AEP 4380: Homework 10

Gregory Kaiser

December 9th, 2019

1 Problem Background and Solution Overview

The Fourier Transform is a powerful tool in the analysis of signals, but can also be applied to differential equations to solve a large variety of problems. An arbitrary function can be broken down into a set of Fourier components, each corresponding to a specific frequency:

$$\psi(x, y, t) = \sum_{ij} a_{ij} e^{i(k_j x + k_i y)} \tag{1}$$

where k_j and k_i are the x and y frequency vectors respectively, and a_{ij} is the Fourier coefficient corresponding to said frequency.[3]

Solutions to partial differential equations with constant coefficients can be found numerically using a discrete form of the Fourier transform, which is limited in the number of Fourier components that it can calculate. Using an online library called fftw3, a.k.a. the "Fastest Fourier Transform in the West," this assignment solves the wave equation in two dimensions given an initial square shaped and Gaussian pulse with zero initial velocity.[2]

The wave equation in two dimensions for a displacement $\psi(x, y)$ is

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

where v is the velocity of a wave through a given propagation medium (or the speed of an electromagnetic wave, as the case may be).

The function $\psi(x, y, t)$ is observed between 0 and 1000 meters in both the x and y directions ($L_x = L_y = 1000m$), with 512 samples taken in both directions as well ($N_x = N_y = 512$). $\psi(x, y, t = 0)$ is a square block and a Gaussian pulse. The square block has a height of 1 and is placed between $0.6N_x$ and $0.7N_x$ in the x direction and between $0.4N_y$ and $0.5N_y$ in the y direction. The Gaussian pulse has a peak of 2, and is centered at $0.5N_y$ and $0.45N_x$, with a width of 10 points in both directions.

$$\psi(x, y, t = 0) = 2 \exp\left[-\frac{(j - .45N_x)^2 + (i - .5N_y)^2}{10^2} \right] + \begin{cases} 1 & \text{if } 0.6N_x < j < 0.7N_x \\ & \text{AND } 0.4N_y < i < 0.5N_y \\ 0 & \text{else} \end{cases} \tag{3}$$

where i and j are the y coordinate and x coordinate respectively.[3]

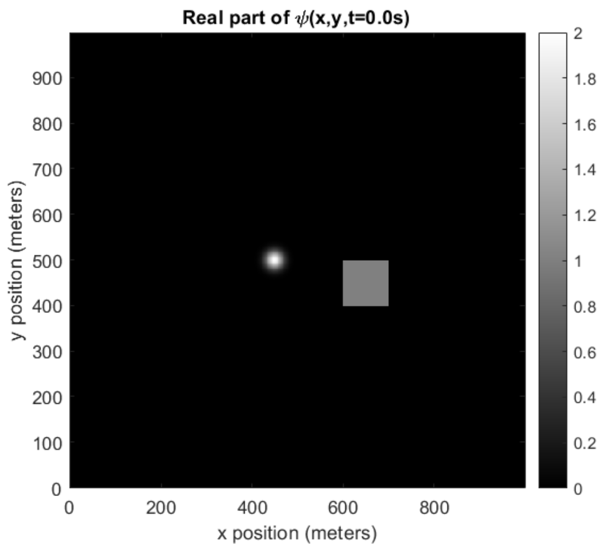

Figure 1: Initial condition of ψ.

2 Solution Description

By plugging the expression of the function ψ in terms of Fourier components (Equation 1) back into the wave equation (Equation 2), each Fourier component can be correlated with its own differential equation.

$$-v^2 \sum_{ij} (k_i^2 + k_j^2) a_{ij}(t) e^{i(k_j x + k_i y)} = \sum_{ij} a_{ij}^{''}(t) e^{i(k_j x + k_i y)} \tag{4}$$

$$-v^2 (k_i^2 + k_j^2) a_{ij}(t) = a_{ij}^{''}(t) \tag{5}$$

Guessing an exponential solution to the above differential equations with waves propagating in either direction means that:

$$a_{ij}(t) = b_{ij} e^{ivt\sqrt{k_i^2+k_j^2}} + c_{ij} e^{-ivt\sqrt{k_i^2+k_j^2}} \tag{6}$$

where b_{ij} and c_{ij} are arbitrary coefficients.[3]

Since the initial displacement starts at rest, this solution for a_{ij} can be plugged into Equation 1, and a derivative with respect to time should be zero:

$$\frac{\partial}{\partial t} \psi(x, y, t = 0) = \sum_{ij} [b_{ij} - c_{ij}]\, iv\sqrt{k_i^2 + k_j^2}) e^{i(k_j x + k_i y)} = 0 \tag{7}$$

