TIF330
Computational Continuum physics

Problem Set 2

Date: 2023-05-01

Students:
Afonso Sequeira Azenha

Problem 3

In this problem, we are asked to develop a `C++` implementation of the 2D Split-Step Spectral Solver for the Time-Dependent Schrödinger Equation (2D TDSE), in the following form:

$$i\frac{\partial}{\partial t}\psi(x,y,t) = \left(-\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2} + V(x,y)\right)\psi(x,y,t) \tag{1}$$

The coordinates belong to the region $-10 \leq x, y \leq 10$, with periodic boundary conditions, and $V(x,y)$ is the potential that does not change in time. Additionally, we consider the following form of the potential:

$$V(x,y) = -5\left(1 + \left(\frac{x}{5}\right)^2 + \left(\frac{y}{4}\right)^2\right)^{-4} \tag{2}$$

To help in its visualization, I also plot it:

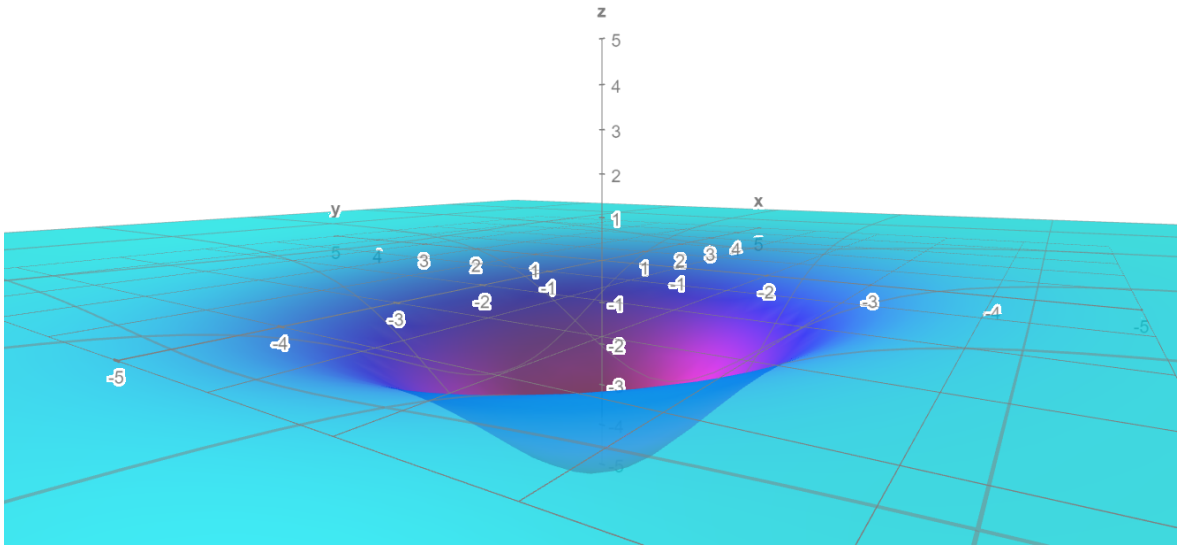

Figure 1: Potential $V(x,y)$. Has a minimum, $V(0,0) = -5$, at the origin and levels off to zero as we move further away from it.

The task is to perform a simulation of the wave function's evolution to see the spectrum of this two-dimensional potential. For that, we shall consider the evolution of $\psi(x,y,t)$, for $t \in [0, 100]$, starting from the initial state:

$$\psi(t=0) = \frac{1}{\sqrt{\pi}} e^{-\left((x-1)^2+(y-1)^2\right)} \tag{3}$$

For this task, I make use of the `fftw`[1] library, which makes computing the required Fourier transforms more direct. Following from our discussion in the lectures, we note that, in our case, the propagator is defined as such:

$$P_{AB}(t,0) = e^{\frac{t}{2}A(0)} e^{\frac{t}{2}B(t)} e^{\frac{t}{2}B(0)} e^{\frac{t}{2}A(t)} = e^{-i\frac{t}{2}V(x,y)} \mathcal{F}^{-1} e^{-ik^2 t} \mathcal{F} e^{-i\frac{t}{2}V(x,y)} \tag{4}$$

Here, we took $A = -iV(x, y)$ and $B = i\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)$. Also, note that $k^2 = k_x^2 + k_y^2$. By performing the specified Fourier transforms, we are able to obtain **only** diagonal operators (in their respective spaces), which heavily simplifies the implementation. There's also a handful of caveats that one needs to keep in mind in regards to the Fourier transform's implementation, using `fftw`. Namely:

1. After a full cycle of Fourier transform (FT) + Inverse FT, we need to normalize our results by the number of points in our sample. In our case, this will be the total number of points of the considered discretized grid, representing the 2D region: $-10 \leq x, y \leq 10$;

2. Additionally, `fftw` utilizes the standard "in-order" output ordering (See Ref. [2]). What this means is that the k values are not ordered like we'd (intuitively) expect, i.e., the k vector has the form:
`[0, k_1, ..., k_n-1, k_n, -k_n, -k_n-1, ..., -k_1]`.
This needs to be kept in mind when applying the operator e^{-ik^2t};

