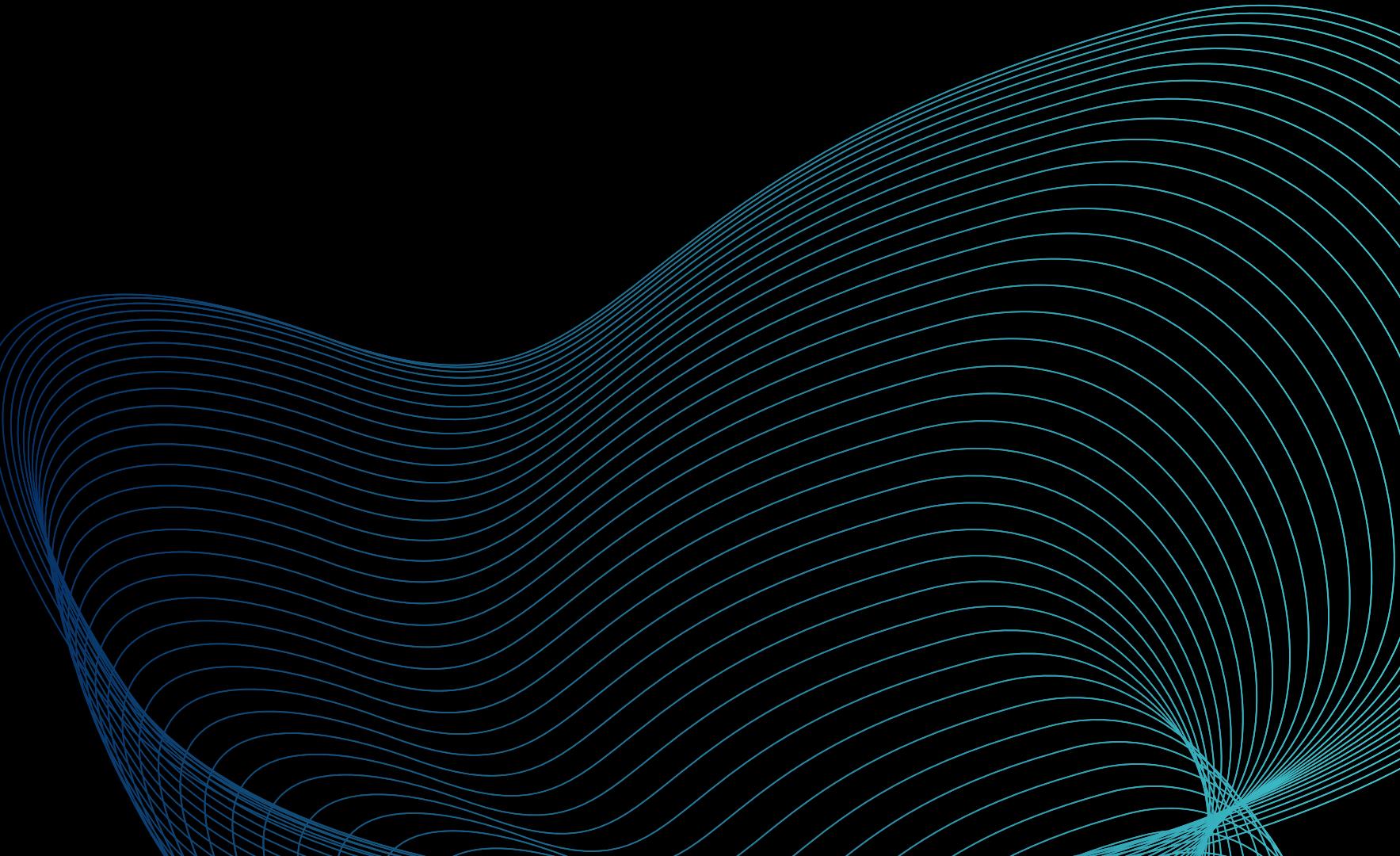


Transformada Rápida de Fourier

INTEGRANTES:

Apaza Coaquira Aaron Misash
Apaza Coaquira Eileen Karin
Caceres Gutierrez Brigham Jeffrey



Introducción



Nos permite transformar una señal que está entre el dominio del tiempo y el dominio de la frecuencia. Es reversible ya que las señales se pueden transformar entre dominios.

