Fast Fourier Transforms for the 2D Wave Equation

Kevin Juan (kj89)

November 28th 2018

The Numerical Methods and Algorithms

The objective of this assignment was to numerically calculate a discrete Fourier Transform (DFT) on the 2D Wave Equation. Given a function $h(t_j)$, the forward DFT is as shown below:

$$H(f_n) = \sum_{j=0}^{N-1} h(t_j) e^{2\pi i f_n t_j} = \sum_{j=0}^{N-1} h_j e^{i\omega t_j} \tag{1}$$

An inverse DFT can also be defined for the function $H(f_n)$ as shown below:

$$h(t_j) = \frac{1}{N} \sum_{j=0}^{N-1} H(f_n) e^{-2\pi i f_n t_j} = \frac{1}{N} \sum_{j=0}^{N-1} H_n e^{-i\omega t_j} \tag{2}$$

The value f_n is the frequency, which can be defined as $f_n = n\Delta f$. The value Δf is defined as $\frac{1}{N\Delta t}$, and the value t_j is the discrete time defined as $t_j = j\Delta t$. The value ω is the angular frequency, which is $\omega = 2\pi f$. The original signal $h(t_j)$ is in the time domain, and its transform, $H(f_n)$, is converted to the frequency domain. In a discrete transform, both the signal and its transform are periodic with period N and are repeated infinitely in both directions. This means that large positive frequencies are equivalent to small negative frequencies. The DFT is calculated such that all the frequencies are positive, so the spectrum comes out in a rearranged order as shown below with the desired order:

$$\text{Desired Order: } n = -\frac{N}{2}, -\frac{N}{2}+1, \ldots, 0, 1, 2, \ldots, \frac{N}{2} \tag{3}$$

$$\text{Rearranged Order: } n = 0, 1, 2, \ldots, \frac{N}{2}, -\frac{N}{2}+1, \ldots, -2, -1 \tag{4}$$

For each transform, we must sum N terms and there are $N \times H_n$ values, so the compute time for the DFT is $\mathcal{O}(N^2)$. To remedy this slowness, we can use a Fast Fourier Transform. In dimensionless form, the transform is defined as follows:

$$H_n = \sum_{j=0}^{N-1} h_j e^{\frac{2\pi i n_j}{N}} \tag{5}$$

If N is highly composite, or composed of many small factors, then this sum can be done quickly with a Fast Fourier Transform. Factors of two, i.e. $N = 2^m$, are usually faster. We can then define j as follows:

$$j = j_0 2^0 + j_1 2^1 + j_2 2^2 + \ldots + j_{m-1} 2^{m-1} \tag{6}$$

The value for j_x can be either 0 or 1. This gives us the following definition for the transform H_n:

$$H_n = \sum_{j_{m-1}} \sum_{j_{m-2}} \cdots \sum_{j_1} \sum_{j_0} h_{j_0, j_1, j_2, \ldots, j_{m-1}} e^{2\pi i (j_0 + j_1 2 + j_2 4 + \ldots + j_{m-1} 2^{m-1}) n/N} \tag{7}$$

Some algebraic manipulation can lead us to the following rearranged equation:

$$H_n = \sum_{j_{m-1}} e^{2\pi i j_{m-1} 2^{m-1} n/N} \left(\sum_{j_{m-2}} e^{2\pi i j_{m-2} 2^{m-2} n/N} \left(\cdots \left(\sum_{j_1} H_n^{(0)} e^{2\pi i j_1 2 n/N} \right) \right) \cdots \right) \tag{8}$$

We see that this is a recursive calculation with a time complexity of $\mathcal{O}(N \log N)$. In practice, it is best to use an existing subroutine. One such example, and perhaps one of the most well-known subroutines, is `fftw3` from FFTW, which is used throughout this simulation. The subroutine relies on a planning stage to test a multitude of different FFT algorithms to find the fastest approach for a given problem size and CPU. The planning only needs to be done once, and can be reused throughout the simulation. This stage is slow, but if many FFTs need to be performed, this method of approach is generally much faster. More on FFTs can be found in §12.0-12.4 in Press et al[1], and documentation for the FFTW package can be found at `www.fftw.org`.

