

```
//-----
//Processing 4.0.1
//VMD code in EEGModeDecompositor_ex49
//By Teiji Sawa, MD, PhD
//Department of Anesthesiology, Kyoto Prefectural University of Medicine
//Jan 22, 2023
//Implementation of python code by K. Dragomiretskiy, & D. Zosso,
//K. Dragomiretskiy, D. Zosso, Variational Mode Decomposition,
//IEEE Trans. on Signal //Processing 62(3) 531 – 544, 2014.
//reference: http://dx.doi.org/10.1109/TSP.2013.2288675
// https://github.com/shenmusmart/VMD\_python
//---Usage-----
//Vmd vmd_1; // Define vmd_1 AS Vmd Class
//vmd_1 = new Vmd(x1, alpha, tau, K, DC, init, tol, N); // Create vmd_1 as an instance of Vmd Class
//-----
```

```
import com.github.psambit9791.jdsp.signal.Convolution;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Random;
```

```
Complex alpha = new Complex(2000.0, 0);
Complex tau = ZERO;
public static final Complex ZERO = new Complex(0, 0);
public static final Complex ONE = new Complex(1, 0);
public static final Complex TWO = new Complex(2, 0);
public static final Complex HALF = new Complex(0.5, 0);
```

```
class Vmd {

    // signal - the time domain signal (1D) to be decomposed
    // alpha - the balancing parameter of the data-fidelity constraint
    // tau - time-step of the dual ascent ( pick 0 for noise-slack )
    // K - the number of modes to be recovered
    // DC - true if the first mode is put and kept at DC (0-freq)
    // init - 0 = all omegas start at 0
    //          1 = all omegas start uniformly distributed
    //          2 = all omegas initialized randomly
    // tol - tolerance of convergence criterion; typically around 1e-6
```

```
double signal2[];
Complex alpha;
Complex tau;
int K;
int DC;
int init;
double tol;
```

```
int N;
```

```
// Constructor
```

```
Vmd(double [] _signal, Complex _alpha, Complex _tau, int _K, int _DC, int _init, double _tol, int _N) {  
    signal2 = _signal;  
    alpha = _alpha;  
    tau = _tau;  
    K = _K;  
    DC = _DC;  
    init = _init;  
    tol = _tol;  
    N = _N;  
}
```

```
double[][] vmd(double [] _signal) {  
    // -----  
    //  signal  - the time domain signal (1D) to be decomposed  
    //  alpha   - the balancing parameter of the data-fidelity constraint  
    //  tau     - time-step of the dual ascent ( pick 0 for noise-slack )  
    //  K       - the number of modes to be recovered  
    //  DC      - true if the first mode is put and kept at DC (0-freq)  
    //  init    - 0 = all omegas start at 0  
    //              1 = all omegas start uniformly distributed  
    //              2 = all omegas initialized randomly  
    //  tol     - tolerance of convergence criterion; typically around 1e-6  
    //  
    //  Output:  
    //  -----  
    //  u       - the collection of decomposed modes  
    //  u_hat   - spectra of the modes  
    //  omega   - estimated mode center-frequencies  
    //  
    // Period and sampling frequency of input signal  
    //-----define-----
```

```
int save_T;  
double fs;  
float T_f;  
int T2_f;  
//Complex alpha= new Complex(2000.0, 0);  
int N2;
```

```
double uDiff= tol + 2.2204e-16; //update step  
int n=1; //loop counter
```

```
Complex[] freqs_sub;  
Complex[] freqs_square;
```

```
Complex[] freqs_square_sub;  
Complex sum_uk[];  
Complex Alpha[];
```

```
Complex[] f_hat;  
Complex[] f_hat1;  
Complex[] f_hat_plus;  
Complex[] f_hat_plus_sub1;  
Complex[] f_hat_plus_sub2;  
Complex[] f_hat_plus_m1;  
Complex[] f_hat_plus_m2;
```

```
Complex[][] omega;  
Complex[][] omega_plus;  
Complex omega_plus_n2;  
Complex omega_plus_comp;
```

```
Complex[][] lamda_hat;  
Complex[] lamda_hat_n1;  
Complex[] lamda_hat_n2;
```

```
Complex[][] u_hat;  
Complex[][] u_hat2;  
Complex[][] u;  
Complex[][] u_hat_n1;  
Complex[] u_hat_n2;  
Complex[] u_hat_n2_ishift;
```

```
Complex[][][] u_hat_plus;  
Complex[] u_hat_plus_n1;  
Complex[] u_hat_plus_n2;  
Complex[] u_hat_plus_n3;  
Complex[] u_hat_plus_n8;  
Complex[] u_hat_plus_n9;  
Complex[] u_hat_plus_n10;  
Complex[] u_hat_plus_n11;  
Complex[][] u_hat_plus_n17;  
Complex[] u_hat_plus_n18;  
Complex[][] u_hat_plus_n20;  
Complex[] u_hat_plus_conj1;  
Complex[] u_hat_plus_conj2;
```

```
Complex[] u_hat_plus_conj3;  
Complex u_hat_plus_conj4;
```

```
Complex[] fft1_result;  
Complex[][] fft2_result;
```

```

Complex[] fft3_result1;
Complex[] fft3_result2;
Complex[] fft3_result3;
Complex[] fft3_result3_t;

double f_mirror[];
double f[];
double freqs[];

double freqs_n1[];
double [][]omega_plus_double;
double[] u_hat_plus_n4;
double[] u_hat_plus_n5;
double u_hat_plus_n6;
double u_hat_plus_n7;
double[] u_hat_plus_n12;
double[] u_hat_plus_n13;
double[] u_hat_plus_n14;
double u_hat_plus_n15;
double u_hat_plus_n16;

double[][] _u_double;
double[][] _u_double2;
double[] u_n1_double;

//-----memory keep-----
f_mirror = new double [2*T1];
//eeg = new double [2*T1];
f = new double [2*T1];

freqs = new double [2*T1];
Alpha = new Complex [K];
sum_uk = new Complex [2*T1];

u_hat = new Complex [2*T1][K];
u_hat2 = new Complex [2*T1][K];
u_hat_plus = new Complex [N][2*T1][K];
u_hat_plus_n1 = new Complex [2*T1];
u_hat_plus_n2 = new Complex [2*T1];
u_hat_plus_n3 = new Complex [2*T1];
u_hat_plus_n4 = new double [2*T1];
u_hat_plus_n5 = new double [2*T1];
u_hat_plus_n8 = new Complex [2*T1];
u_hat_plus_n9 = new Complex [2*T1];
u_hat_plus_n10 = new Complex [2*T1];
u_hat_plus_n11 = new Complex [2*T1];
u_hat_plus_n12 = new double [2*T1];