3. There is also the need to cast/re-interpret everything that goes into `fftw` as an `fftw_complex` type, which definitely adds another layer of complexity to the program, making it slightly less (easily) readable;

4. Finally, note that I will make use of the function `fftw_plan_dft_2d`, for performing a 2D Discrete Fourier Transform (DFT). This function returns both the real and imaginary parts of the FT. I can afterwards compute the spectrum (absolute value squared);

Before applying this to the actual problem, I set out to understand if my code was correct. Check the section **Benchmarking**, below, before proceeding forward. Anyways, using a time-step `dt = 0.1`, the following spectrum was obtained:

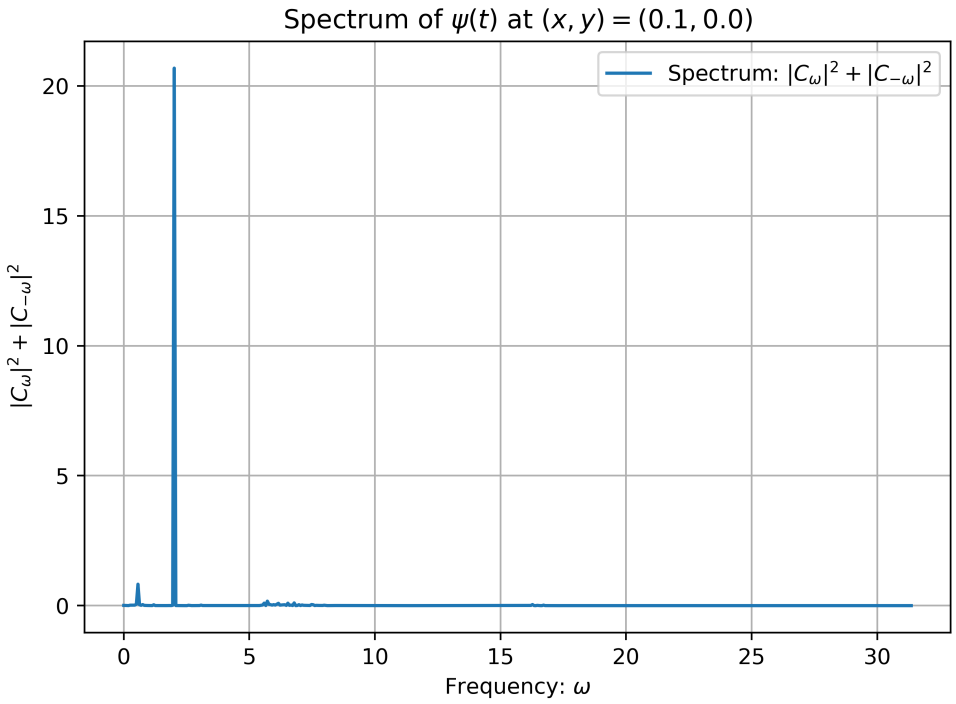

Figure 2: [Full] Spectrum of the 2D Wavefunction $\psi(0.1, 0.0)$.

And, in more detail, for $\omega \in [0, 5]$:

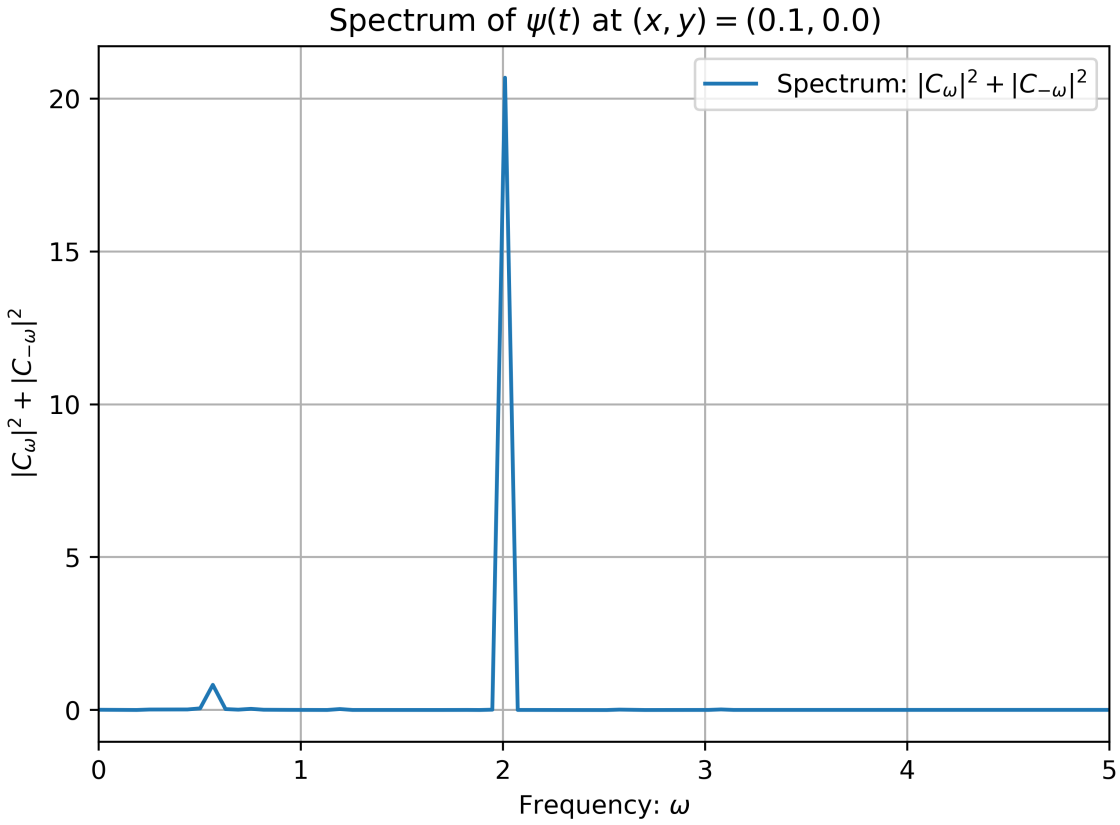

Figure 3: [Detail] Spectrum of the 2D Wavefunction $\psi(0.1, 0.0)$.

Physically speaking, the peaks in this spectrum correspond to the solution's (energy) eigenstates. The lowest frequency peak (The first one, at $\omega \approx 0.5$ appears to just be some noise), corresponding to this solution's ground-state, appears at $\omega = 2.01062$ (Adimensional units, corresponding to atomic units!). This is equivalent to an energy of 2.01062 Ha $\approx$ 54.71 eV. (For comparison: Hydrogen's ground-state energy = 13.6 eV). Additionally, I decided to repeat the entire calculation for different values of `dt`, to see if the same problem from the **Benchmarking** section was still present here. To my surprise, it appeared to have been somewhat mitigated. I tested for `dt = 0.5`, `dt = 0.2` and `dt = 0.05` and obtained the values $E_{1,1} = 2.45044$, $E_{1,1} = 1.94779$ and $E_{1,1} = 1.88496$, respectively, which are reasonably similar to what was obtained for `dt = 0.1`. So, maybe, it really had to do with the forcing of the Dirichlet boundary conditions... Although, then, I'm not sure how they should be implemented in the code (if we really wanted to solve the particle in a box). Either way, I'm fairly relieved to see that the spectrum doesn't change too much for different values of `dt`. Also, it makes sense that the fundamental energy is smaller than 5 (minimum of $V(x, y)$), as it thus corresponds to a bound state. After $T = 100$, we also notice that no other modes are present in the system, as per Fig. 2. These higher energy modes decay faster, which explains why they do not appear in Fig. 2. Additionally, since their energy would surely be higher than 5, they would no longer correspond to bound states. As such, they escape the system more easily. In order to verify this conjecture, I decided to plot the spectrum, but for only `T = 10`, with `dt = 0.01`.

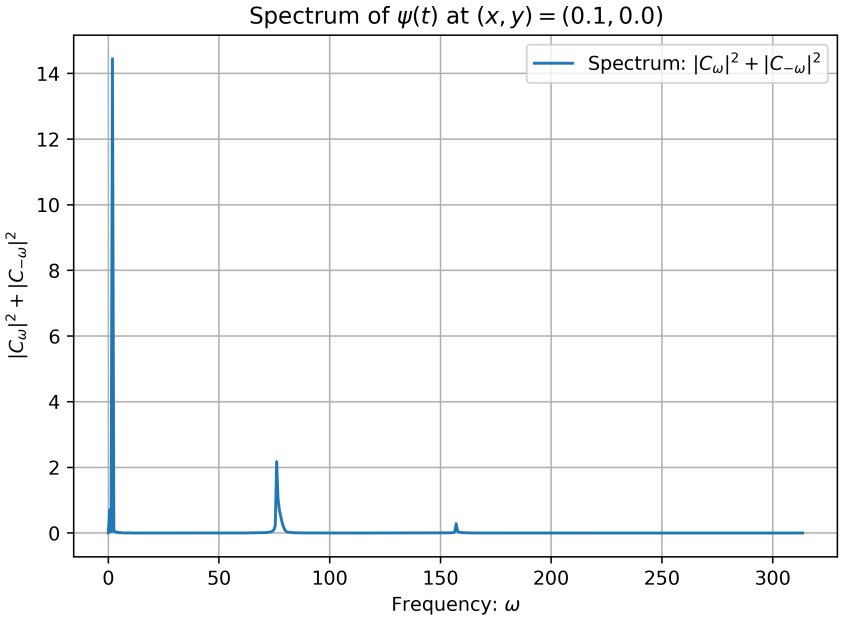

Figure 4: [Full] Spectrum of the 2D Wavefunction $\psi(0.1, 0.0)$, for `T = 10` and `dt = 0.01`.

As can be seen, there are indeed higher energy modes, which level off to zero as `T` increases! Here, the fundamental peak is located at $\omega = 1.88496$. In order to see this decay, I also repeat the spectrum for `T = 100`, but for `dt = 0.01`, so we can have access to these higher frequencies.