The 2D Wave Equation

This particular simulation will look at the 2D wave equation in Cartesian coordinates as shown below:

$$v^2 \left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right] \psi = \frac{\partial^2 \psi}{\partial t^2} \tag{9}$$

The value v is the velocity of the wave propagation, which is 343 m/s for this application. This partial differential equation can be solved using spectral methods, whereby

the solution is found in Fourier Transform space the transformed back into real space. The solution to Equation 9 can be expressed as a Fourier Series as shown below:

$$\psi(\vec{x},t) = \sum_{ij} a_{ij}(t)\exp(i\vec{k}_{ij}\cdot\vec{x}) \tag{10}$$

The vector $\vec{k}_{ij}$ is the 2D spatial frequency vector. By plugging Equation 10 into Equation 9, we get the following:

$$-v^2\sum_{ij} k_{ij}^2 a_{ij}(t)\exp(i\vec{k}_{ij}\cdot\vec{x}) = \sum_{ij} a_{ij}''(t)\exp(i\vec{k}_{ij}\cdot\vec{x}) \tag{11}$$

If we equate the coefficients for the exponential terms, we get the following:

$$-v^2 k_{ij}^2 a_{ij}(t) = a_{ij}''(t) \tag{12}$$

$$a_{ij}(t) = b_{ij}\exp(iv|\vec{k}|_{ij}t) + c_{ij}\exp(-iv|\vec{k}|_{ij}t) \tag{13}$$

$$\psi(\vec{x},t) = \sum_{ij}[b_{ij}\exp(iv|\vec{k}|_{ij}t) + c_{ij}\exp(-iv|\vec{k}|_{ij}t)]\exp(i\vec{k}_{ij}\cdot\vec{x}) \tag{14}$$

The constants b_{ij} and c_{ij} are complex and independent of time. If we specify that the wave is initially at rest :

$$\frac{\partial\psi(\vec{x},t=0)}{\partial t} = 0 = \sum_{ij}[b_{ij}-c_{ij}](iv|\vec{k}_{ij})\exp(i\vec{k}_{ij}\cdot\vec{x}) \tag{15}$$

We therefore know that the solution is restricted such that $b_{ij} = c_{ij}$ and that the solution to the wave equation for any time is given as such:

$$\psi(\vec{x},t) = \sum_{ij} d_{ij}\cos(v|\vec{k}_{ij}|t)\exp(i\vec{k}_{ij}\cdot\vec{x}) \tag{16}$$

$$\psi(\vec{x},t) = FT^{-1}[\Psi(\vec{k}_{ij}\cos(v|\vec{k}_{ij}|t)] \tag{17}$$

The coefficient d_{ij} is equivalent to $2b_{ij}$, and can be found by taking a Fourier Transform of the wave function at $t = 0$. These values for d_{ij} can then be multiplied by the cosine term in Equation 16 to get the wave function at any time. For this assignment, we assume that $L_x = L_y = 500$ m, $N_x = N_y = 512$, and the initial wave function is given by 3 Gaussians as defined below:

$$\psi(\vec{x},t=0) = \sum_{i=1}^{3} A_i\exp\left[-\frac{|\vec{x}-\vec{x}_i|^2}{s_i^2}\right] \tag{18}$$

The vector x_i is given by $(0.4, 0.5, 0.6)L_x$, the vector y_i is given by $(0.4, 0.6, 0.4)L_y$, and the vector s_i is given by $(10, 20, 10)$ all in m. The amplitudes of the waves are $A_i = (1, 2, 1)$.

Implementation and Results

The initial real part of the wave function is given below in the contour plot and surface plot:

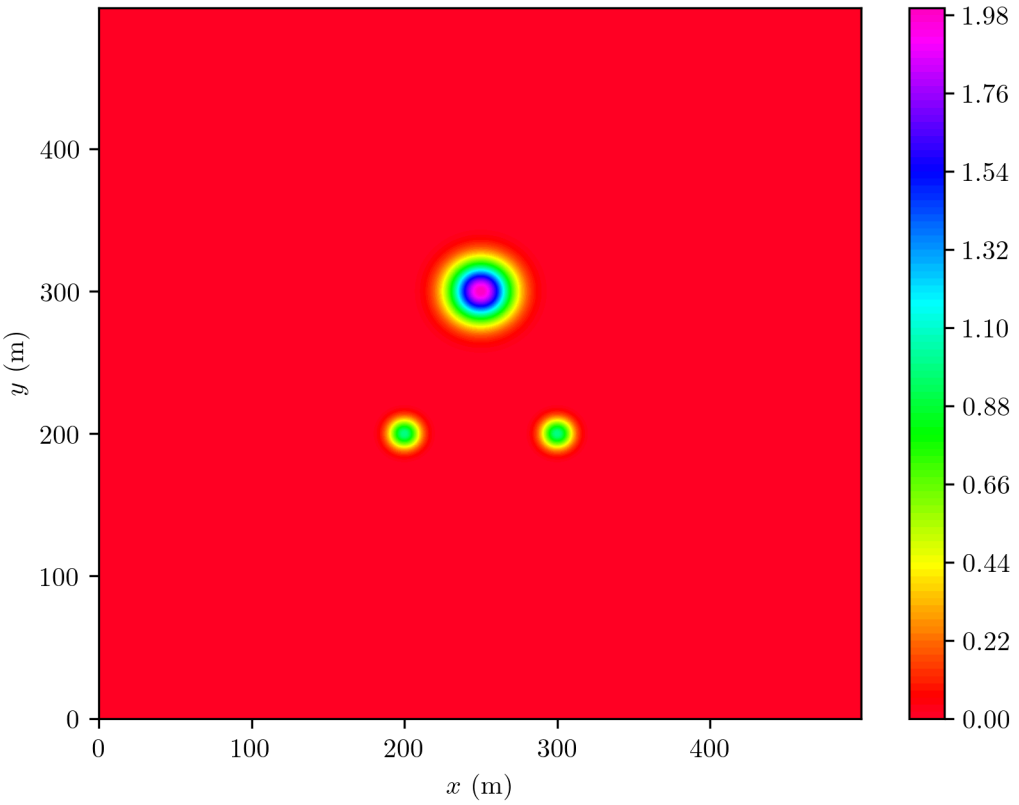

Figure 1: Contour plot of the wave function at time $t = 0$.

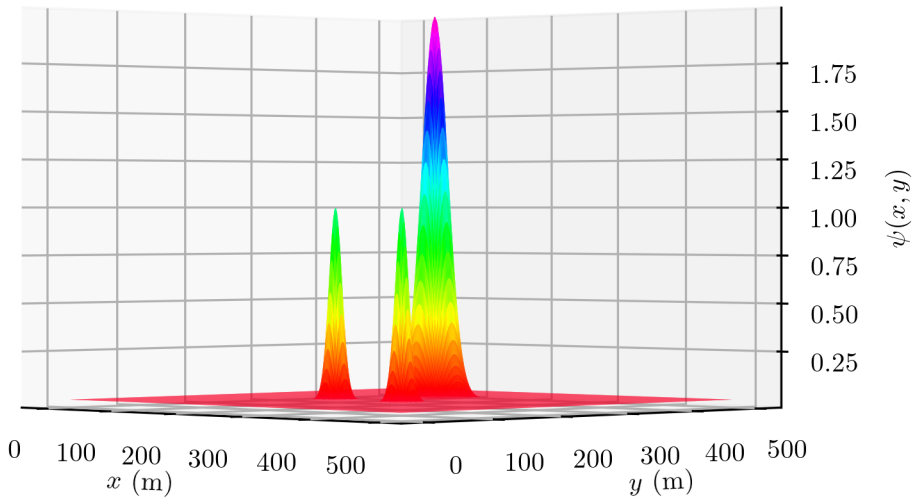

Figure 2: 3D surface plot of the wave function at time $t = 0$.

The wave function was then plotted at times $t = 0.1$, $t = 0.2$, and $t = 0.4$ as shown below:

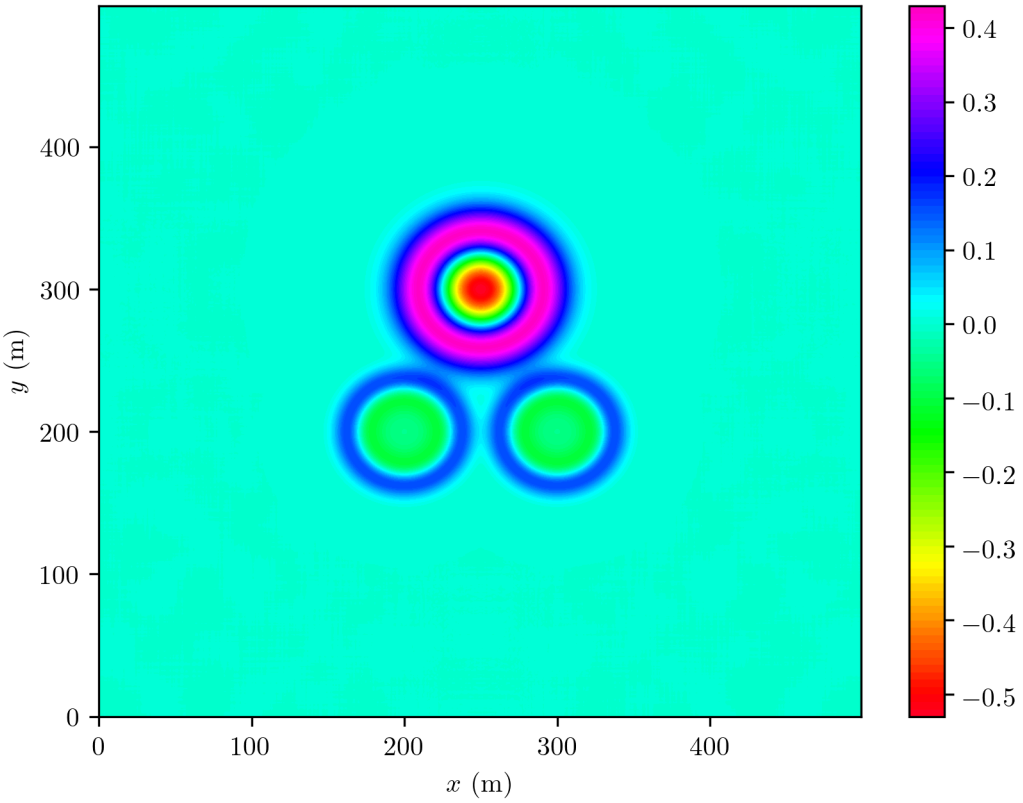

Figure 3: Contour plot of the wave function at time $t = 0.1$.

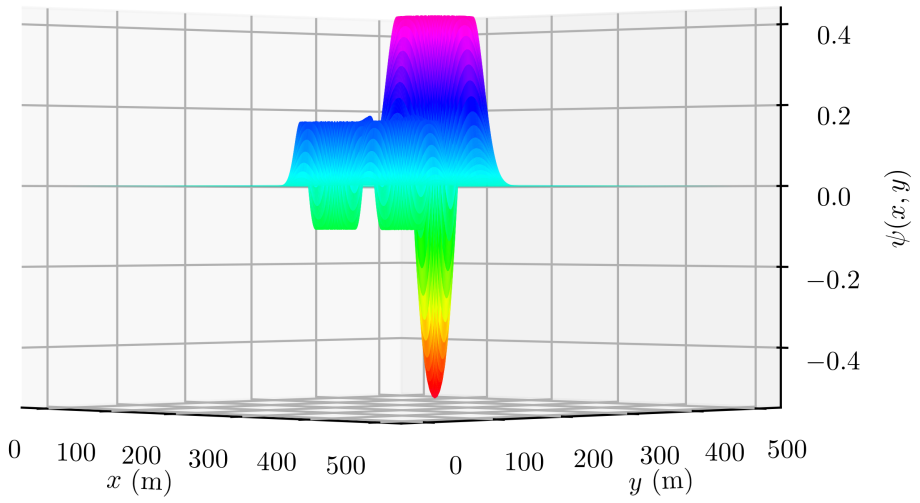

Figure 4: 3D surface plot of the wave function at time $t = 0.1$.