```

```

u_hat_plus_n13 = new double [2*T1];
u_hat_plus_n14 = new double [2*T1];
u_hat_plus_n17 = new Complex [2*T1][K];
u_hat_plus_n18 = new Complex [2*T1];
u_hat_plus_n20 = new Complex [T1][K];

```

```

f_hat = new Complex [2*T1];
f_hat1 = new Complex [freqs.length];
f_hat_plus = new Complex [2*T1];
f_hat_plus_sub1 = new Complex [2*T1];
f_hat_plus_sub2 = new Complex [2*T1];
f_hat_plus_m1 = new Complex [2*T1];
f_hat_plus_m2 = new Complex [2*T1];

```

```

u = new Complex [K][2*T1];

```

```

_u_double = new double [K][2*T1];
_u_double2 = new double [K][T1];
u_n1_double = new double [T1];
u_hat_n1 = new Complex [2*T1][K];
u_hat_n2 = new Complex [2*T1];
u_hat_n2_ishift = new Complex [2*T1];
u_hat_plus_conj1 = new Complex [2*T1];
u_hat_plus_conj2 = new Complex [2*T1];
u_hat_plus_conj3 = new Complex [2*T1];

```

```

omega = new Complex [N][K];
omega_plus = new Complex [N][K];
omega_plus_double = new double [N][K];