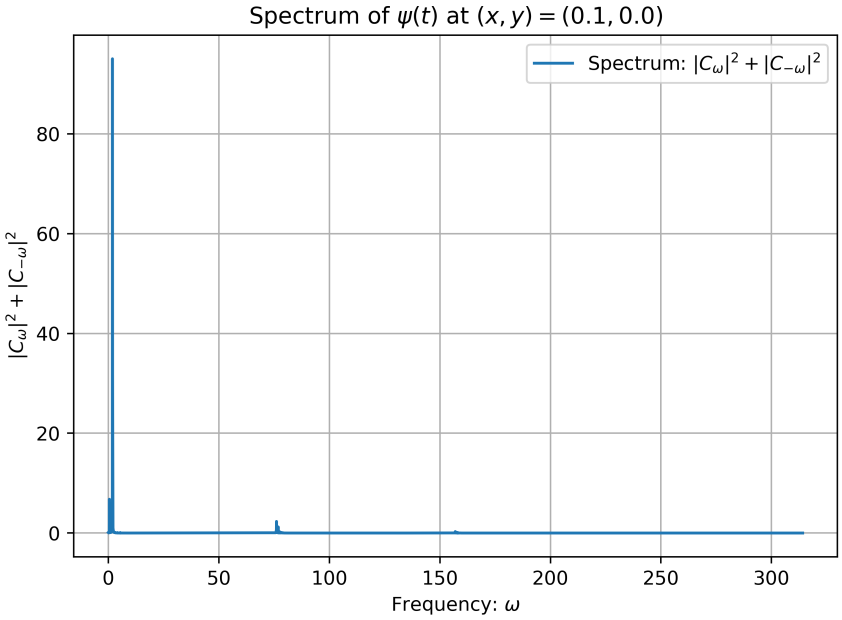

Figure 5: [Full] Spectrum of the 2D Wavefunction $\psi(0.1, 0.0)$, for `T = 100` and `dt = 0.01`.

As expected, those higher energies have decayed (note the different scale on the y axis, also). With all that said, I believe I've answered all the asked questions.

Benchmarking

As a way to benchmark my code, so as to figure out if my implementation was correctly coded, I attempted to solve a simpler problem, whose solution we know analytically. I chose the particle in a (2D) box model. In this case, $V(x, y) = 0$, such that the 2D TDSE simplifies to:

$$i\frac{\partial}{\partial t}\psi(x, y, t) = \left(-\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2}\right)\psi(x, y, t) \tag{5}$$

The solution to this partial differential equation can be found by separation of variables. For additional details, check Ref. [3]. In a box of size 1 by 1 (With a normalized wavefunction), one obtains:

$$\psi(x, y, t) = 2e^{-iE_{n_x,n_y}t}\sin(n_x\pi x)\sin(n_y\pi y) \tag{6}$$

$$E_{n_x,n_y} = \frac{\pi^2}{2}[n_x^2 + n_y^2] \tag{7}$$

Note, however, that this problem assumes the Dirichlet boundary conditions that $\psi(t) = 0$ at the boundaries of our grid. This needs to be hard coded into our program, since, by default, the Split-Step Fourier Solver considers periodic boundary conditions, due to how the FFT works. In order to implement the required Dirichlet boundary conditions, we have to explicitly set $\psi(t) = 0$ at the boundaries after **each** Fourier transform, both forward and backward. Additionally, I had to re-normalize the wavefunction after every (time-)iteration, since we're are actively changing values at the boundary, which messes up the correct normalization. With all of that said, I attempted to obtain the fundamental energy of the particle in a box model, which should be: $E_{1,1} = \pi^2 \approx 9.870$, for $n_x = n_y = 1$. For `gridX = gridY = 201`, `T = 100` and `dt = 0.15`, the following spectrum was obtained[1].

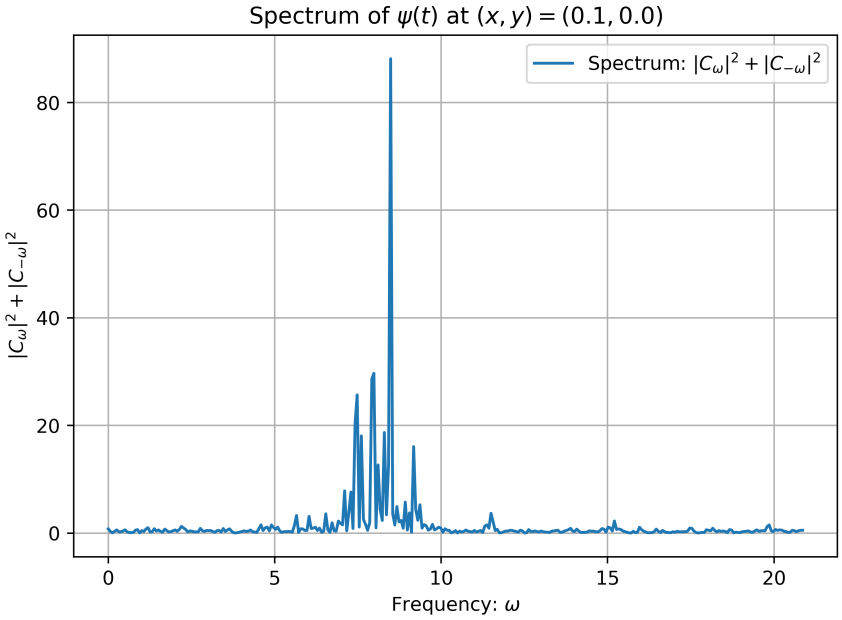

Figure 6: Obtained spectrum for $\psi(t = 0) = 1$ (Normalized afterwards), with `dt = 0.15`.

