DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Resolviendo la Ecuación de Onda mediante la Transformada de Fourier Acelerada

8 de Septiembre de 2025

Organización del Computador II

Integrante	LU	Correo electrónico
Polonuer, Joaquin	1612/21	jtpolonuer@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

1.1. La Ecuación de Onda

La ecuación de onda representa uno de los fenómenos físicos más fundamentales en la naturaleza, describiendo la propagación de perturbaciones en medios continuos. Desde ondas sonoras y electromagnéticas hasta vibraciones mecánicas, este modelo matemático encuentra aplicación en campos tan diversos como la acústica, la óptica, la sismología y la ingeniería estructural [1].

La ecuación de onda unidimensional se expresa como:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{1}$$

donde $u(x,t)$ representa el desplazamiento de la onda en el punto x y tiempo t, y c es la velocidad de propagación característica del medio. Esta ecuación en derivadas parciales de segundo orden describe fenómenos como:

- Vibraciones de cuerdas tensadas
- Propagación de ondas sonoras en tubos
- Ondas electromagnéticas en líneas de transmisión

Naturalmente, su extensión a dos dimensiones espaciales es:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = c^2 \nabla^2 u \tag{2}$$

Esta formulación bidimensional modela fenómenos como:

- Vibraciones de membranas (tambores, diafragmas)
- Ondas superficiales en líquidos
- Propagación de ondas sísmicas en planos
- Ondas electromagnéticas

1.2. La Transformada de Fourier

La Transformada de Fourier constituye una herramienta matemática fundamental para el análisis de fenómenos ondulatorios, permitiendo descomponer señales complejas en sus componentes frecuenciales básicas. Esta transformación resulta especialmente poderosa en el contexto de la resolución de ecuaciones diferenciales parciales [2].

La Transformada de Fourier continua de una función $f(x)$ se define como:

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x) e^{-i\omega x} dx \tag{3}$$

y su transformada inversa:

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega x} d\omega \tag{4}$$

1.3. Resolución de la Ecuación de Onda mediante la Transformada de Fourier

La aplicación de la Transformada de Fourier a la ecuación de onda permite convertir la ecuación en derivadas parciales en una ecuación diferencial ordinaria, lo que simplifica notablemente su resolución. Para abordar la ecuación (1), se aprovechan propiedades fundamentales de la transformada de Fourier relacionadas con las derivadas:

Propiedad 1: La transformada de Fourier de la derivada n-ésima de una función es igual a la multiplicación por $(i\omega)^n$ de la transformada de la función original. En particular, para $n = 1$ se obtiene el caso de la derivada simple.

$$\mathcal{F}\left\{ \frac{\partial^n f}{\partial x^n} \right\} = (i\omega)^n \hat{f}(\omega) \tag{5}$$

Propiedad 2: Cuando se transforma respecto a una variable diferente a la que se deriva, la transformada de la derivada es igual a la derivada de la transformada:

$$\mathcal{F}_x \left\{ \frac{\partial f}{\partial t} \right\} = \frac{\partial \hat{f}}{\partial t} \tag{6}$$

donde $\mathcal{F}_x$ denota la transformada de Fourier respecto a la variable x.

Es por estas dos propiedades que al aplicar la Transformada de Fourier espacial, obtenemos:

$$\frac{\partial^2 \hat{u}}{\partial t^2} = -c^2 \omega^2 \hat{u} \tag{7}$$

donde $\hat{u}(\omega, t)$ es la transformada de Fourier de $u(x, t)$ respecto a x.

Que es ecuación diferencial ordinaria con solución analítica conocida:

$$\hat{u}(\omega, t) = A(\omega) e^{ic\omega t} + B(\omega) e^{-ic\omega t} \tag{8}$$

Los coeficientes $A(\omega)$ y $B(\omega)$ se determinan a partir de las condiciones iniciales, y la solución final se obtiene aplicando la transformada inversa.

1.4. Transformada Discreta de Fourier

Para implementaciones computacionales, la Transformada de Fourier continua debe discretizarse. La Transformada Discreta de Fourier (DFT) de una secuencia finita $x[n]$ de N elementos se define como:

$$\hat{x}[k] = \sum_{n=0}^{N-1} x[n] e^{-i2\pi kn/N}, \quad k = 0, 1, \ldots, N-1 \tag{9}$$

donde $\hat{x}[k]$ representa los coeficientes espectrales discretos. La transformada inversa se expresa como:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} \hat{x}[k] e^{i2\pi kn/N}, \quad n = 0, 1, \ldots, N-1 \tag{10}$$

La implementación directa de la DFT requiere $O(N^2)$ operaciones complejas, lo que resulta computacionalmente prohibitivo para secuencias largas. Esta limitación motivó el desarrollo del algoritmo Fast Fourier Transform.

1.5. Transformada Rápida de Fourier (Cooley-Tukey)

El algoritmo FFT, desarrollado por Cooley y Tukey en 1965 [3], reduce la complejidad computacional de $O(N^2)$ a $O(N \log N)$ mediante la estrategia de divide and conquer. Para $N = 2^m$, el algoritmo descompone la DFT en DFTs más pequeñas.