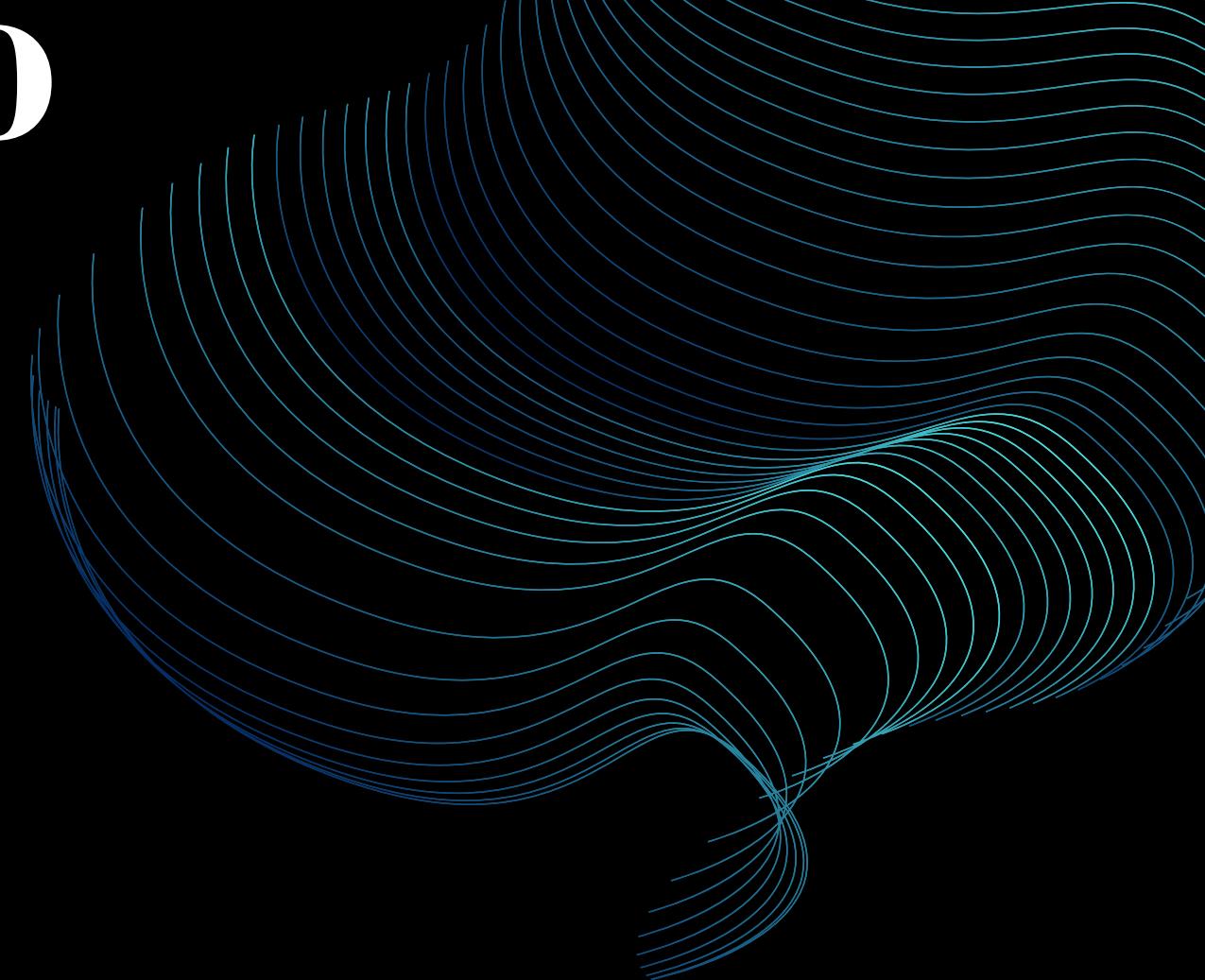
Es uno de los algoritmos más importantes, se usa para la comprensión de imagen, comprensión de audio, procesamiento de señal, etc.

Marco Teórico

Serie Fourier:

Ecuación original:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{n\pi t}{L}\right) + b_n \sin\left(\frac{n\pi t}{L}\right) \right]$$



Notación por integración

$$F(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx$$

$$f(x) = \int_{-\infty}^{\infty} F(\xi) e^{-2\pi i x \xi} \xi$$

Dónde:

$F(\xi)$ → una función en el espacio de frecuencia

ξ → Es un valor en el espacio de frecuencia

$f(x)$ → una función en el espacio real, y

x → un valor en el espacio real.