[1]Since we're interested in positive frequencies, I plot $|C_\omega|^2 + |C_{-\omega}|^2$, where the C's represent the

The peak corresponding to the fundamental frequency is located at $E_{1,1} = 8.4823$, which is slightly away from the expected theoretical value of $\pi^2 \approx 9.870^2$. I also noticed that the spectrum changed significantly for different values of `dt` (Which seemed strange, since the method should be exact for arbitrarily large time-steps, since $V(x, y) = 0$, as per the lecture notes). For instance, compare with the spectrum for `dt = 0.075`, half of what we considered previously.

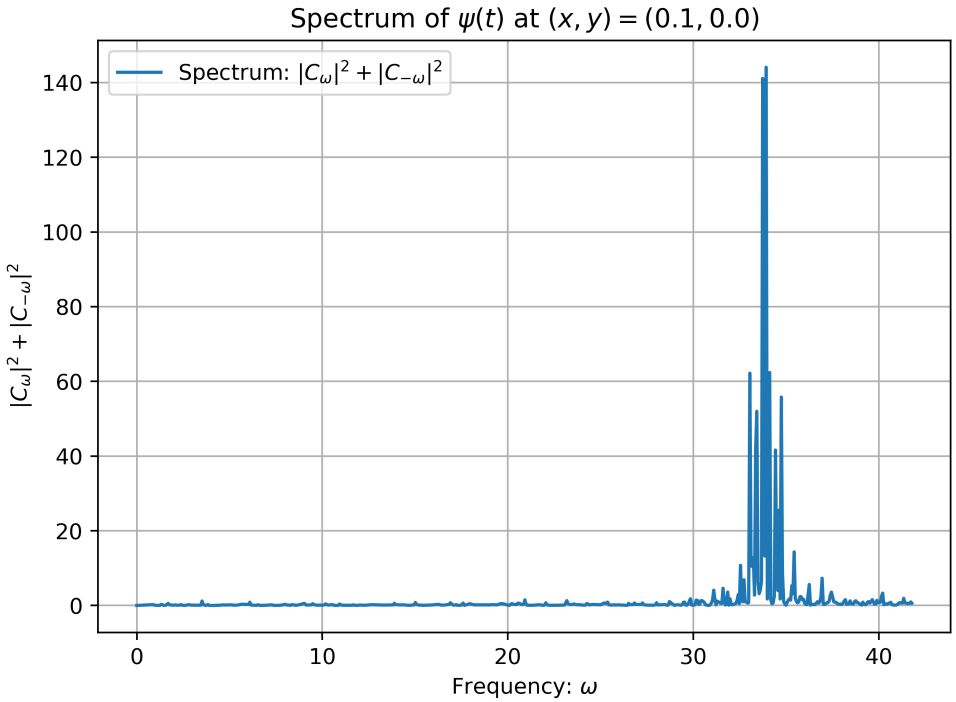

Figure 7: Obtained spectrum for $\psi(t = 0) = 1$ (Normalized afterwards), with `dt = 0.075`.

Here, we notice that the peak appears to have moved to higher frequency values, with $E_{1,1} = 33.9292$. I'm not entirely sure what might have caused this. I can only suspect that the scheme isn't as stable for different values of `dt`, which might lead to a fundamentally distinct accumulation of errors, for the different values of `dt`. This ultimately would result in different values of $E_{1,1}$. I also tried different values of `gridX(Y)` (number of points in the grid), as well as greater values of `T` (Equivalent to `tMax`). As far as the former is considered, the denser the grid, the better/more precise our results seemed to be. However, the change wasn't all too noticeable to justify the added computational cost. Pretty much the same could be said for the value of `T`: the higher it is, the more convergent our result appeared to be. But, still, `T = 100` seemed to be more than enough. After this analysis, I'm not too confident about my code... However, I've gone over it again and again and simply cannot find any more mistakes/bugs. Maybe it has something to do with trying to force Dirichlet boundary conditions and how that could interact with the FFT, but I'm unsure. Either way, I will use this and attempt to extract meaningful results for the more general problem, with an actual potential, $V(x, y) \neq 0$ (and periodic boundary conditions). As always,

Fourier coefficients.

[2]In this discussion, given how we use the adimensional TDSE, the frequency, ω, and the energy, E, have the same units, so I don't bother distinguishing them.

the developed code is presented in the appendix.