El algoritmo separa la secuencia de entrada en muestras pares e impares:

$$\hat{x}[k] = \sum_{n \text{ par}} x[n] e^{-i2\pi kn/N} + \sum_{n \text{ impar}} x[n] e^{-i2\pi kn/N} \tag{11}$$

Sustituyendo $n = 2r$ para índices pares y $n = 2r + 1$ para impares:

$$\hat{x}[k] = \sum_{r=0}^{N/2-1} x[2r] e^{-i2\pi kr/(N/2)} + e^{-i2\pi k/N} \sum_{r=0}^{N/2-1} x[2r+1] e^{-i2\pi kr/(N/2)} \tag{12}$$

Definiendo:

$$\hat{x}_{\text{par}}[k] = \sum_{r=0}^{N/2-1} x[2r] e^{-i2\pi kr/(N/2)} \tag{13}$$

$$\hat{x}_{\text{impar}}[k] = \sum_{r=0}^{N/2-1} x[2r+1] e^{-i2\pi kr/(N/2)} \tag{14}$$

La ecuación se simplifica a:

$$\hat{x}[k] = \hat{x}_{\text{par}}[k] + W_N^k \cdot \hat{x}_{\text{impar}}[k] \tag{15}$$

donde $W_N^k = e^{-i2\pi k/N}$ es el factor de giro (twiddle factor).

Aprovechando la periodicidad $\hat{x}_{\text{par}}[k + N/2] = \hat{x}_{\text{par}}[k]$ y la simetría $W_N^{k+N/2} = -W_N^k$:

$$\hat{x}[k] = \hat{x}_{\text{par}}[k] + W_N^k \cdot \hat{x}_{\text{impar}}[k] \tag{16}$$

$$\hat{x}[k + N/2] = \hat{x}_{\text{par}}[k] - W_N^k \cdot \hat{x}_{\text{impar}}[k] \tag{17}$$

Este proceso se aplica recursivamente hasta obtener DFTs de un solo elemento.

1.5.1. Algoritmo Recursivo FFT

El algoritmo recursivo implementa directamente la estrategia divide and conquer descrita anteriormente, con complejidad $O(N \log N)$:

```
def fft_recursive(x):
    n = len(x)
    if n <= 1:
        return x

    # Dividir en pares e impares
    x_par = [x[2*i] for i in range(n//2)]
    x_impar = [x[2*i+1] for i in range(n//2)]

    # Aplicar FFT recursivamente
    y_par = fft_recursive(x_par)
    y_impar = fft_recursive(x_impar)

    # Combinar resultados
    y = [0] * n
    for k in range(n//2):
        w = exp(-2j * pi * k / n)
        y[k] = y_par[k] + w * y_impar[k]
        y[k + n//2] = y_par[k] - w * y_impar[k]

    return y
```

Sin embargo, las implementaciones prácticas utilizan la version iterativa, que mantienen la misma complejidad algorítmica pero con mejor rendimiento en la práctica [4], evitando usar intensivamente el stack para hacer llamadas recursivas.

1.5.2. Algoritmo Iterativo FFT

La versión iterativa corresponde a una implementación *bottom-up*, comienza resolviendo DFTs de tamaño 2 y va duplicando la longitud en cada nivel. Para que los subproblemas se alineen correctamente en cada etapa, la señal de entrada debe reordenarse según el patrón de *bit-reversal*. De este modo, los accesos secuenciales en los bucles anidados producen exactamente las mismas combinaciones que la versión recursiva.

```
def fft_iterative(x):
    n = len(x)
    if n <= 1:
        return x

    # Bit-reversal
    j = 0
    for i in range(1, n):
        bit = n >> 1
        while j & bit:
            j ^= bit
            bit >>= 1
        j ^= bit
        if i < j:
```

```
            x[i], x[j] = x[j], x[i]

    # FFT iterativa
    length = 2
    while length <= n:
        w = exp(-2j * pi / length)
        for i in range(0, n, length):
            wn = 1 + 0j
            for j in range(length // 2):
                u = x[i + j]
                v = x[i + j + length // 2] * wn
                x[i + j] = u + v
                x[i + j + length // 2] = u - v
                wn *= w
        length <<= 1

    return x
```

Esta implementación iterativa no solo ahorra el costo de llamar multiples veces a la funcion, sino que ademas permite realizar optimizaciones como las que veremos a continuacion, haciendo uso de herramientas de paralelización, lo que resulta en mejor rendimiento en la práctica.

1.6. Transformada de Fourier Bidimensional

Para la resolución numérica de la ecuación de onda en dos dimensiones, es necesario extender la Transformada de Fourier al caso bidimensional. La Transformada Discreta de Fourier en 2D de una matriz $x[m, n]$ de dimensiones $M \times N$ se define como:

$$\hat{x}[k, l] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x[m, n] e^{-i2\pi(km/M+ln/N)} \tag{18}$$

donde $k = 0, 1, \ldots, M-1$ y $l = 0, 1, \ldots, N-1$.

La transformada inversa se expresa como:

$$x[m, n] = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} \hat{x}[k, l] e^{i2\pi(km/M+ln/N)} \tag{19}$$

Una propiedad fundamental de la FFT bidimensional es su separabilidad, que permite descomponer el cálculo en aplicaciones consecutivas de FFT unidimensionales:

$$\hat{x}[k, l] = \sum_{m=0}^{M-1} e^{-i2\pi km/M} \left[\sum_{n=0}^{N-1} x[m, n] e^{-i2\pi ln/N} \right] \tag{20}$$

Esto se puede implementar eficientemente mediante el siguiente algoritmo de dos pasos:

1. **FFT por filas**: Aplicar FFT 1D a cada fila de la matriz de entrada:

$$\hat{y}[m, l] = \sum_{n=0}^{N-1} x[m, n] e^{-i2\pi ln/N} \tag{21}$$

2. **FFT por columnas**: Aplicar FFT 1D a cada columna del resultado anterior:

$$\hat{x}[k, l] = \sum_{m=0}^{M-1} \hat{y}[m, l] e^{-i2\pi km/M} \tag{22}$$

Esto se implementa en C de la siguiente manera:

```
static void fft2d(Complex *data, int rows, int cols, int inverse)
{
```

```
    Complex *temp = (Complex *)malloc(cols * sizeof(Complex));

    for (int i = 0; i < rows; i++)
    {
        memcpy(temp, &data[i * cols], cols * sizeof(Complex));
        fft_1d(temp, cols, inverse);
        memcpy(&data[i * cols], temp, cols * sizeof(Complex));
    }

    temp = (Complex *)realloc(temp, rows * sizeof(Complex));
    for (int j = 0; j < cols; j++)
    {
        for (int i = 0; i < rows; i++)
        {
            temp[i] = data[i * cols + j];
        }
        fft_1d(temp, rows, inverse);
        for (int i = 0; i < rows; i++)
        {
            data[i * cols + j] = temp[i];
        }
    }

    free(temp);
}
```

1.6.1. Aplicación a la Ecuación de Onda

En el contexto de la resolución de la ecuación de onda bidimensional, la FFT 2D permite transformar el operador Laplaciano ∇^2 del dominio espacial al dominio frecuencial:

$$\nabla^2 u(x,y) \xrightarrow{\text{FFT 2D}} -(k_x^2 + k_y^2)\hat{u}(k_x, k_y) \tag{23}$$

donde $k_x = 2\pi n_x / L_x$ y $k_y = 2\pi n_y / L_y$ son los números de onda espaciales discretos, con n_x, n_y los índices frecuenciales y L_x, L_y las dimensiones del dominio computacional.

Esta transformación convierte la ecuación diferencial parcial en una ecuación algebraica en el dominio frecuencial, facilitando significativamente su resolución numérica mediante métodos espectrales.

2. Metodología

Se propone implementar un simulador físico que permita visualizar la evolución de una onda a traves de un campo. Para esto, desarrollé las interfaces `WaveSimulation2D` y `WaveVisualizer` (ver seccion experimental). A su vez, la interfaz `WaveSimulation2D` se implemento en varios backends distintos: Python, NumPy, C, C + ASM (Assembly), y C + AVX.

El objetivo es evaluar el rendimiento de cada implementación midiendo la cantidad de pasos de simulación puede procesar cada backend en un segundo. Esta métrica es fundamental para evaluar la eficiencia computacional de diferentes enfoques de implementación.

2.1. Implementación Propuesta

Con el objetivo de facilitar la experimentación, se propone utilizar un diseño comun a todos los backends. A modo de ejemplo, se muestra la implementacion de uno de ellos:

```
class ASMWaveSimulation2D:
    def __init__(self, size=256, domain_size=10.0, wave_speed=1.0, dt=0.01):
        self.c_core = c_backend_asm
        self._sim_ptr = self.c_core.create_simulation(size, domain_size, wave_speed, dt)

    def add_wave_source(self, x_pos, y_pos, amplitude=1.0, frequency=3.0, width=0.5):
        self.c_core.add_wave_source(self._sim_ptr, x_pos, y_pos, amplitude, frequency, width)
```

```
    def step(self):
        self.c_core.step_simulation(self._sim_ptr)

    def get_intensity(self):
        return self.c_core.get_intensity(self._sim_ptr)

    def get_real_part(self):
        return self.c_core.get_real_part(self._sim_ptr)
```

La clase principal está hecha en Python, porque facilita la visualización. Sin embargo, toda la lógica y el procesamiento se realiza en C y Assembler. A continuación se muestra un diagrama:

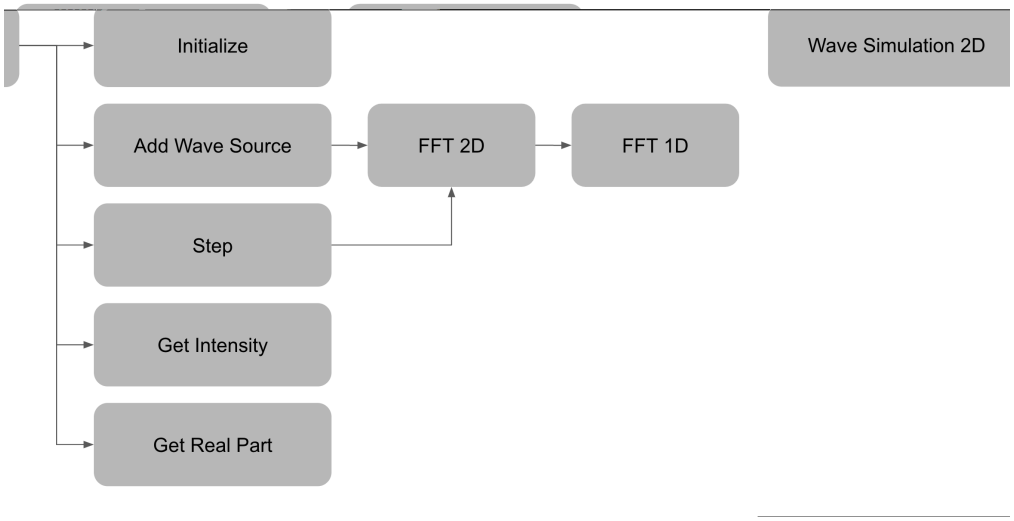

Figura 1: Diagrama de arquitectura del simulador de ondas

En el ejemplo anterior, Wave Simulation 2D provee una fachada en Python, FFT1D esta programada en Assembler y todo lo demas se programa en C.