```

```

freqs_square_sub = new Complex [2*T1];
freqs_square = new Complex [2*T1];
freqs_sub = new Complex [2*T1];
freqs_n1 = new double [2*T1];

```

```

lamda_hat = new Complex [N][freqs.length];
lamda_hat_n1 = new Complex [freqs.length];
lamda_hat_n2 = new Complex [freqs.length];

```

```

fft1_result = new Complex [2*T1];
fft2_result = new Complex [K][2*T1];
fft3_result1 = new Complex [2*T1];
fft3_result2 = new Complex [2*T1];
fft3_result3 = new Complex [2*T1];
fft3_result3_t = new Complex [2*T1];

```

```

//-----body-----

```

```

signal2 = _signal;
save_T = signal2.length; //signal size 1024 to save_T
fs = 1/(save_T); //Sampling interval fs is 1/1024

// extend the signal by mirroring
// Specifically, the first half of the signal is inverted
// and added to the beginning of the original signal,
// and the second half of the signal is inverted and added to the tail of the original signal.
// Then, signal size is 2048 data points.

for (int i=0; i<int(T1/2); i++) {
    f_mirror[i] = signal2[int(T1/2)-1-i];
}
for (int i=int(T1/2); i<int(3*T1/2); i++) {
    f_mirror[i] = signal2[i-int(T1/2)];
}
for (int i=int(3*T1/2); i<int(4*T1/2); i++) {
    f_mirror[i] = signal2[int(T1-1)-(i-int(3*T1/2))];
}

//double eeg data: f = f_mirror;
f = f_mirror; //Store the mirrored EEG in a new variable f

// Time Domain 0 to T (of mirrored signal)
T_f = float(f.length); //Store 2048 mirror extension signal data in T_f

for (int i=0; i<T_f; i++) {
    t[i] = 1/T_f*(i+1);
}

// Spectral Domain discretization freq[]
for (int i=0; i<T_f; i++) {
    freqs[i] = t[i] - 0.5 - 1/T_f;
}

// For future generalizations: individual alpha for each mode
// penalty factor, balance parameter
for (int i=0; i<K; i++) {
    Alpha[i] = alpha.multiply(ONE);
}

//int N = 500

// Construct and center f_hat: Fourier transform of the signal
FastFourierTransformer fft1 = new FastFourierTransformer(DftNormalization.STANDARD);
fft1_result= fft1.transform(f, TransformType.FORWARD);

```

```

f_hat = fftshift(fft1_result); //Fourier transform result to f_hat with zero frequency component shifted to
    center of spectrum

f_hat_plus = f_hat; //Copy f_hat to f_hat_plus

for (int i=0; i<int(T_f)/2; i++) {
    f_hat_plus[i] = ZERO; //Initialize 1-1024 data of f_hat_plus with 0
}

// matrix keeping track of every iterant // could be discarded for mem
for (int i=0; i < N; i++) {
    for (int j=0; j < 2*T1; j++) {
        for (int k=0; k < K; k++) {
            u_hat_plus[i][j][k] = ZERO; //Create a complex zero array with u_hat_plus.shape:= (500, 2048, 3)
        }
    }
}

// Initialization of omega_k
for (int j = 0; j < N; j++) {
    for (int k = 0; k < K; k++) {
        omega_plus[j][k] = ZERO; //Initialization of frequency variable omega plus
    }
}

if (init == 1) {
    for (int i=1; i < K+1; i++) {
        Complex K_comp = new Complex(K, 0);
        Complex i_comp = new Complex(i, 0);
        omega_plus[0][i-1] = (HALF.divide(K_comp)).multiply(i_comp.subtract(ONE));
    }
} else if (init == 2) {
    for (int k = 0; k < K; k++) {
        Complex random_comp = new Complex(random(1), 0);
        Complex fs_comp = new Complex(fs, 0);
        omega_plus[0][k] =
            ((fs_comp.log()).exp()).add((HALF.log()).subtract(fs_comp.log()).multiply(random_comp));
        //missing sort()
    }
} else {
    for (int k = 0; k < K; k++) {
        omega_plus[0][k] = ZERO;
    }
}

if (DC == 1) {
    omega_plus[0][0] = ZERO;
}