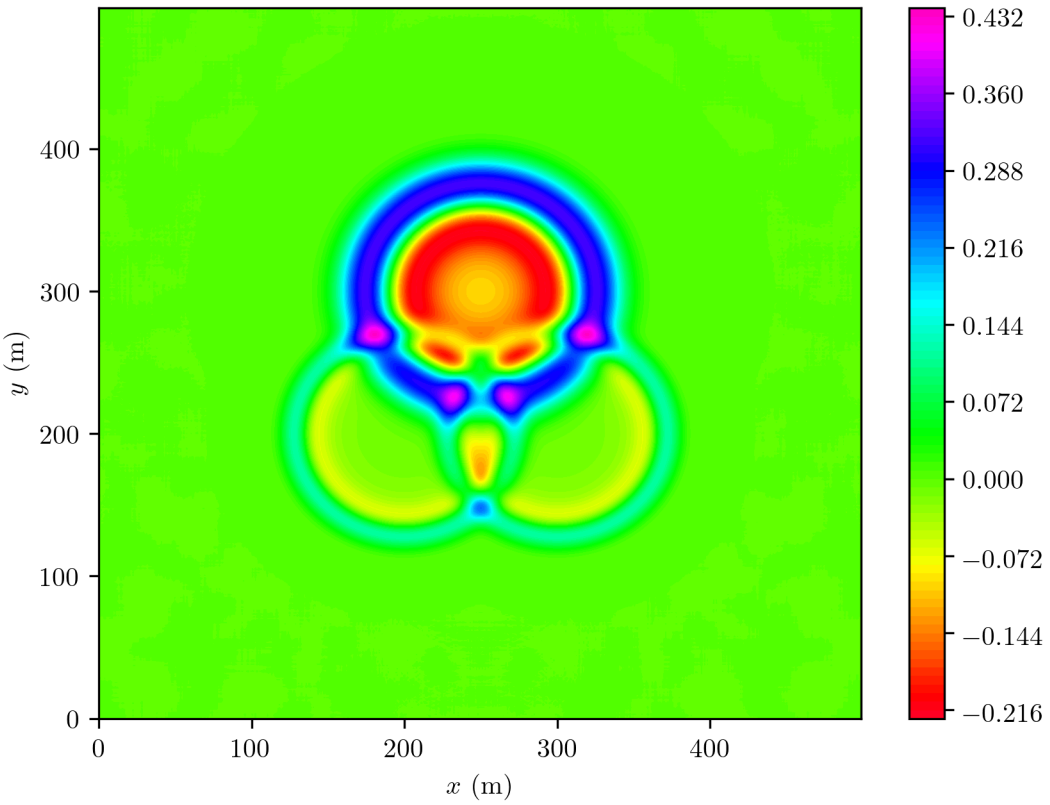

Figure 5: Contour plot of the wave function at time $t = 0.2$.

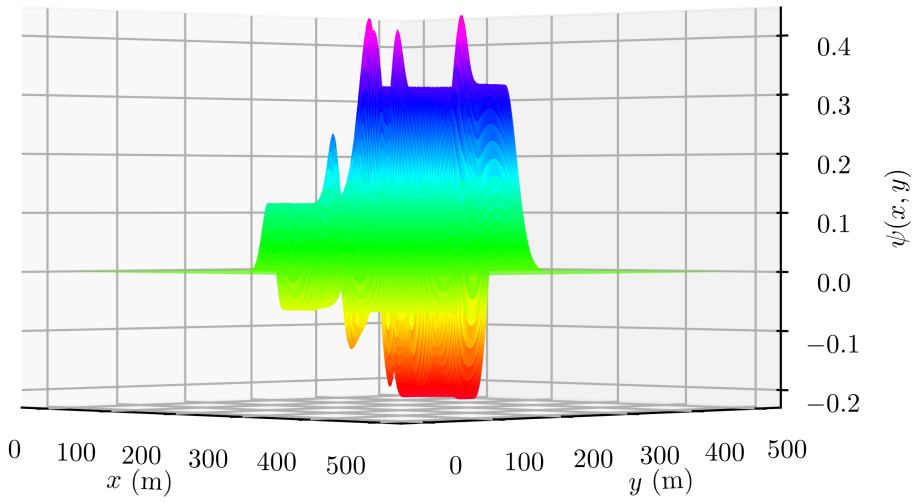

Figure 6: 3D surface plot of the wave function at time $t = 0.2$.