2.1.1. Initialize

Establece el escenario matemático de la simulación: crea una grilla 2D que representa el espacio físico donde se propagarán las ondas, prepara las herramientas matemáticas (transformadas de Fourier) y define los parámetros fundamentales como la velocidad de propagación.

2.1.2. Add Wave Source

Introduce perturbaciones localizadas en el dominio, como tirar una piedra al agua. Genera patrones de ondas circulares que se propagan desde un punto específico, con una amplitud y frecuencia determinadas. Cada fuente crea ondas concéntricas que luego interactúan entre sí y con las ondas ya existentes.

2.1.3. Step

Hace avanzar el tiempo de la simulación resolviendo la ecuación de ondas. Calcula cómo cada punto de la grilla cambia su altura/amplitud basándose en la física de propagación de ondas. Utiliza métodos espectrales que transforman el problema al dominio de frecuencias, donde es más eficiente calcular la evolución temporal.

2.1.4. Get Intensity

Calcula la intensidad o densidad de energía en cada punto del dominio. Representa qué tan fuerte es la onda en cada ubicación, siempre como valores positivos. Es útil para visualizar dónde se concentra la energía de las ondas y observar patrones de interferencia constructiva.

2.1.5. Get Real Part

Extrae la parte real del campo de ondas complejo, representando el desplazamiento físico real que veríamos. Puede ser positiva (cresta) o negativa (valle), mostrando las oscilaciones reales de la superficie. Es la representación más directa de altura de la onda en cada punto del espacio.

Como se ve en el diagrama, necesitamos calcular la transformada de fourier para cada paso de la simulacion. Es por esto que la propuesta del trabajo es tratar de optimizar el algoritmo a distintos niveles y comparar sus rendimientos.

2.2. Python y NumPy

Para establecer un baseline de rendimiento, implementé dos versiones en Python:

2.2.1. Python puro

La implementación utiliza el algoritmo iterativo FFT descrito en la sección 1.5, utilizando listas de números complejos. Esta versión sirve como referencia para entender cuánto más rápido funcionan las implementaciones en C y las librerías optimizadas como NumPy.

Esta implementación presenta limitaciones inherentes de Python:

- Lentitud en operaciones numéricas
- Uso ineficiente de memoria, ya que las listas se almacenan de forma dispersa en memoria

2.2.2. NumPy

Implementación optimizada que aprovecha las operaciones vectorizadas y bibliotecas optimizadas de álgebra lineal [5]. En lugar de implementar la FFT manualmente, utilizamos directamente la función optimizada de NumPy:

```
def fft2(self, x):
    return np.fft.fft2(x)
```

Como se observará en los resultados experimentales, la implementación de NumPy es extremadamente rápida y difícil de superar, aunque lograremos un rendimiento bastante similar con nuestras implementaciones optimizadas en C.

2.3. C

Como se ve en `ASMWaveSimulation2D`, la clase implementada en Python es simplemente una fachada, y toda la lógica y estructuras de datos utilizadas para correr la simulación se manejan desde C. Esto funciona así en el backend de C puro, como en el de C + AVX y C + Assembler.

En el archivo `c_backend.c` definimos dos structs fundamentales: `Complex` y `WaveSimulation`.

```
typedef struct
{
    double real;
    double imag;
} Complex;

typedef struct
{
    Complex *wave;
    Complex *wave_k;
    double *grid_coords;
    double *k_grid_coords;
    double *K;
    int size;
    double domain_size;
    double wave_speed;
    double dt;
    double dx;
} WaveSimulation;
```

WaveSimulation funciona de la siguiente manera:

- **grid_coords**: Es una grilla cuadrada de un tamaño determinado (`size`). En cada posición guarda un valor (x, y) que determina a qué punto del dominio equivale esa posición. En la práctica, los valores se guardan todos contiguos en memoria: $[x_0, y_0, x_1, y_1, \ldots]$.
- **k_grid_coords**: Es exactamente lo mismo pero en el dominio de las frecuencias. Recordemos que la transformada de Fourier permite resolver la ecuación diferencial fácilmente al cambiar el dominio del espacio a las frecuencias.
- **wave** y **wave_k**: Funcionan en conjunto con `grid_coords` y `k_grid_coords`. Tienen, para cada valor de la grilla, el valor de la función en ese punto. Básicamente, `wave[i][j] = phi(grid_coords[i][j])`.
- **K**: Matriz que almacena, para cada punto en el dominio de las frecuencias, el valor de $k^2 = k_x^2 + k_y^2$, necesario para la evolución temporal de la ecuación de onda en el dominio espectral.
- Los demás elementos son parámetros que manejan el paso del tiempo y la velocidad de las olas. El primero no afecta la precisión de la simulación, dado que la solución utilizada es analítica. El segundo, `wave_speed`, es un parámetro de la ecuación diferencial.

Este struct funciona gracias a tres métodos principales: `create_wave_simulation`, `add_wave_source` y `wave_sim_step`.