```

```

}

// start with empty dual variables   Lagrangian multiplier  $\lambda$ 
for (int i = 0; i < N; i++) {
    for (int j=0; j < freqs.length; j++) {
        lamda_hat[i][j] = ZERO;
    }
}

for (int i=0; i<freqs.length; i++) {
    sum_uk[i] = ZERO; //accumulator
}
T2_f = int(T_f);

// ----- Main loop for iterative updates
// Update n to minimize u_hat and omega_hat
// Algorithm 2 Complete optimization of VMD

while (uDiff > tol && n < N) {
    //Repeat if tolerance of tol convergence criterion is greater than Diff or less than N=2000.
    //if (uDiff > tol) {
    // update first mode accumulator
    int k = 1;

    for (int j=0; j < freqs.length; j++) {
        u_hat_plus_n1[j] = u_hat_plus[n-1][j][K-1];
        u_hat_plus_n2[j] = u_hat_plus[n-1][j][0];
        sum_uk[j] = (u_hat_plus_n1[j].add(sum_uk[j])).subtract(u_hat_plus_n2[j]);
        //update spectrum of first mode through Wiener filter of residuals
        //Wiener filter of residuals equation (27)   u_hat_plus[][][]
        lamda_hat_n1[j] = lamda_hat[n-1][j];
        Complex freqs_comp = new Complex(freqs[j], 0.0);

        //Calculating freqs_square
        freqs_square_sub[j] = freqs_comp.subtract(omega_plus[n-1][k-1]);
        freqs_square[j] = freqs_square_sub[j].multiply(freqs_square_sub[j]);

        //Calculation of u_hat_plus[][][]
        f_hat_plus_sub1[j] = (f_hat_plus[j].subtract(sum_uk[j])).subtract(lamda_hat_n1[j].divide(TWO));
        f_hat_plus_sub2[j] = ONE.add(Alpha[k-1].multiply(freqs_square[j]));
        u_hat_plus[n][j][k-1] = f_hat_plus_sub1[j].divide(f_hat_plus_sub2[j]);
    }

    //update first omega if not held at 0
    //Equation (28) in the paper Substitutes the inner product for the calculation of the integral.

```



```

if (DC == 0) {
    //Computing omega_plus_double[][]
    //np.dot(freqs[T//2:T], (np.square(np.abs(u_hat_plus[n, T//2:T, k-1]))).T)
    //Squared absolute value of u_hat_plus (n=0, last 1024 data, k=0th) and inner product of frequency
    freqs[T//2:T]
    //np.sum(np.square(np.abs(u_hat_plus[n, T//2:T, k-1])))
    //Sum of squares of absolute values of u_hat_plus(n=0, last 1024 data, k=0th)

    for (int j=0; j < freqs.length/2; j++) {
        freqs_n1[j] = freqs[int(T2_f/2)+j];
        u_hat_plus_n3[j] = u_hat_plus[n][int(T2_f/2)+j][k-1];
        u_hat_plus_n4[j] = ((u_hat_plus_n3[j]).abs())*((u_hat_plus_n3[j]).abs());
    }

    u_hat_plus_n5 = array_transpose_double(u_hat_plus_n4);
    u_hat_plus_n6 = array_sum_double(u_hat_plus_n4);
    u_hat_plus_n7 = array_dot_double(freqs_n1, u_hat_plus_n5);
    omega_plus_double[n][k-1] = u_hat_plus_n7/u_hat_plus_n6;
}

for (k=2; k<K+1; k++) {
    //accumulator sum_uk[] calculation
    for (int j=0; j < freqs.length; j++) {
        u_hat_plus_n8[j] = u_hat_plus[n][j][k-2];
        u_hat_plus_n9[j] = u_hat_plus[n-1][j][k-1];

        sum_uk[j] = (u_hat_plus_n8[j].add(sum_uk[j])).subtract(u_hat_plus_n9[j]);

        //mode spectrum //Update u^_k for all  $\omega \geq 0$ :
        lamda_hat_n2[j] = lamda_hat[n-1][j];
    }

    omega_plus_n2 = omega_plus[n-1][k-1];

    for (int i=0; i < freqs.length; i++) {
        //double freq = freqs[i];
        Complex freqs_comp = new Complex(freqs[i], 0.0);
        freqs_sub[i] = freqs_comp.subtract(omega_plus_n2);
        freqs_square[i] = freqs_sub[i].multiply(freqs_sub[i]);
    }

    for (int i=0; i < freqs.length; i++) {
        f_hat1[i] = f_hat_plus[i].subtract(sum_uk[i]);
        f_hat_plus_m1[i] = (f_hat1[i]).subtract(lamda_hat_n2[i].divide(TWO));

        f_hat_plus_m2[i] = (ONE.add(Alpha[k-1].multiply(freqs_square[i])));
    }
}