Therefore $b_{ij} = c_{ij}$ and $\psi(x, y, t)$ can be written as:

$$\psi(x, y, t) = \sum_{ij} 2 b_{ij} cos(vt\sqrt{k_i^2 + k_j^2}) e^{i(k_j x + k_i y)} \tag{8}$$

which is recognizable as the inverse discrete Fourier transform of $2b_{ij}cos(vt\sqrt{k_i^2+k_j^2})$.[3]

So, to solve for $\psi(x,y,t)$, one must calculate $2b_{ij}$ using the forward Fourier transform of the initial $\psi(x,y,t=0)$. Those values are multiplied by the cosine function, and then that spectrum is moved back to real space with the inverse Fourier transform.[3]

2.1 Using the fftw library

To test the fftw library[2], a plain sin wave $y(x) = sin(2\pi x)$ was generated with 1000 points over a 10 second period and then fed into the fftw functions. A required planning step was also executed once, which estimated the best method of executing the Fourier Transform. This function is very fast compared to its competitors, partly because the "fast" fourier transform takes advantage of a recursive solution to the discrete Fourier Transform, allowing it to occur in $\mathcal{O}(N\log_2 N)$ time, as opposed to the usual $\mathcal{O}(N^2)$ time of a normal Fourier transform. The fftw library gains extra time advantages by the use of this planning step, but was not deeply investigated for this assignment.[2][3]

The fftw functions fill the input array with the magnitude of the coefficient on each Fourier component, but in a rearranged order. To test the sin wave's output, the magnitude array was rearranged to fix this, and then plotted against a precalculated array of frequency values:

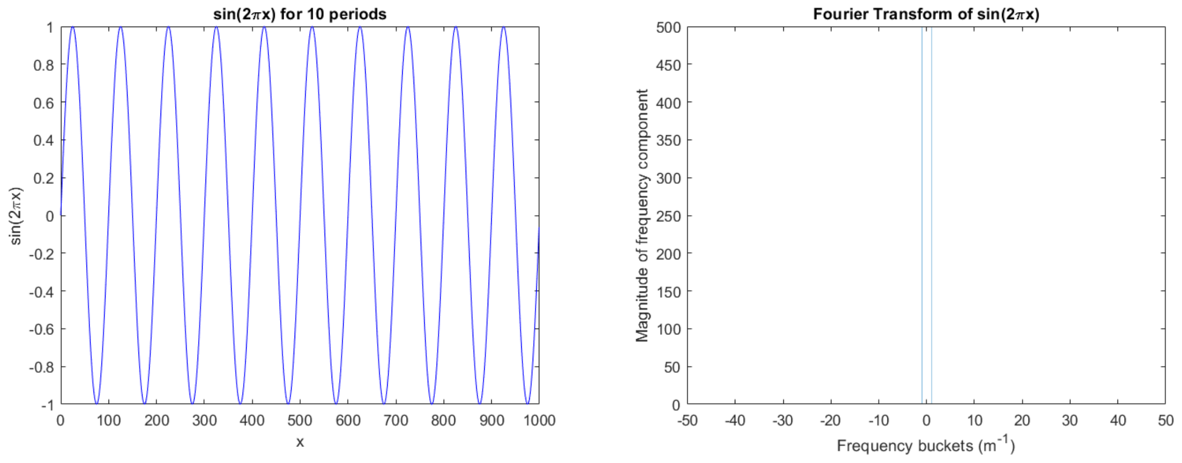

Figure 2: Fourier transform of a sin wave.

As expected, this produced two spikes at +1 and -1, corresponding to wavenumbers +1 and -1 m^{-1}, since the sin wave was generated with $|k| = 1m^{-1}$.

2.2 Solving the wave equation

$\psi(x,y,t=0)$ was initialized using the fftw library's version of memory allocation using fftw_malloc(), and offsets for the rows and columns of $\psi(x,y)$. Each piece of memory corresponds to a type called fftw_complex, which has a real and complex part. The offsets allowed the initial condition described in Equation 3 to be set.

Vectors corresponding to k_j and k_i are generated alongside $\psi(x,y,t=0)$, but in the order which corresponds to the output of the forward transform. This allows them to be easily correlated to their coefficients when used in the argument of the cosine function.

$$k_j = \begin{cases} \frac{j}{L_x} & \text{if } j < \frac{N_x}{2} \\ \frac{j}{L_x} - \frac{N_x}{L_x} & \text{else} \end{cases} \tag{9}$$

$$k_i = \begin{cases} \frac{i}{L_i} & \text{if } i < \frac{N_y}{2} \\ \frac{i}{L_y} - \frac{N_y}{L_y} & \text{else} \end{cases} \tag{10}$$

A factor of 2π is used to ensure that the wave propagates at the correct speed. This comes from the definition of the Fourier transform, which isn't taken into account in the derivation above.