2.3.1. create_wave_simulation

Inicializa la simulación configurando los dominios espacial y frecuencial para resolver $\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u$ con FFT:

- Aloca arrays `wave` y `wave_k` para amplitudes complejas en dominios espacial y frecuencial, `grid_coords` y `k_grid_coords` para coordenadas, y `K` para magnitudes del vector de onda
- Inicializa `grid_coords` con coordenadas (x, y) uniformes usando `start = -domain_size/2.0` y `step = domain_size/size`, centrando el dominio en $[-L/2, +L/2]$ con espaciado $dx = L/N$
- Implementa el mapeo FFT estándar donde frecuencias $k_x = j/(N \cdot dx)$ para $j \leq N/2$ y $k_x = (j - N)/(N \cdot dx)$ para $j > N/2$.
- Computa $|\vec{k}| = 2\pi\sqrt{k_x^2 + k_y^2}$ donde el factor 2π convierte frecuencias espaciales a números de onda.

2.3.2. add_wave_source

Añade fuente puntual gaussiana como condición inicial de la ecuación de onda:

- Computa `r_sq = (x_val - x_pos) * (x_val - x_pos) + (y_val - y_pos) * (y_val - y_pos)` para obtener r^2 desde la posición de la fuente
- Calcula `envelope = amplitude * exp(-r_sq / (width * width))` implementando $A(x, y) = A_0 e^{-r^2/w^2}$ que localiza espacialmente la fuente
- Genera `phase = frequency * r` donde $\phi(r) = f \cdot r$ crea frentes de onda concéntricos con número de onda f
- Suma `new_val = {envelope * cos(phase), envelope * sin(phase)}` al array `wave`, implementando $u(x, y) = A(x, y)e^{i\phi(r}$ mediante `complex_add`
- Copia `wave` a `wave_k` y aplica `fft2d(..., 0)` para obtener $\tilde{u}(\vec{k})$ donde evolucionará temporalmente la simulación

2.3.3. wave_sim_step

Evoluciona la simulación un paso temporal usando la solución exacta espectral:

- Calcula `omega = sim->wave_speed * sim->K[idx]` implementando $\omega = c|\vec{k}|$.
- Computa `phase = -omega * sim->dt` para el argumento del operador de evolución temporal $e^{-i\omega t}$
- Multiplica cada modo por `phase_factor = {cos(phase), sin(phase)}` usando `complex_mul`, implementando exactamente la solución analítica $\tilde{u}(\vec{k}, t) = \tilde{u}(\vec{k}, 0)e^{-i\omega(\vec{k})t}$
- Copia `wave_k` a `wave` y aplica `fft2d(..., 1)` para recuperar $u(x, y, t + dt)$ en el dominio espacial

Estos métodos son fundamentales y, como se mencionó previamente, son la base necesaria que nos permite acelerar la simulación en todos los backends. Ya aclarado esto, la implementación de la transformada de Fourier:

```
static void fft_1d(Complex *x, int n, int inverse)
{
    assert(n > 0 && (n & (n - 1)) == 0 && "La longitud debe ser potencia de 2");

    bit_reverse(x, n);

    for (int len = 2; len <= n; len <<= 1)
    {
        double angle = 2.0 * M_PI / len * (inverse ? 1 : -1);
        Complex w = {cos(angle), sin(angle)};

        for (int i = 0; i < n; i += len)
        {
            Complex wn = {1.0, 0.0};
            for (int j = 0; j < len / 2; j++)
            {
                Complex u = x[i + j];
                Complex v = complex_mul(x[i + j + len / 2], wn);
                x[i + j] = complex_add(u, v);
                x[i + j + len / 2] = complex_sub(u, v);
                wn = complex_mul(wn, w);
            }
        }
    }

    if (inverse)
    {
        for (int i = 0; i < n; i++)
        {
            x[i].real /= n;
            x[i].imag /= n;
        }
    }
}
```

2.4. C + ASM

La motivación de esta optimización es que necesitamos calcular la transformada todo el tiempo para volver desde el dominio de las frecuencias al dominio espacial, y esto requiere calcular la transformada en cada paso de la simulación.

La implementación en Assembler puede entenderse fácilmente haciendo un paralelismo línea por línea con la implementación en C. Sin embargo, hay algunos detalles interesantes que vale la pena mencionar.

2.4.1. Uso de la pila x87 para operaciones matemáticas

La arquitectura x86-64 incluye una pila de registros especializada, conocida como la **pila x87**, diseñada para operaciones matemáticas en punto flotante. Esta pila, compuesta por ocho registros de 80 bits (st0–st7), permite realizar cálculos complejos de manera eficiente, especialmente en operaciones trigonométricas y exponenciales, que son fundamentales para la Transformada de Fourier.

En la implementación en Assembly, la pila x87 se utiliza para calcular funciones como seno y coseno de un ángulo, aprovechando instrucciones dedicadas como `fsin` y `fcos`.

A continuacion se demuestra un ejemplo, equivalente a la operación en C `Complex w = cos(angle), sin(angle)`, en el que cargamos el ángulo en la pila, calculamos el seno, almacenamos el resultado en memoria y luego calcular el coseno sobre el mismo ángulo. Finalmente, ambos resultados se transfieren a registros xmm para su uso vectorizado.

```
.declarar_w:
fld     st0                             ; Copio el angulo devuelta en st0, st1 = angulo
fsin                                    ; st0 = sin(ang)   (ángulo sigue en st1)
```

```
fstp    qword [rsp]                    ; guardar sin en memoria
movhpd  xmm6, [rsp]                    ; xmm6 = [?, w_i]

fcos                                   ; st0 = cos(ang)
fstp    qword [rsp]                    ; guardar cos
movlpd   xmm6, [rsp]                    ; xmm6 = [w_r, w_i]
; (pila x87 vacía)
```

En este código, se observa cómo la pila x87 permite calcular ambas funciones trigonométricas sin necesidad de recalcular el ángulo ni acceder repetidamente a memoria, optimizando así el rendimiento en operaciones matemáticas intensivas.