✓ DTF

- Para determinar el DTF de una señal discreta (donde es el tamaño de su dominio), multiplicamos cada uno de sus valores por elevado a alguna función

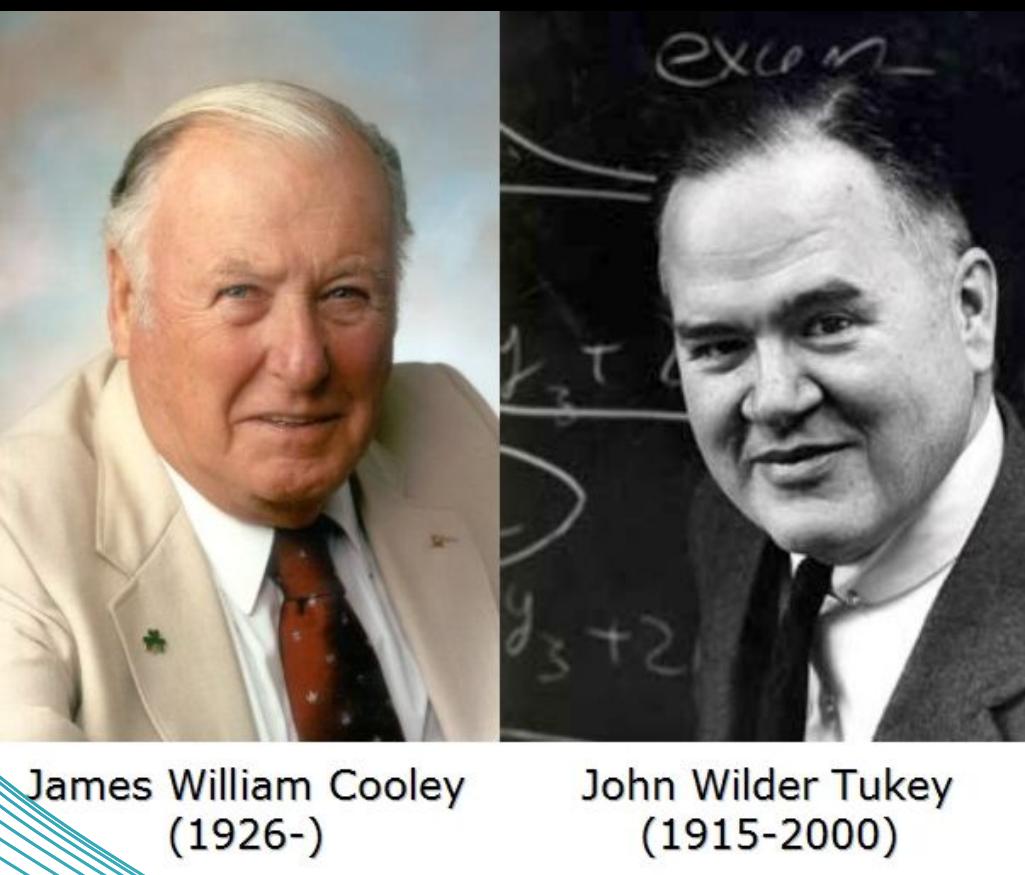
- A continuación, sumamos los resultados obtenidos
- Si usáramos una computadora para calcular la Transformada Discreta de Fourier de una señal, necesitaría realizar N (multiplicaciones) $\times N$ (sumas) = $O(N^2)$ operaciones.

$$x [K] = \sum_{n=0}^{N-1} x [n] e^{\frac{-j2\pi kn}{N}}$$

Donde $k = 0, \dots, N - 1$

Algoritmo de Cooley-Tukey

El algoritmo de Cooley-Tukey calcula el DFT directamente con menos sumas y sin multiplicaciones matriciales.



James Cooley co-inventó con John Tukey la transformada rápida de Fourier (FFT) para convertir las señales del dominio del tiempo al dominio de la frecuencia.

Algoritmo de Cooley-Tukey

Supongamos que separamos la Transformada de Fourier en subsecuencias indexadas pares e impares.

$$\begin{cases} n = 2r & \text{if } even \\ n = 2r + 1 & \text{if } odd \end{cases}$$

where $r = 1, 2, \dots, \frac{N}{2} - 1$

Algoritmo de Cooley-Tukey

La ventaja de este enfoque radica en el hecho de que las subsecuencias indexadas pares e impares se pueden calcular simultáneamente.

$$x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r]e^{\frac{-j2\pi k(r)}{N/2}} + x[k] = e^{\frac{-j2\pi k}{N}} \sum_{r=0}^{\frac{N}{2}-1} x[2r+1]e^{\frac{-j2\pi k(r)}{N/2}}$$

$$x[k] = x_{even}[k] + e^{\frac{-j2\pi k}{N}} x_{odd}[k]$$

El truco para el algoritmo de Cooley-Tukey es la recursión

Diagrama Mariposa

Es una porción de la computación que combina los resultados de pequeñas transformadas discretas de Fourier (DFT) en una DFT mas grande, o viceversa.