In summary:
1) $2b_{ij} = FT\{\psi(x, y, t = 0)\}$
2) $\psi(x, y, t) = FT^{-1}\{2b_{ij}cos\left(2\pi vt\sqrt{k_i^2 + k_j^2}\right)\}$

3 Results and Interpretation

The images on the following page correspond to the real part of $\psi(x, y, t)$ at different times of propagation.

This looks like a pool of water, if an object or two were dropped in. The waves overlap and add together as waves should, and propagate outwards from their original location.

Since the top of the Gaussian pulse moves from 500m to about 750m in 0.8seconds, it is traveling at about 312m/s, which is close to the exact value of 343m/s (750m is a very rough estimate), confirming that the wave travels at the correct speed.

The color scaling on the right hand side of each figure was allowed to vary based on the input for ease of visualization. If that color scaling is held fixed, the waves lose strength as they propagate outwards, which also matches the physical intuition of spreading water waves (the ranges shown become smaller and smaller as time goes on).

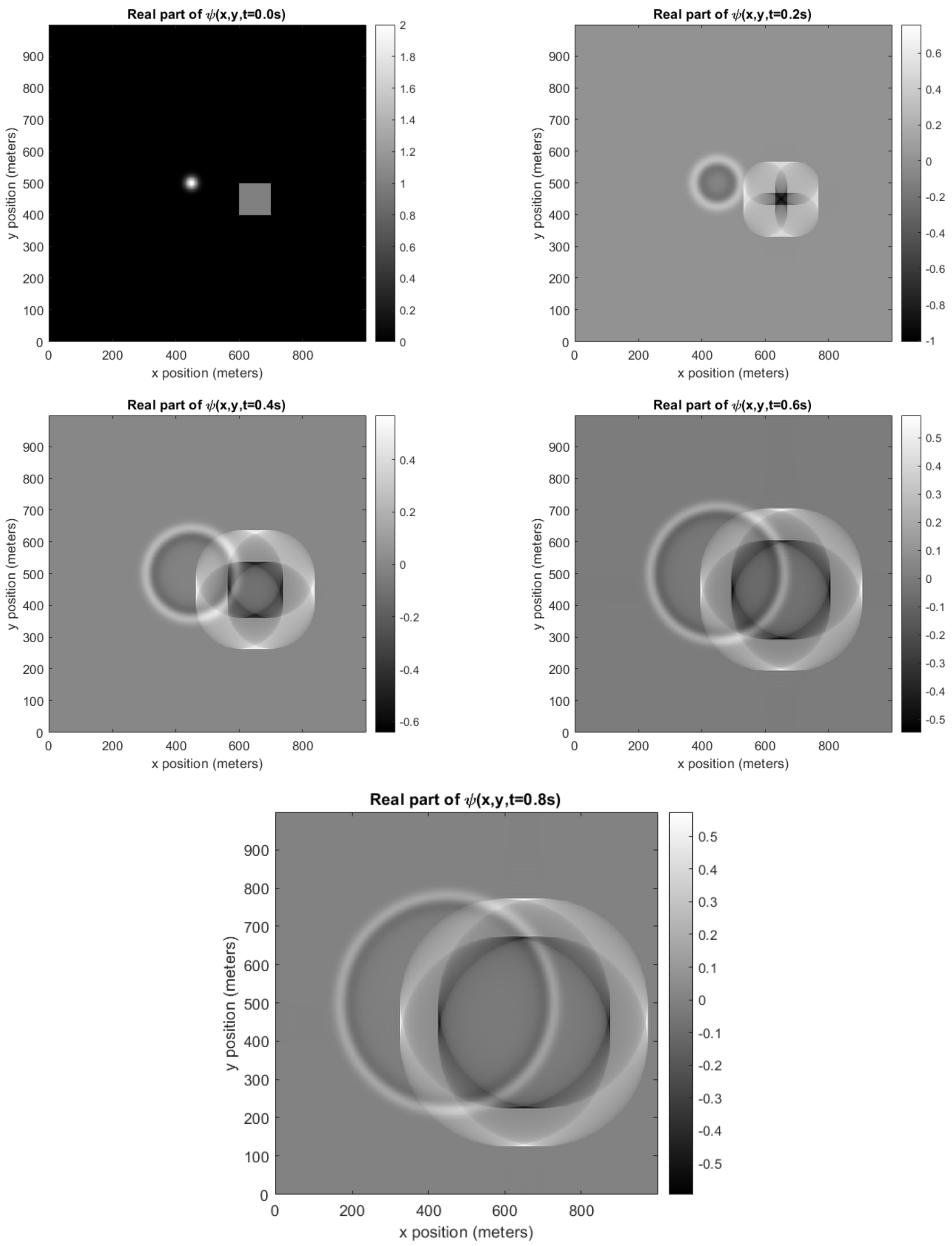

Figure 3: The real part of ψ plotted at t=0, 0.2, 0.4, 0.6, and 0.8s

This is a very powerful tool not just for wave propagation, but for differential equations of all kinds. Knowing how to use a library as powerful as fftw will surely come in handy in the future. This would also be great content for animation, since it is simulating a real-time set of events.

Unfortunately, it would require generating new solutions for each time step, but given the speed of fftw, this would not be difficult.

References

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes, The Art of Scientific Computing.* (3rd edit.), Cambridge Univ. Press, 2007, (ISBN 978-0-521-88068- 8, QA297 .N866 2007)

[2] Matteo Frigo and Steven G. Johnson, ”Fastest Fourier Transform in the West.” fftw.org. Accessed 12/7/19.

[3] Kirkland, Earl. *The Fast Fourier Transform (FFT) and Spectral Methods.* AEP 4380 Fall 2019 Assignment 10. https://courses.cit.cornell.edu/aep4380/secure/hw10f19.pdf