```

```

//u_hat_plus k=0, 1, 2 renew
u_hat_plus[n][i][k-1]= f_hat_plus_m1[i].divide(f_hat_plus_m2[i]);
}

//center frequencies Update  $\omega^k$ :
for (int j=0; j < freqs.length/2; j++) {
    freqs_n1[j] = freqs[int(T2_f/2)+j];
    u_hat_plus_n10[j] = u_hat_plus[n][int(T2_f/2)+j][k-1];
    u_hat_plus_n11[j] = u_hat_plus[n][int(T2_f/2)+j][k-1];
    u_hat_plus_n12[j] = ((u_hat_plus_n10[j]).abs())*(((u_hat_plus_n10[j]).abs()));
}

u_hat_plus_n13 = array_transpose_double(u_hat_plus_n12);

for (int j=0; j < freqs.length/2; j++) {
    u_hat_plus_n14[j] = ((u_hat_plus_n11[j]).abs())*(((u_hat_plus_n11[j]).abs()));
}

u_hat_plus_n15 = array_dot_double(freqs_n1, u_hat_plus_n13);
u_hat_plus_n16 = array_sum_double(u_hat_plus_n14);
omega_plus_double[n][k-1]=u_hat_plus_n15/u_hat_plus_n16;
omega_plus[n][k-1] = new Complex(omega_plus_double[n][k-1], 0);

//Dual ascent
//Thesis formula (29)

for (int i=0; i < freqs.length; i++) {
    lamda_hat_n1[i] = lamda_hat[n-1][i];
}

for (int i=0; i < freqs.length; i++) {
    for (int j=0; j < K; j++) {
        u_hat_plus_n17[i][j] = u_hat_plus[n][i][j];
    }
    u_hat_plus_n18[i] = ZERO;

    for (int j=0; j < K; j++) {
        u_hat_plus_n18[i] = (u_hat_plus_n18[i]).add(u_hat_plus_n17[i][j]);
    }
    lamda_hat[n][i]=lamda_hat_n1[i].add(tau.multiply(u_hat_plus_n18[i].subtract(f_hat_plus[i])));
}
}

//loop counter Comparison with convergence criteria
n = n + 1;
//println("n=", n);
//converged yet?