2.4.2. Representación de números complejos

Dado que la Transformada de Fourier trabaja con números complejos, es fundamental definir cómo vamos a manejarlos en Assembler. Como vimos en la implementación de C, la función recibe un puntero a un arreglo de complejos. Como ya vimos antes, el tipo de dato `Complex` ocupa 128 bits, o dos doubles, por lo que resulta ideal usar registros xmm para operar con ellos. En este trabajo, trabajamos con números complejos en los registros xmm de la siguiente manera:

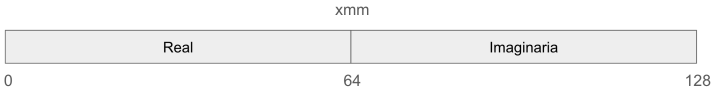

Figura 2: Representación de números complejos en memoria para la implementación en Assembly

Esta representación no solo permite hacer solo una lectura de memoria por cada complejo, lo cual es razonable, sino que además permite aprovechar las instrucciones de packed double para paralelizar sumas y restas, es decir, al sumar dos números complejos, la suma de la parte real e imaginaria se realiza simultáneamente.

```
; ---------------- x[i + j] = complex_add(u, v) --------------
movapd  xmm11, xmm0                    ; xmm11 = u_r, u_i
addpd   xmm11, xmm4                    ; xmm11 = u_r + v_r, u_i + v_i
movapd  [rdi],   xmm11
```

Otro detalle interesante de la implementación en Assembler es la utilización de una macro para el cálculo de la multiplicación compleja, dado que la misma se utiliza más de una vez. Esto resulta conveniente porque:

- Facilita la edición del código si queremos hacer optimizaciones posteriores
- Al añadirse al código al momento de compilar, no tiene efectos en la performance, como sí tendría utilizar una función

La macro se define de la siguiente manera:

```
; Macro para multiplicación compleja: result = a * b
; Parámetros: a, b, result
; Fórmula: (a_r + a_i*i) * (b_r + b_i*i) = (a_r*b_r - a_i*b_i) + (a_r*b_i + a_i*b_r)*i
%macro COMPLEX_MUL 3
    movapd  %3, %1                     ; t1 = a
    mulpd   %3, %2                     ; t1 = [ar*br, ai*bi]
    xorpd   %3, [rel COMPLEX_NEGHI]    ; t1 = [ar*br, -(ai*bi)]

    movapd  xmm15, %1
    shufpd  xmm15, xmm15, 1   ; xmm15 = [ai, ar]
    mulpd   xmm15, %2         ; xmm15 = [ai*br, ar*bi]

    haddpd  %3, xmm15         ; %3 = [ar*br - ai*bi, ai*br + ar*bi]
%endmacro
```

Y utiliza una mascara definida en la seccion .rodata que permite negar la parte imaginaria del numero.

2.5. C + AVX

Como se mencionó en la sección anterior, cada número complejo ocupa 128 bits, por lo que el uso de los registros xmm, a pesar de permitir paralelizar las operaciones efectivamente, solo soporta operar sobre un elemento del arreglo a la vez. Es por esto que se propone utilizar AVX para acelerar el ciclo interno de la transformada, habitualmente llamado *butterfly*.

AVX2 (Advanced Vector Extensions 2) es una extensión del conjunto de instrucciones x86-64 introducida por Intel en 2013 [6]. Los registros introducidos duplican el ancho de los registros SSE (128 bits), por lo que permiten operar sobre 4 valores double precision simultáneamente [7].

2.5.1. Representación de números complejos

El uso de estos registros es analogo al explicado en la implementacion de C + ASM, solo que guardando dos numeros complejo por cada registro ymm:

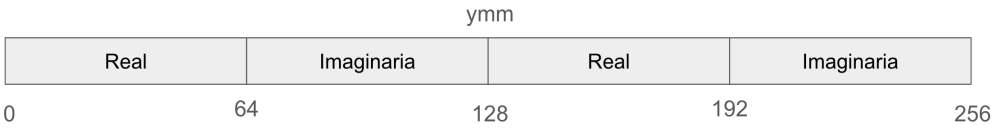

Figura 3: Esquema del ciclo butterfly vectorizado con AVX para la FFT

Para simplificar la implementación y aprovechar las optimizaciones realizadas por el compilador, utilizamos C para añadir estas operaciones *inline*.

2.5.2. Implementación del Ciclo Butterfly

La implementación AVX mantiene la misma estructura algorítmica que las versiones anteriores, con la diferencia clave en el ciclo butterfly, donde se procesan múltiples elementos simultáneamente. Las funciones que permiten esta aceleración son:

- `_mm256_loadu_pd`: Carga 256 bits de memoria como 4 packed doubles.
- `_mm256_setr_pd`: Carga 4 doubles en un registro avx.
- `_mm256_add_pd` y `_mm256_sub_pd`: Similares a addpd y subpd de Assembler, suma 4 packed doubles, posición a posición.
- `_mm256_mul_pd`: Multiplica 4 packed doubles, posición a posición.
- `_mm256_hadd_pd` y `_mm256_hsub_pd`: Realizan sumas y restas horizontales entre pares de elementos del registro.
- `_mm256_storeu_pd`: Guarda 4 packed doubles en memoria.