Source Code

```
/*  AEP 4380 Assignment 10
    The Fast Fourier Transform (FFT) and SPectral Methods

    Run on Windows core i7 with gcc version 8.2.0 (MinGW.org GCC-8.2.0-3)

    Gregory Kaiser December 9th 2019

*/
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>

//fft from FFTW3 at fftw.org
#include "fftw3/fftw3.h"

//Kirkland, E: https://courses.cit.cornell.edu/aep4380/secure/arrayt.hpp
#include "arrayt.hpp"//for vectors, matrices, ease of use

using namespace std; //makes writing code easier

double pi = 4.0*atan(1.0);

int main(){

    ofstream fp; //output file for sin wave testing
    fp.precision(7);
    fp.open("hw10_1.dat");
    if(fp.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
    }

    ofstream fpinit; //output file for initial cond
    fpinit.precision(7);
    fpinit.open("hw10_psi_i.dat");
    if(fpinit.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
    }

    ofstream fpx; //output file for x pixels
    fpx.precision(7);
    fpx.open("hw10_x.dat");
    if(fpx.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
    }

    ofstream fpy; //output file for y pixels
    fpy.precision(7);
    fpy.open("hw10_y.dat");
    if(fpy.fail()){
        cout<<"cannotopenfile"<<endl;
        return(EXIT_SUCCESS);
}

ofstream fpt; //output file for later time
fpt.precision(7);
fpt.open("hw10_psi_t.dat");
if(fpt.fail()){
    cout<<"cannotopenfile"<<endl;
    return(EXIT_SUCCESS);
}

//HW Assignment
//example code comes from E. Kirkland's
//"The Fast Fourier Transform (FFT) and Spectral Methods"
//https://courses.cit.cornell.edu/aep4380/secure/hw10f19.pdf
//test a 1d fft on a sin wave
fftw_plan planTf, planTi, planTf2, planTi2; //fft plans
fftw_complex *y, *y2;

int i,j,n=1000;//number of points to sample
double T=10;//sampling period
double x_pos, x_init = 0.0, dx = T/n; //sin wave generation constants
arrayt<double> k(n);//the frequency values at a given index
arrayt<double> Yk(n,2);//the amplitude at a given frequency index
//-----------TESTING A SINE WAVE-----------
//creates an array of complex numbers
y = (fftw_complex*) fftw_malloc(n*sizeof(fftw_complex));
if( NULL == y ) {
    cout << "Cannot allocate array y" << endl;
    exit(EXIT_FAILURE);
}
//forward dft plan
planTf = fftw_plan_dft_1d(n, y, y, FFTW_FORWARD, FFTW_ESTIMATE);
//create sin wave
for(i=0;i<n;i++){
    x_pos = x_init + i*dx;
    y[i][0] = sin(2*pi*x_pos);//creates a sine wave with wavelength 1
    y[i][1] = 0.0;
    k(i) = i/T-1/(2*dx);//this corresponds to the frequency in a given bucket
}
//perform forward dft
fftw_execute_dft(planTf, y, y);
//transform is now stored in y
//fix the order so that the results correspond to amplitudes in frequency space
for(i=0;i<n;i++){
    if(i<(n/2)){
        Yk(i+(n/2),0) = y[i][0];
        Yk(i+(n/2),1) = y[i][1];
    }
    else{
        Yk(i-(n/2),0) = y[i][0];
        Yk(i-(n/2),1) = y[i][1];
    }
}
//so now Yk and k have the amplitudes and frequencies of the fourier transform
for(i=0;i<n;i++){
    fp<<Yk(i,0)<<setw(15)<<Yk(i,1)<<setw(15)<<i<<setw(15)<<k(i)<<endl;
}
//-----------END TESTING SINE WAVE--------------
//-----------BEGIN 2D WAVE EQUATION---------------
```

```
int nx=512, ny=512;//number of samples