```

```

uDiff = 2.2204e-16;

for (int i = 1; i < K+1; i++) {
    for (int j = 0; j < freqs.length; j++) {
        u_hat_plus_n1[j] = u_hat_plus[n-1][j][i-1];
        u_hat_plus_n2[j] = u_hat_plus[n-2][j][i-1];
        u_hat_plus_n3[j] = u_hat_plus_n1[j].subtract(u_hat_plus_n2[j]);
        u_hat_plus_conj1[j] = u_hat_plus_n3[j].conjugate();
        u_hat_plus_conj2[j] = u_hat_plus_conj1[j].conjugate();
    }

    u_hat_plus_conj3 = array_transpose(u_hat_plus_conj2);
    u_hat_plus_conj4 = array_dot(u_hat_plus_n3, u_hat_plus_conj3);
    uDiff = uDiff + 1/float(T2_f)* (float)((u_hat_plus_conj4).abs());
}
uDiff = abs_double(uDiff);
}
println("n=", n);
// ----- Postprocessing and cleanup
// discard empty space if converged early

N2 = min(N, n);

for (int i = 0; i < N2; i++) {
    for (int j = 0; j < K; j++) {
        double omega_plus_temp = omega_plus_double[i][j];
        omega_plus_comp = new Complex(omega_plus_temp, 0.0);
        omega_plus[i][j] = omega_plus_comp;
        omega[i][j] = omega_plus[i][j];
    }
}

// Signal reconstruction: Calculate IMF from frequency components by inverse Fourier transform
for (int i=0; i<T2_f; i++) {
    for (int j=0; j<K; j++) {
        u_hat[i][j] = ZERO;
    }
}

for (int i=0; i<int(T2_f/2); i++) {
    for (int j=0; j<K; j++) {
        u_hat_plus_n20[i][j] = u_hat_plus[N2-1][i+int(T2_f/2)][j];
        u_hat[int(T2_f/2)+i][j]=(u_hat_plus_n20[i][j]);
        u_hat[int(T2_f/2)-i][j]=(u_hat_plus_n20[i][j]).conjugate();
        u_hat_n1[i][j] = u_hat[T2_f-1][j];
        u_hat[0][j]= (u_hat_n1[i][j]).conjugate();
    }
}

```

```

}

for (int j=0; j < K; j++) {
    for (int i=0; i < T2_f; i++) {
        u[j][i] = ZERO;
    }
}

for (int k=1; k < K+1; k++) {
    for (int i=0; i < T2_f; i++) {
        u_hat_n2[i] = u_hat[i][k-1];
    }

    u_hat_n2_ishift = ifftshift(u_hat_n2); //reverse shift
    FastFourierTransformer fft2 = new FastFourierTransformer(DftNormalization.STANDARD);
    fft2_result[k-1] = fft2.transform(u_hat_n2_ishift, TransformType.INVERSE); //inverse Fourier
    transform

    for (int i=0; i < T2_f; i++) {
        _u_double[k-1][i] = fft2_result[k-1][i].getReal();
        // _u_double[k-1][i] = fft2_result[k-1][i].abs();
    }
}

// remove mirror part
for (int i=0; i<K; i++) {
    for (int j=int(T2_f/4); j<3*int(T2_f/4); j++) {
        _u_double[i][j-int(T2_f/4)]=_u_double[i][j];
    }
}

//recompute spectrum: Obtain the frequency spectrum from the IMF by Fourier transform.
for (int i=0; i< int(T2_f/2); i++) {
    for (int j=0; j<K; j++) {
        u_hat[i][j] = ZERO;
    }
}

for (int k=1; k < K+1; k++) {
    for (int i=0; i < int(T2_f/2); i++) {
        //imfs_float[k-1][i] = (float) _u_double[k-1][i];
        u_n1_double[i] = _u_double[k-1][i];
    }

    FastFourierTransformer fft3 = new FastFourierTransformer(DftNormalization.STANDARD);
    fft3_result1 = fft3.transform(u_n1_double, TransformType.FORWARD);

```

```

fft3_result2 = fftshift(fft3_result1);

for (int i=0; i< int(T2_f/2); i++) {
    fft3_result3[i] = fft3_result2[i].conjugate();
}

fft3_result3_t=array_transpose(fft3_result3);

for (int i=0; i < int(T2_f/2); i++) {
    u_hat2[i][k-1]= fft3_result3_t[i];
    _u_double2[k-1][i] = u_hat2[i][k-1].abs();
}
}

return _u_double;
}
}

public static int[] argsort(final double[] a) {
    return argsort(a, true);
}

public static int[] argsort(double[] a, boolean ascending) {
    Integer[] indexes = new Integer[a.length];
    for (int i = 0; i < indexes.length; i++) {
        indexes[i] = i;
    }
    Arrays.sort(indexes, new Comparator<Integer>() {
        @Override
        public int compare(final Integer i1, final Integer i2) {
            return (ascending ? 1 : -1) * Double.compare(a[i1], a[i2]);
        }
    });
    return asArray(indexes);
}

public static <T extends Number> int[] asArray(final T... a) {
    int[] b = new int[a.length];
    for (int i = 0; i < b.length; i++) {
        b[i] = a[i].intValue();
    }
    return b;
}

float[] array_square(float _array[]) {
    float _array2[];