References

[1] fftw Website. Available at: https://www.fftw.org/;

[2] What FFTW actually computes. Available at: http://fftw.org/fftw3_doc/The-1d-Discrete-Fourier-Transform-_0028DFT_0029.html#The-1d-Discrete-Fourier-Transform-_0028DFT_0029;

[3] R. Becerril, F.S. Guzmán, A. Rendón-Romero, and S. Valdez-Alvarado. "Solving the time-dependent Schrodinger equation using finite difference methods". Available at: https://www.scielo.org.mx/pdf/rmfe/v54n2/v54n2a3.pdf;

Developed code

C++ code: `P3_Fast.cpp`

```
#include <iostream>
#include <cmath>
#include <vector>
#include <complex>
#include <fftw3.h>
#include <string>
#include <fstream>

/* Potential: Independent of time! */
double V(double x, double y){
    return(-5. * pow((1. + pow(x/5., 2.) + pow(y/4., 2.)), -4.));
}
/* Primitive normalization (Riemann sum type). Ideally, should
   implement more sophisticated 2D integration techniques. */
template <size_t gridX, size_t gridY>
double Get_Norm(std::complex<double> (&grid)[gridX][gridY], double dx,
   double dy){
    double norm = 0;
    for (size_t m = 0; m < gridX - 0; m++)
        for (size_t n = 0; n < gridY - 0; n++)
            norm += (abs(grid[m][n])*abs(grid[m][n])) * (dx*dy);
    return(norm);
}

template <size_t gridX, size_t gridY>
void Update_psiGrid(std::complex<double> (&grid)[gridX][gridY], double
   dx, double dy, double dkx, double dky, double tMax, double dt, int
   ind, std::vector<std::complex<double>> &psiXY){
    fftw_complex *out, *outR;
    fftw_plan p, pR;
    std::complex<double> *oneDarrayPointer = (std::complex<double> *)
   grid;
    out  = (fftw_complex *) fftw_malloc(gridX * gridY * sizeof(
   fftw_complex));
    outR = (fftw_complex *) fftw_malloc(gridX * gridY * sizeof(
   fftw_complex));
    p   = fftw_plan_dft_2d(gridX, gridY, (fftw_complex *)
   oneDarrayPointer, out,  FFTW_FORWARD,  FFTW_ESTIMATE);
    pR  = fftw_plan_dft_2d(gridX, gridY, (fftw_complex *)
   oneDarrayPointer, outR, FFTW_BACKWARD, FFTW_ESTIMATE);
    const std::complex<double> imU(0., 1.); /* Imaginary unit*/
    std::vector<double> kxVec, kyVec;

    /* Defining kVec's for later */
    for (size_t i = 0; i < gridX; i++)
    {
```

```
    if(i < gridX/2 + 1) /* Since I use an odd number of points, I
add +1 */
        {
            kxVec.push_back(dkx * i);
            kyVec.push_back(dky * i);
        }
    else /* Due to "fftw3's" weirdness... */
        {
            kxVec.push_back(dkx * (i - gridX));
            kyVec.push_back(dky * (i - gridX));
        }
}
/* Time loop */
for (size_t tIndex = 0; tIndex < int(tMax/dt); tIndex++)
{
    /* Applying exp(-i*t/2*V(0)) */
    for (size_t m = 0; m < gridX; m++)
        for (size_t n = 0; n < gridY; n++)
            grid[m][n] *= std::exp(-imU * dt/2. * V(-10. + m*dx,
-10. + n*dy));

    /* Applying 2D FFT: Check the other script for help */
    fftw_execute(p);
    for (size_t i = 0; i < gridX; i++)
        for (size_t j = 0; j < gridY; j++)
          grid[i][j] = std::complex(out[i*gridX + j][0], out[i*
gridX + j][1]);

    /* There's some room for improvement here. No need to reassign
everything to 'grid' right away. Can use 'out' */
    /* Applying exp(-i*k^2*t) */
    for (size_t i = 0; i < gridX; i++)
        for (size_t j = 0; j < gridY; j++)
            grid[i][j] *= std::exp(-imU * (kxVec[i]*kxVec[i] +
kyVec[j]*kyVec[j]) * dt);

    /* Applying Inverse 2D FFT: Check the other script for help */
    fftw_execute(pR);
    for (size_t i = 0; i < gridX; i++)
        for (size_t j = 0; j < gridY; j++)
            grid[i][j] = std::complex(outR[i*gridX + j][0], outR[i*
gridX + j][1])/double(gridX*gridY);

    /* Finally, applying exp(-i*t/2*V(t)); Same as before, since V
!= V(t) */
    for (size_t m = 0; m < gridX; m++)
        for (size_t n = 0; n < gridY; n++)
            grid[m][n] *= std::exp(-imU * dt/2. * V(-10. + m*dx,