double lx=1000,ly=1000;//sampling region
double dx2=lx/nx, dy2=ly/ny;//spacing of sample points
double v=343;//velocity in m/s
arrayt<double> kx(nx);//the frequency values at a given x position
arrayt<double> ky(ny);//the frequency values at a given y position

//the following comes directly from Prof Kirkland's example for
//how to use the fft library
y2 = (fftw_complex*) fftw_malloc(nx*ny*sizeof(fftw_complex));
if( NULL == y2 ) {
    cout << "Cannot allocate array y2" << endl;
    exit(EXIT_FAILURE);
}
//forward plan
planTf2 = fftw_plan_dft_2d( nx, ny, y2, y2, FFTW_FORWARD, FFTW_ESTIMATE);
//inverse plan
planTi2 = fftw_plan_dft_2d( nx, ny, y2, y2, FFTW_BACKWARD, FFTW_ESTIMATE);

//initialize with gaussian and square wave pulse
int ox=1,oy=ny;//offsets for x and y positions
//gaussian pulse constants. pixel position of center and width of pulse
int gx = (int) nx*.45, gy = (int) ny*0.5, sa=10;
double mag=2, dist;//distance squared to center of gauss pulse
for(i=0;i<ny;i++){
    for(j=0;j<nx;j++){
        //to access point psi_{i,j}, use offsets ox and oy
        //square wave
        if((j>0.6*nx&&j<0.7*nx)&&(i>.4*ny&&i<.5*ny)){
            y2[oy*i+ox*j][0] = 1;
            y2[oy*i+ox*j][1] = 0;
        }
        else{
            y2[oy*i+ox*j][0] = 0;
            y2[oy*i+ox*j][1] = 0;
        }
        //add gaussian
        dist = (j-gx)*(j-gx)+(i-gy)*(i-gy);
        y2[oy*i+ox*j][0] += mag*exp(-dist/(sa*sa));
        y2[oy*i+ox*j][1] += 0;//pure real initial signal

        //printout the initial data in x y format
        fpinit<<y2[oy*i+ox*j][0]<<setw(15);

        if(i==ny-1){
            //create the x frequency range in the weird fft output order
            if(j<(nx/2)){
                kx(j) = j/lx;
            }
            else{
                kx(j) = j/lx-1/dx2;
            }
            fpx<<j<<endl;
        }
    }
    fpinit<<endl;
    //create the y frequency range in the weird fft output order
    if(i<(ny/2)){
        ky(i) = i/ly;
        }
        else{
            ky(i) = i/ly-1/dy2;
        }
        fpy<<i<<endl;
    }

    //execute fourier transform of initial condition
    fftw_execute_dft( planTf2, y2, y2);
    //y2 now contains d_ij, coefficients of each frequency (in weird order)
    //but the k vectors were generated in a weird order also

    double time = .8;//the time into the future where we want the solution
    double kmag = 0;//magnitude of the k vector
    //for each k vector coefficient (matrix of possible d_n), multiply by cos(2pivkt)
    for(i=0;i<ny;i++){
        for(j=0;j<nx;j++){
            kmag = sqrt(kx(j)*kx(j)+ky(i)*ky(i));
            y2[oy*i+ox*j][0] *= cos(2*pi*v*kmag*time);
            y2[oy*i+ox*j][1] *= cos(2*pi*v*kmag*time);
        }
    }

    //perform inverse transform
    fftw_execute_dft(planTi2, y2, y2);
    //y2 should now contain real space solution (normalize during printout)
    for(i=0;i<ny;i++){
        for(j=0;j<nx;j++){
            //printout the data in x y format
            fpt<<y2[oy*i+ox*j][0]/(nx*ny)<<setw(15);
        }
        fpt<<endl;
    }


    fp.close();
    fpinit.close();
    fpx.close();
    fpy.close();
    fpt.close();
    return(EXIT_SUCCESS);
} //end main
```