Imagina que necesitamos realizar un FFT de una matriz de solo 2 elementos.

Podemos representar esta adición con la siguiente mariposa (radix-2):

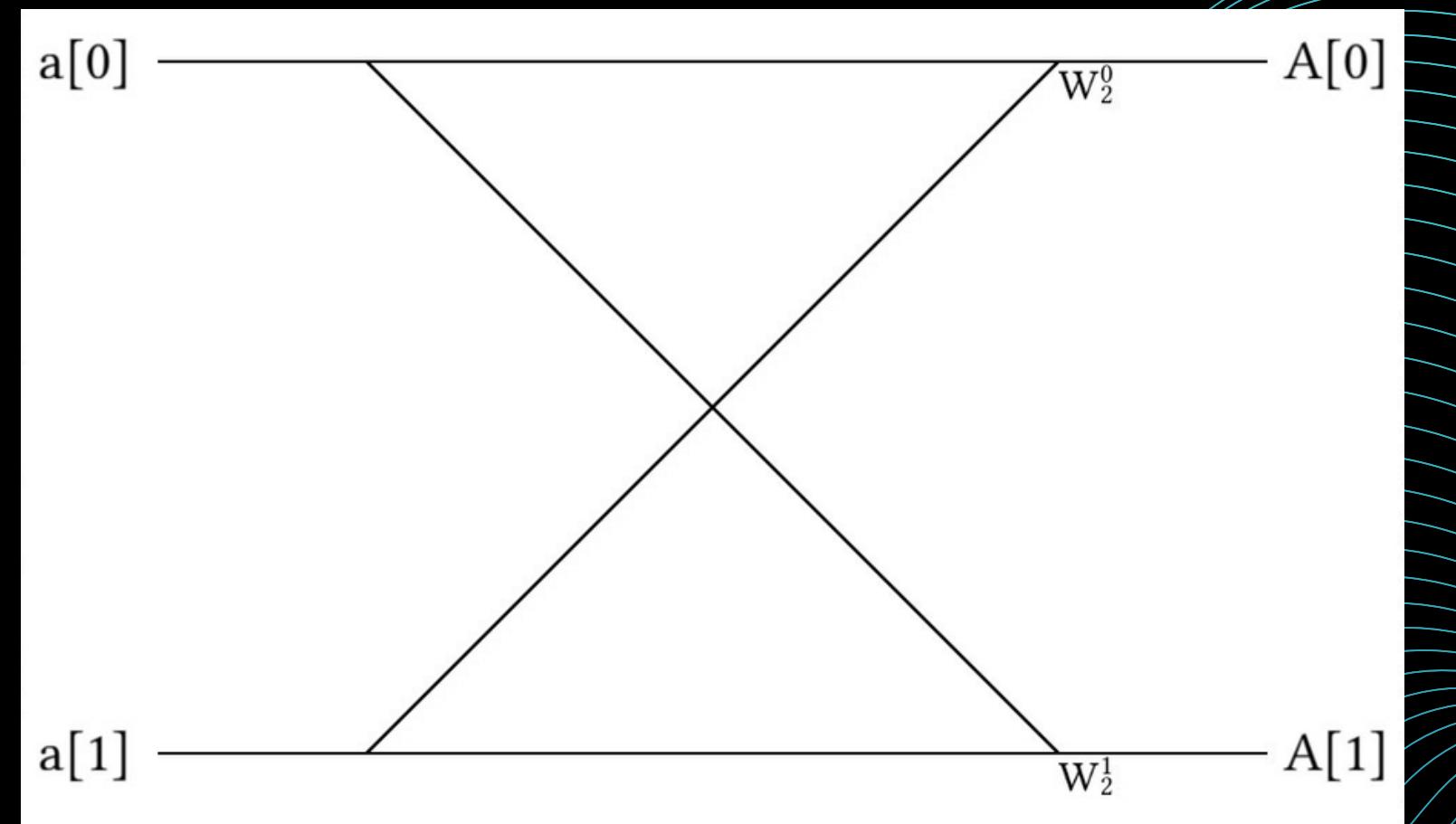


Diagrama Mariposa

Resulta que la segunda mitad de nuestra matriz de ω los valores son siempre los negativos de la primera mitad, por lo que $\omega_{20} = -\omega_{21}$.