-10. + n*dy));
```

```
        std::cout << "Iteration " << tIndex << " of " << int(tMax/dt) -
    1 << "... (Norm_Check) = " << Get_Norm(grid, dx, dy) << std::endl;
        psiXY.push_back(grid[ind][gridY/2]); /* Thing to FT in time,
   later. Choose, here, between Psi and |Psi|^2 */
        /* psiXY.push_back(abs(grid[ind][gridY/2]) * abs(grid[ind][
   gridY/2])); */
    }
    fftw_destroy_plan(p);
    fftw_destroy_plan(pR);
    fftw_free(out);
    fftw_free(outR);
    /* Periodic boundary conditions are automatically verified. */
}

void Write_Vec_to_file(std::vector<std::complex<double>> &Vec, std::
   string filename)
{
    using namespace std;
    cout << "Entering \"Write_Vec_to_file\"... " << endl;
    fstream w_File;
    w_File.open(filename, fstream::out);
    for (int i = 0; i < size(Vec); i++)
    {
        if(i < (size(Vec) - 1)) w_File << real(Vec[i]) << "," << imag(
   Vec[i]) << endl;
        else                    w_File << real(Vec[i]) << "," << imag(
   Vec[i]);
    }
}

void Write_k_Vec_to_file(std::vector<double> &Vec, std::string filename
   )
{
    using namespace std;
    cout << "Entering \"Write_T_Vec_to_file\"... " << endl;
    fstream w_File;
    w_File.open(filename, fstream::out);
    for (int i = 0; i < size(Vec); i++)
    {
        if(i < (size(Vec) - 1)) w_File << Vec[i] << endl;
        else                    w_File << Vec[i];
    }
}

int main(int argc, const char** argv) {
    const int gridX    = 201; /* How should I choose these values? 2^i,
    3^i, etc. ?*/
    const int gridY    = 201;
```

```
 const double dx    = 20./(gridX - 1);
 const double dy    = 20./(gridY - 1);
 const double dkx   = 2.*M_PI/20.;
 const double dky   = 2.*M_PI/20.;
 const double tMax  = 100.;
 const double dt    = 0.01;
 double       norm  = 0.;
 int          ind   = 0;
 bool         Flag  = true;
 std::complex<double> psiGrid[gridX][gridY];
 std::vector<double> FreqVec;
 std::vector<std::complex<double>> psiXY;

 /* Note: 'm' indexes x (rows); 'n' indexes y (columns) */
 for (size_t m = 0; m < gridX; m++)
 {
     if (fabs(-10. + m*dx - 0.1) < 0.1 && Flag == true)
     {
         ind = m;
         std::cout << "ind = " << m << std::endl;
         Flag = false;
         // break;
     }
     for (size_t n = 0; n < gridY; n++)
         psiGrid[m][n] = 1./sqrt(M_PI) * exp(-(-10. + m*dx-1)*(-10.
+ m*dx-1)-(-10. + n*dy-1)*(-10. + n*dy-1)); /* Initial condition */
 }

 norm = Get_Norm(psiGrid, dx, dy);
 for (size_t m = 0; m < gridX; m++)
     for (size_t n = 0; n < gridY; n++)
         psiGrid[m][n] /= sqrt(norm); /* Primitive normalization */

 std::cout << "Before normalization: " << norm << "\nAfter
normalization : " << Get_Norm(psiGrid, dx, dy) << std::endl;

 /* Specific point: x = 0.1, y = 0; */
 std::cout << "x = 0.1: " << -10. + ind*dx << std::endl;
 std::cout << "y = 0  : " << -10. + gridY/2*dy << std::endl;

 Update_psiGrid(psiGrid, dx, dy, dkx, dky, tMax, dt, ind, psiXY);
 Write_Vec_to_file(psiXY, "PsiXY.txt");

 /* Final step: FFT psiXY (in time domain) */
 const int gridT = int(tMax/dt);
 const double dw = 2 * M_PI /tMax;
 std::vector<std::complex<double>> psiXY_FT;
 fftw_complex *outT;
 fftw_plan pT;
```

```
    outT = (fftw_complex *) fftw_malloc(gridT * sizeof(fftw_complex));
    pT   = fftw_plan_dft_1d(gridT, reinterpret_cast<fftw_complex*>(&
   psiXY[0]), outT, FFTW_FORWARD, FFTW_ESTIMATE);

    fftw_execute(pT);
    fftw_destroy_plan(pT);
    for (size_t i = 0; i < gridT; i++)
    {
            psiXY_FT.push_back(std::complex(outT[i][0], outT[i][1])/
   double(sqrt(gridT)));
            if(i < gridT/2)
                FreqVec.push_back(dw * i);
            else
                FreqVec.push_back(dw * (int(i) - gridT));
    }
    fftw_free(outT);
    Write_Vec_to_file(psiXY_FT, "PsiXY_FT.txt");
    Write_k_Vec_to_file(FreqVec, "wVec.txt");

    return 0;
}
```