Y vale la pena detenerse a considerar la implementación de `complex_mul_simd`, dado que utiliza operaciones interesantes:

```
// Recibe: z1 = [a1, b1, a2, b2], z2 = [c1, d1, c2, d2]
// Devuelve: [(a1*c1-b1*d1), (a1*d1+b1*c1), (a2*c2-b2*d2), (a2*d2+b2*c2)]
static inline __m256d complex_mul_simd(__m256d z1, __m256d z2)
{
    __m256d ac_bd = _mm256_mul_pd(z1, z2);               // = [a1*c1, b1*d1, a2*c2, b2*d2]
    __m256d z2_swapped = _mm256_shuffle_pd(z2, z2, 0x5); // = [d1, c1, d2, c2] (intercambio real/imag)
    __m256d ad_bc = _mm256_mul_pd(z1, z2_swapped);       // = [a1*d1, b1*c1, a2*d2, b2*c2]

    __m256d real_parts = _mm256_hsub_pd(ac_bd, ac_bd); // = [a1*c1-b1*d1, a1*c1-b1*d1, ...]
    __m256d imag_parts = _mm256_hadd_pd(ad_bc, ad_bc); // = [a1*d1+b1*c1, a1*d1+b1*c1, ...]

    return _mm256_unpacklo_pd(real_parts, imag_parts); // Resultado = [a1*c1-b1*d1, a1*d1+b1*c1, ...]
}
```

3. Experimentos

Se realizaron experimentos para evaluar el rendimiento de cada backend implementado. Para cada backend, verifiqué el correcto funcionamiento mediante una simulacion interactiva, y medí el rendimiento en distintos tamaños.

Los experimentos se realizaron en un sistema con las siguientes especificaciones:

- **Procesador**: Intel x86-64 con soporte para AVX2
- **Sistema Operativo**: Linux 6.8.0-78-generic
- **Compilador**: GCC sin optimizaciones específicas (compilación por defecto)
- **Parámetros de simulación**:
  - Tamaño del dominio: 8.0 unidades
  - Velocidad de onda: 2.0 unidades/segundo
  - Intervalo de tiempo: 0.02 segundos
  - Pasos de simulación: 50 (para medición de rendimiento)

Se evaluaron cinco implementaciones diferentes:

1. **Python**: Implementación en Python puro como baseline
2. **NumPy**: Utilizando la biblioteca NumPy optimizada
3. **C**: Implementación en C con optimizaciones del compilador
4. **C + ASM**: C con rutinas críticas en Assembly x86-64
5. **C + AVX**: Utilizando extensiones AVX para paralelización vectorial

3.1. Visualización Interactiva

Obviamente, una parte fundamental del trabajo es poder visualizar interactivamente la simulación. Por ese motivo, se implementó una visualización que permite agregar ondas haciendo clic en cualquier lugar del campo.

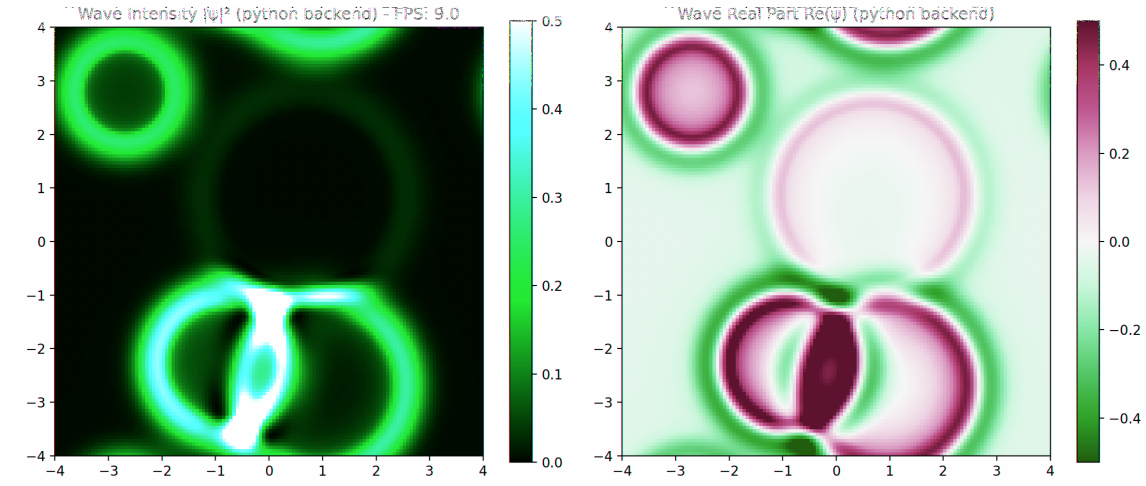

Figura 4: Visualización interactiva del simulador de ondas 2D

Se puede acceder y testear la simulacion siguiendo las instrucciones del README.md presente en el repositorio del proyecto: `https://github.com/JoaquinPolonuer/tp-final-orga2`.

3.2. Rendimiento por Tamaño de Grilla

Una vez verificado el correcto funcionamiento de cada backend, decidí medir más precisamente la performance. Para esto, dejé de lado la visualización y simplemente medí cantidad de pasos que puede computar la simulacion por segundo. Básicamente, nos importa cuánto tarda en correr la función `step` en cada uno de los backends.

Un paréntesis importante es que, a los efectos de la visualización, las implementaciones en C tienen un pequeño *overhead* porque deben convertir su grilla a un numpy array y esto consume un tiempo extra. En este trabajo evité lidiar con eso y simplemente medí el tiempo que tarda en correr cada paso de la simulación, porque es lo que decidí optimizar.