```

```

_array2 = new float [_array.length];

for (int i=0; i<_array.length; i++) {
    _array2[i] = _array[i] * _array[i];
}
return _array2;
}

```

```

Complex[] fftshift(Complex _complex[]) {
    Complex complex2[];
    complex2 = new Complex [_complex.length];
    for (int i=0; i<_complex.length/2; i++) {
        complex2[_complex.length/2+i] = _complex[i];
    }
    for (int i=_complex.length/2; i<_complex.length; i++) {
        complex2[i-_complex.length/2] = _complex[i];
    }
    return complex2;
}

```

```

Complex[] ifftshift(Complex _complex[]) {
    Complex complex2[];
    complex2 = new Complex[_complex.length];
    for (int i=0; i<_complex.length/2; i++) {
        complex2[_complex.length/2+i] = _complex[i];
    }
    for (int i=_complex.length/2; i<_complex.length; i++) {
        complex2[i-_complex.length/2] = _complex[i];
    }
    return complex2;
}

```

```

double[] fftshift_d(double _double[]) {
    double double2[];
    double2 = new double[_double.length];
    for (int i=0; i<_double.length/2; i++) {
        double2[_double.length/2+i] = _double[i];
    }
    for (int i=_double.length/2; i<_double.length; i++) {
        double2[i-_double.length/2] = _double[i];
    }
    return double2;
}

```

```

double[] ifftshift_d(double _double[]) {
    double double2[];
    double2 = new double[_double.length];

```

```

    for (int i=0; i<_double.length/2; i++) {
        double2[_double.length/2+i] = _double[i];
    }
    for (int i=_double.length/2; i<_double.length; i++) {
        double2[i-_double.length/2] = _double[i];
    }
    return double2;
}

double[] append_double(double [] array1, double [] array2) {
    double [] array;
    array = new double [array1.length+array2.length];
    for (int i = 0; i < array1.length; i++) {
        array[i] = array1[i];
    }
    for (int i = 0; i < array2.length; i++) {
        array[i+array1.length] = array2[i];
    }
    return(array);
}

Complex[] array_transpose(Complex _array[]) {
    Complex array2[];
    array2 = new Complex [_array.length];
    for (int j=0; j<_array.length; j++) {
        array2[j]= _array[j];
    }
    return array2;
}

double[] array_transpose_double(double _array[]) {
    double array2[];
    array2 = new double[_array.length];
    for (int j=0; j<_array.length; j++) {
        array2[j]= _array[j];
    }
    return array2;
}

Complex array_dot(Complex _array1[], Complex _array2[]) {
    Complex dot_result = new Complex(0, 0);
    for (int i=0; i<_array1.length; i++) {
        dot_result = dot_result.add(_array1[i].multiply(_array2[i]));
    }
    return dot_result;
}

```

```
double array_dot_double(double _array1[], double _array2[]) {  
    double dot_result = 0;  
    for (int i=0; i<_array1.length; i++) {  
        dot_result += (_array1[i]*_array2[i]);  
    }  
    return dot_result;  
}
```

```
Complex array_sum(Complex _array1[]) {  
    Complex sum_result = new Complex(0, 0);  
    for (int i=0; i<_array1.length; i++) {  
        sum_result = sum_result.add(_array1[i]);  
    }  
    return sum_result;  
}
```

```
double array_sum_double(double _array1[]) {  
    double sum_result = 0;  
    for (int i=0; i<_array1.length; i++) {  
        sum_result += _array1[i];  
    }  
    return sum_result;  
}
```

```
double abs_double(double _uDiff) {  
    double _temp;  
    if (_uDiff >= 0) {  
        _temp = _uDiff;  
    } else {  
        _temp = - _uDiff;  
    }  
    return _temp;  
}
```