Por lo que podemos utilizar el siguiente diagrama de mariposas:

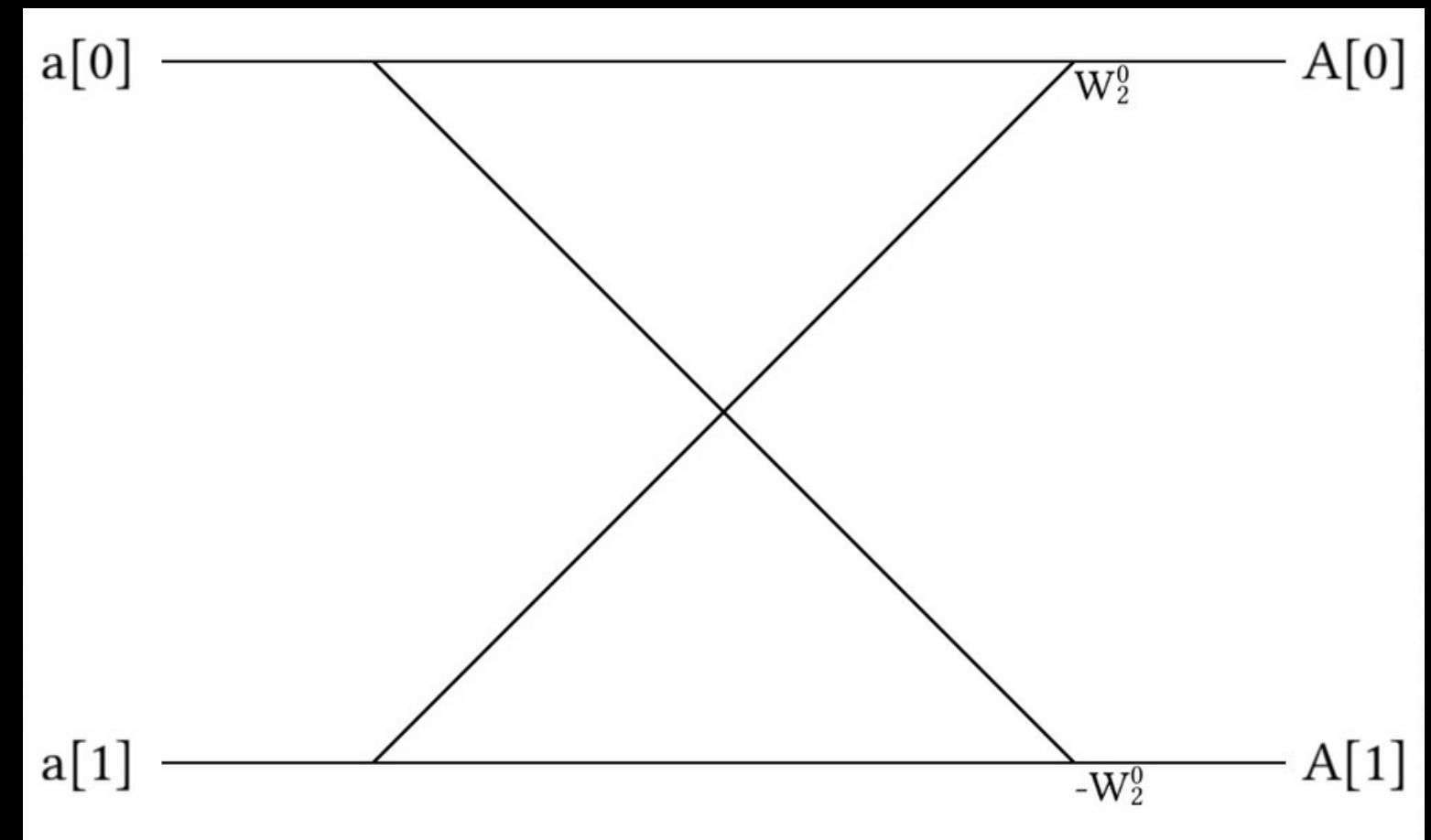
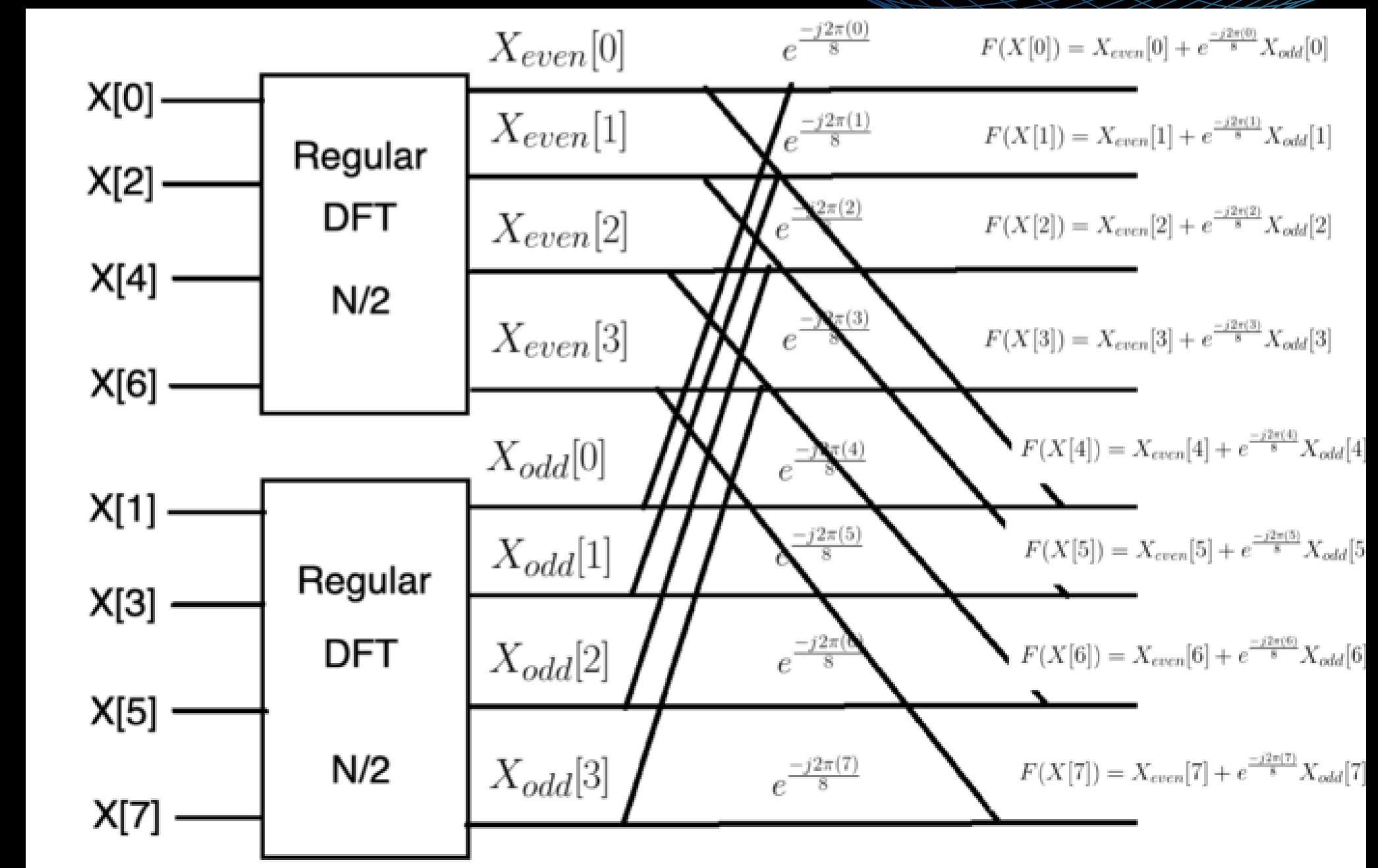


