    return poly.Polynomial([1 / 2, 1 / 2]) ** N * B


def get_N1_N2_N3(U_roots):
    N_1, N_2, N_3 = 0, 0, 0
    for i in range(len(U_roots)):
```

```python
        if isinstance(U_roots[i], complex) and U_roots[i].imag != 0:
            N_3 += 1
        else:
            if np.abs(U_roots[i] >= 1):
                N_1 += 1
            else:
                N_2 += 1
    return N_1, N_2, N_3 // 2


def get_Q_special(Q):
    ret_coef = [Q.coef[0]]
    for i in range(1, len(Q.coef)):
        ret_coef.append(Q.coef[i] / (2 ** i))
    print(ret_coef)


def get_daubechies_coef(N, a_N):
    Q_ = Q(N)
    print(f'Q={Q_}')

    get_Q_special(Q_)

    U_ = U(N, Q_)
    print(f'U(N, Q)={U_}')

    U_roots = U_.roots()
    print(f'U_roots={U_roots}')

    print(f'a_N={a_N}')

    N_1, N_2, N_3 = get_N1_N2_N3(U_roots)
    print(f'N_1 = {N_1}, N_2 = {N_2}, N_3 = {N_3}')

    z_k_1 = z_k(U_roots, N_1, 'real')
    print(f'z_k_1={z_k_1}')

    z_k_2 = z_k(U_roots, N_2, 'real')
    print(f'z_k_2={z_k_2}')

    z_k_3 = z_k(U_roots, N_3, 'complex')
    print(f'z_k_3={z_k_3}')

    B_1_ = B_1(N_1, z_k_1)
    print(f'B_1 = {B_1_}')

    B_2_ = B_2(N_2, z_k_2)
    print(f'B_2 = {B_2_}')

    B_3_ = B_3(N_3, z_k_3)
    print(f'B_3 = {B_3_}')
```

```python

    B_ = B(a_N, B_1_, B_2_, B_3_)
    print(f'B = {B_}')

    M_0_sqrt_2 = M_0(N, B_) * np.sqrt(2)   # TODO 3
    print(M_0_sqrt_2)
    return M_0_sqrt_2.coef


N = 6
a_N = 0

if N == 2:
    a_N = 1
if N == 3:
    a_N = 3 / 4
if N == 5:
    a_N = 35 / 64
if N == 6:
    a_N = - 63 / 128

daubechies_coef = get_daubechies_coef(N, a_N)
daubechies_coef_true = [
    -0.0010773011,
    0.0047772575,
    0.0005538422,
    -0.0315820393,
    0.0275228655,
    0.0975016056,
    -0.1297668676,
    -0.2262646940,
    0.3152503517,
    0.7511339080,
    0.4946238904,
    0.1115407434]
print(f'|daubechies_coef - daubechies_coef_true| = {LA.norm(daubechies_coef - daubechies_coef_true)}')
```

- Сравнение полученных коэффициентов с приведенными в [Добеши].

```
|daubechies_coef - daubechies_coef_true| = 8.372036942082707e-11
```

Все знаки после запятой, приведённые в массиве *daubechies_coef_true* совпали