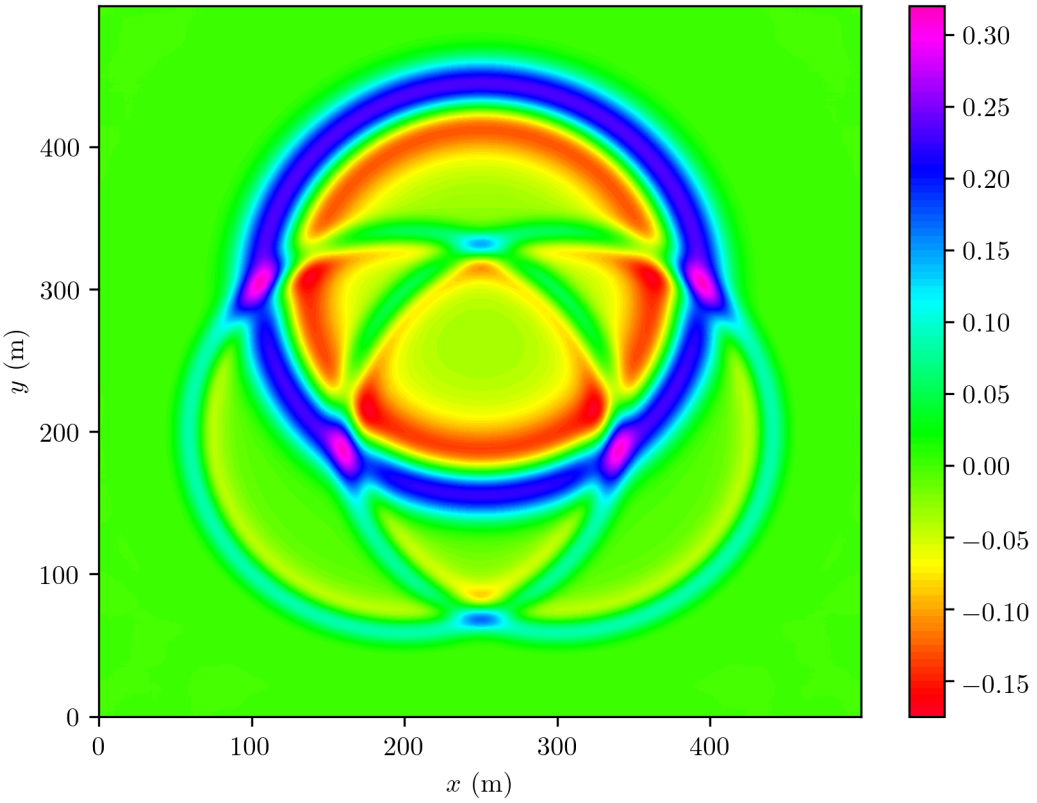

Figure 7: Contour plot of the wave function at time $t = 0.4$.

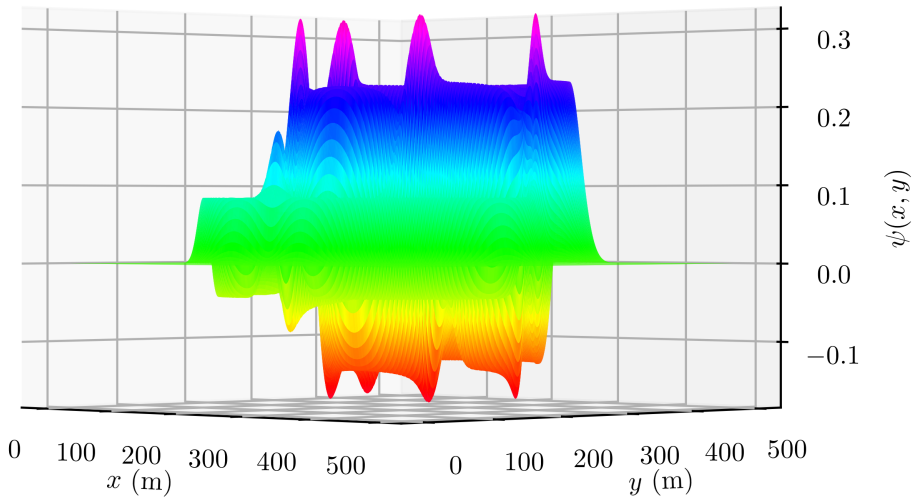

Figure 8: 3D surface plot of the wave function at time $t = 0.4$.

As expected, at times after $t = 0$, the Gaussians begin to show oscillatory behavior and eventually collide with each other as they disperse and grow in diameter.