Diagrama Mariposa

Imagina que necesitamos realizar un FFT de una matriz de solo 2 elementos. Podemos representar esta adición con la siguiente mariposa (radix-2):



Código -DFT

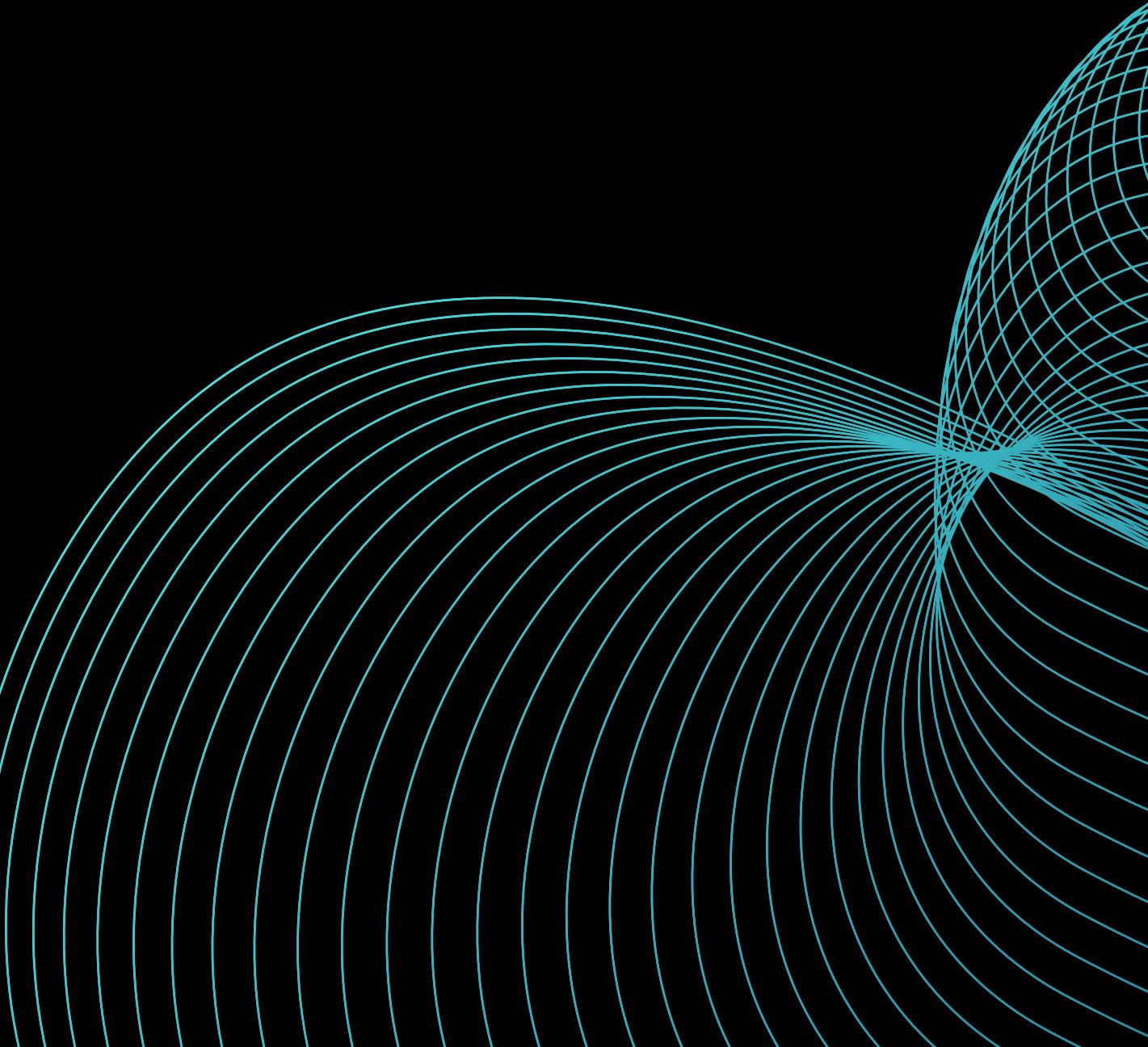
```
#include <complex>
#include <windows.h>
#include <vector>

using namespace std;

const double PI = 3.14159265;

vector<complex<double>> m_DFT(vector<complex<double>>
    m_input) {
    vector<complex<double>> m_output;
    int m_N = m_input.size();

    complex<double> m_suma(0, 0);
    for (int m_k = 0; m_k < m_N; m_k++)
    {
        m_suma.real(0);
        m_suma.imag(0);
        for (int m_n = 0; m_n < m_N; m_n++)
        {
            double m_theta = -2 * PI * m_k * m_n / m_N;
            complex<double> W_N(exp(complex<double>(0, m_theta)));
            m_suma += m_input[m_n] * W_N;
        }
        m_output.push_back(m_suma);
    }
    return m_output;
}
```