Cuadro 1: Rendimiento de diferentes implementaciones (steps per second)

Método	**16×16**	**32×32**	**64×64**	**128×128**	**256×256**	**512×512**	**1024×1024**
Python	626,5	154,3	37,3	9,1	2,1	0,5	0,1
ASM	54.928,0	16.541,7	4.321,6	1.064,1	245,3	57,2	12,0
C	56.649,2	14.737,5	3.341,2	789,0	178,1	41,6	9,4
C_AVX	**65.495,1**	18.320,5	4.338,2	1.026,6	230,9	53,7	12,2
Numpy	19.148,6	**12.521,8**	**5.472,7**	**1.610,4**	**368,8**	**88,5**	**15,7**

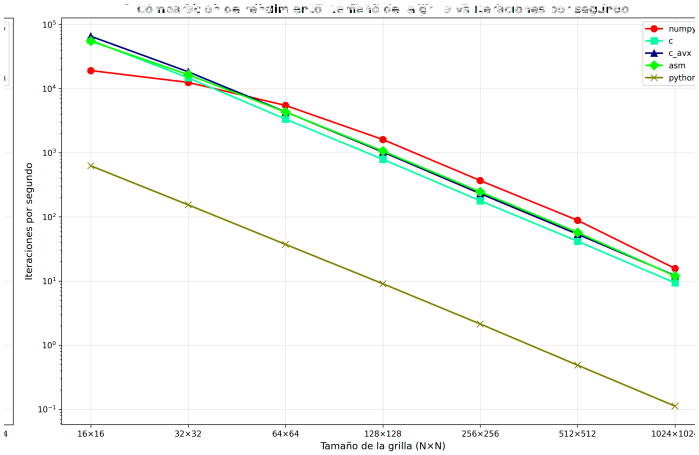

Figura 5: Comparación rendimiento entre los distintos backends. Se utiliza escala logaritmica en ambos ejes.

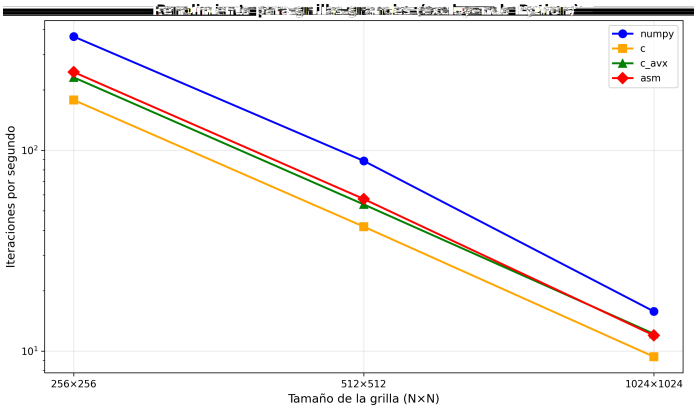

Figura 6: Comparación detallada del rendimiento de las implementaciones rapidas en grillas grandes.

3.3. Analisis

Los resultados experimentales presentados en la tabla **??** revelan patrones interesantes en el rendimiento de las diferentes implementaciones:

- **Python vs. Implementaciones Optimizadas:** La implementación en Python puro muestra un rendimiento dramáticamente inferior, siendo hasta 1000 veces más lenta que las implementaciones optimizadas.
- **NumPy:** Domina consistentemente en grillas grandes, evidenciando la efectividad de bibliotecas altamente optimizadas.
- La implementacion en **C** es una alternativa excelente, combinando simplicidad de implementacion con buena performance. Supera a Numpy en grillas chicas, y a pesar de no ser la mejora en grillas grandes, se mantiene competititiva.
- **C + AVX** y **C + ASM** mejoran ampliamente el rendimiento de la implementacion en C (alrededor de un 25 %) y se mantienen muy parejos entre ellos. Esto ultimo resulta llamativo, ya que el rendimiento es muy similar a pesar de que ambos implementan optimizaciones sumamente diferentes.

4. Conclusiones

En este trabajo se exploraron distintas implementaciones de un simulador fisico 2D basado en la Transformada Rapida de Fourier. El trabajo combina un extenso marco teorico matematico y fisico con una implementacion computacional altamente optimizada.

Se compararon 5 backends distintos, demostrando la importancia de utilizar optimizaciones en bajo nivel cuando la performance es critica. Los resultados revelaron que el estado del arte de la tecnología es altamente performante y dificil de superar, pero aún asi las implementaciones propias utilizando Assembler y paralelizacion con AVX alcanzaron un rendimiento muy competitivo.

Referencias

[1] W. A. Strauss, *Partial differential equations: an introduction*. John Wiley & Sons, 2008.

[2] G. Strang, *Computational science and engineering*. Wellesley-Cambridge Press, 2007, vol. 1.

[3] J. W. Cooley y J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, n.º 90, págs. 297-301, 1965.

[4] P. Duhamel y M. Vetterli, "Fast Fourier transforms: a tutorial review and a state of the art," *Signal processing*, vol. 19, n.º 4, págs. 259-299, 1990.

[5] C. R. Harris et al., "Array programming with NumPy," *Nature*, vol. 585, n.º 7825, págs. 357-362, 2020.

[6] Intel Corporation, "Intel Intrinsics Guide," *Intel Developer Zone*, 2016. dirección: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.

[7] A. Fog, *Optimizing software in C++*. Copenhagen University College of Engineering, 2016.