Source Code

```
/* AEP 4380 Homework #9

Fast Fourier Transform using fftw3
Tested on a 2D Wave Equation

Run on a core i7 using clang 1000.11.45.2 on macOS Mojave

Kevin Juan 28 November 2018
*/
```

```
#include "fftw3.h"
#include <cmath>
#include <cstdlib>
#include <fstream>  //  stream file IO
#include <iomanip>  //  to format the output
#include <iostream> //  stream IO
#define _USE_MATH_DEFINES

using namespace std;

int main() {
    int Nx = 512, Ny = 512, i, j;
    double k, t = 0.1, Lx = 500.0, Ly = 500.0;
    double xi[] = {0.4 * Lx, 0.5 * Lx, 0.6 * Lx};
    double yi[] = {0.4 * Ly, 0.6 * Ly, 0.4 * Ly};
    double si[] = {10.0, 20.0, 10.0};
    double Ai[] = {1.0, 2.0, 1.0};
    double kx[Ny];
    double ky[Ny];
    double v = 343.0, x, y, init1, init2, init3;

    fftw_complex *psi;
    fftw_plan forward, inverse;
    psi = (fftw_complex *)fftw_malloc(Nx * Ny * sizeof(fftw_complex));

    if (psi == NULL) {
        cout << "Cannot allocate array y" << endl;
        exit(EXIT_FAILURE);
    }
    ofstream fpx;
    fpx.open("x_data.dat");
    if (fpx.fail()) {
        cout << "cannot open file" << endl;
        return (EXIT_SUCCESS);
    }
    ofstream fpy;
    fpy.open("y_data.dat");
    if (fpy.fail()) {
        cout << "cannot open file" << endl;
        return (EXIT_SUCCESS);
    }
    ofstream fpReInit;
    fpReInit.open("wave_Re_init_data.dat");
    if (fpReInit.fail()) {
        cout << "cannot open file" << endl;
```

```
    return (EXIT_SUCCESS);
}

// define plans
forward = fftw_plan_dft_2d(Nx, Ny, psi, psi, FFTW_FORWARD,
FFTW_ESTIMATE);
inverse = fftw_plan_dft_2d(Nx, Ny, psi, psi, FFTW_BACKWARD,
FFTW_ESTIMATE);

// initialize wave vectors
for (i = 0; i < Nx; i++) {
    if (i >= Nx / 2) {
        kx[i] = (i - Nx) / Lx;
        ky[i] = (i - Ny) / Ly;
    } else {
        kx[i] = i / Lx;
        ky[i] = i / Ly;
    }
}

// initialize waves
for (j = Ny - 1; j >= 0; j--) {
    y = j * Ly / Ny;
    fpy << y << endl;
    for (i = 0; i < Nx; i++) {
        x = i * Lx / Nx;
        if (j == 0) {
            fpx << x << endl;
        }
        init1 = Ai[0] * exp(-((x - xi[0]) * (x - xi[0]) + (y - yi[0])
        * (y - yi[0])) / (si[0] * si[0]));
        init2 = Ai[1] * exp(-((x - xi[1]) * (x - xi[1]) + (y - yi[1])
        * (y - yi[1])) / (si[1] * si[1]));
        init3 = Ai[2] * exp(-((x - xi[2]) * (x - xi[2]) + (y - yi[2])
        * (y - yi[2])) / (si[2] * si[2]));
        psi[j + i * Ny][0] = init1 + init2 + init3;
        psi[j + i * Ny][1] = 0.0;
        fpReInit << setw(15) << psi[j + i * Ny][0] << setw(15);
    }
    fpReInit << endl;
}

fftw_execute_dft(forward, psi, psi); // forward fft

ofstream fpRe;
```

```
    fpRe.open("wave_Re_data.dat");
    if (fpRe.fail()) {
        cout << "cannot open file" << endl;
        return (EXIT_SUCCESS);
    }

    // multiply dij by cosine
    for (i = 0; i < Nx; i++) {
        for (j = 0; j < Ny; j++) {
            k = sqrt(kx[i] * kx[i] + ky[j] * ky[j]);
            psi[j + i * Ny][0] =
                psi[j + i * Ny][0] * cos(2.0 * M_PI * v * k * t);
            psi[j + i * Ny][1] =
                psi[j + i * Ny][1] * cos(2.0 * M_PI * v * k * t);
        }
    }

    fftw_execute_dft(inverse, psi, psi); // inverse fft

    // write new wave to file
    for (j = Ny - 1; j >= 0; j--) {
        for (i = 0; i < Nx; i++) {
            fpRe << setw(15) << psi[j + i * Ny][0] / (Nx * Ny) << setw(15);
        }
        fpRe << endl;
    }
}
```

References

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, editors. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK ; New York, 3rd ed edition, 2007. OCLC: ocn123285342.