Código -Cooley-Tukey

```
#include <complex>
#include <windows.h>
#include <vector>

using namespace std;

const double PI = 3.14159265;

vector<complex<double>> rec_FFT(vector<complex<double>>
    m_input)
{
    int m_N = m_input.size();
    vector<complex<double>> m_output(m_N);

    if (m_N == 1)
        return m_input;

    complex<double> w_n(exp(complex<double>(0, (-2) * PI /
        m_N)));
    complex<double> w(1, 0);

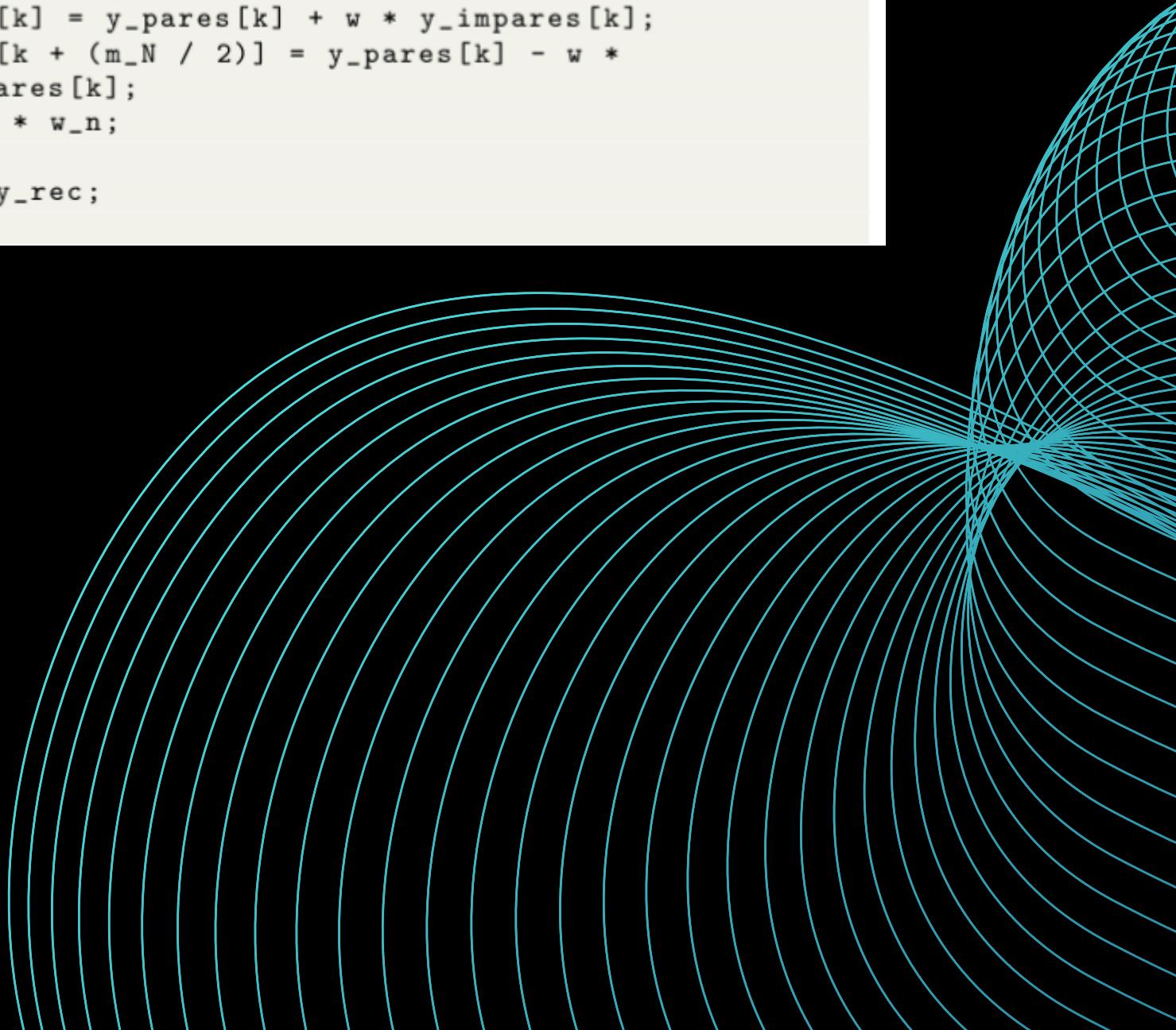
    ///DIVIDE
    vector<complex<double>> pares(m_N / 2);
    vector<complex<double>> impares(m_N / 2);

    int indx = 0;
    for (int idxpar = 0; idxpar < m_N; idxpar += 2)
    {
        pares[indx] = m_input[idxpar];
        indx++;
    }

    indx = 0;
    for (int idximpar = 1; idximpar < m_N; idximpar
        +=2)
    {
        impares[indx] = m_input[idximpar];
        indx++;
    }
}
```

```
//CONQUISTA
vector<complex<double>> y_pares = rec_FFT(pares);
vector<complex<double>> y_imparas = rec_FFT(imparas);

//VENCERAS o COMBINARAS o MEZCLARAS
vector<complex<double>> y_rec(m_N);
for (int k = 0; k < m_N / 2; k++)
{
    y_rec[k] = y_pares[k] + w * y_imparas[k];
    y_rec[k + (m_N / 2)] = y_pares[k] - w *
        y_imparas[k];
    w = w * w_n;
}
return y_rec;
```



Gracias