Listing 1: C++ code utilized in Problem 3 for the provided potential. For ease of copying: `https://codeshare.io/OdwqQ7`.

C++ code: `P3_Box_Clean.cpp`

```
#include <iostream>
#include <cmath>
#include <vector>
#include <complex>
#include <fftw3.h>
#include <string>
#include <fstream>

/* Potential: Independent of time! */
double V(double x, double y){
    // return(-5. * pow((1. + pow(x/5., 2.) + pow(y/4., 2.)), -4.));
    return(0.); // Trying particle in a 2D box.
}
/* Primitive normalization (Riemann sum type). Ideally, should
   implement more sophisticated 2D integration techniques. */
template <size_t gridX, size_t gridY>
double Get_Norm(std::complex<double> (&grid)[gridX][gridY], double dx,
   double dy){
    double norm = 0;
    for (size_t m = 0; m < gridX - 0; m++)
        for (size_t n = 0; n < gridY - 0; n++)
            norm += (std::abs(grid[m][n])*std::abs(grid[m][n])) * (dx*
   dy);
```

```
    return(norm);
}

template <size_t gridX, size_t gridY>
void Update_psiGrid(std::complex<double> (&grid)[gridX][gridY], double
    dx, double dy, double dkx, double dky, double tMax, double dt, int
    ind, std::vector<std::complex<double>> &psiXY){
    fftw_plan p, pR;
    p   = fftw_plan_dft_2d(gridX, gridY, reinterpret_cast<fftw_complex
    *>(&grid[0]), reinterpret_cast<fftw_complex*>(&grid[0]),
    FFTW_FORWARD, FFTW_ESTIMATE);
    pR  = fftw_plan_dft_2d(gridX, gridY, reinterpret_cast<fftw_complex
    *>(&grid[0]), reinterpret_cast<fftw_complex*>(&grid[0]),
    FFTW_BACKWARD, FFTW_ESTIMATE);
    const std::complex<double> imU(0., 1.); /* Imaginary unit*/
    std::vector<double> kxVec(gridX, 0.), kyVec(gridY, 0.);

    /* Defining kVec's for later */
    for (size_t i = 0; i < gridX; i++)
    {
        kxVec[i] = (i < gridX/2 + 1) ? dkx * i : dkx * (i - gridX);
        kyVec[i] = (i < gridX/2 + 1) ? dky * i : dky * (i - gridX);
    }

    /* Time loop */
    for (size_t tIndex = 0; tIndex < int(tMax/dt); tIndex++)
    {
        /* Applying exp(-i*t/2*V(0)): V = 0, so we don't need to do
    anything. */

        /* Applying 2D FFT: Check the other script for help */
        fftw_execute(p);

        // Enforcing Psi = 0 at boundaries. For the particle in a 2D
    box case.
        for (size_t m = 0; m < gridX; m++)
        {
            for (size_t n = 0; n < gridY; n++)
            {
                if(m == 0)       {grid[m][n] = 0;};
                if(m == gridX-1) {grid[m][n] = 0;};
                if(n == 0)       {grid[m][n] = 0;};
                if(n == gridY-1) {grid[m][n] = 0;};
            }
        }

        /* There's some room for improvement here. No need to reassign
    everything to 'grid' right away. Can use 'out' */
        /* Applying exp(-i*k^2*t) */
```

```
        for (size_t i = 0; i < gridX; i++)
            for (size_t j = 0; j < gridY; j++)
                grid[i][j] *= std::exp(-1. * imU * (kxVec[i]*kxVec[i] +
    kyVec[j]*kyVec[j]) * dt);

        /* Applying Inverse 2D FFT: Check the other script for help */
        fftw_execute(pR);

        for (size_t i = 0; i < gridX; i++)
            for (size_t j = 0; j < gridY; j++)
                grid[i][j] /= double(gridX*gridY); /* Normalization */

        // Enforcing Psi = 0 at boundaries. For the particle in a 2D
   box case.
        for (size_t m = 0; m < gridX; m++)
        {
            for (size_t n = 0; n < gridY; n++)
            {
                if(m == 0)       {grid[m][n] = 0;};
                if(m == gridX-1) {grid[m][n] = 0;};
                if(n == 0)       {grid[m][n] = 0;};
                if(n == gridY-1) {grid[m][n] = 0;};
            }
        }

        /* Finally, applying exp(-i*t/2*V(t)); Same as before, since V
   != V(t): V = 0, so we don't need to do anything. */

        double norm = Get_Norm(grid, dx, dy); /* Normalizing */
        for (size_t m = 0; m < gridX; m++)
            for (size_t n = 0; n < gridY; n++)
                grid[m][n] /= sqrt(norm);

        std::cout << "Iteration " << tIndex << " of " << int(tMax/dt) -
    1 << "... (Norm_Check) = " << Get_Norm(grid, dx, dy) << std::endl;
        psiXY.push_back(grid[ind][gridY/2]);
    }
    fftw_destroy_plan(p);
    fftw_destroy_plan(pR);
}

void Write_Vec_to_file(std::vector<std::complex<double>> &Vec, std::
   string filename)
{
    using namespace std;
    cout << "Entering \"Write_Vec_to_file\"... " << endl;
    fstream w_File;
    w_File.open(filename, fstream::out);
    for (int i = 0; i < size(Vec); i++)
```

```
    {
        if(i < (size(Vec) - 1)) w_File << real(Vec[i]) << "," << imag(
    Vec[i]) << endl;
        else                    w_File << real(Vec[i]) << "," << imag(
    Vec[i]);
    }
}

void Write_k_Vec_to_file(std::vector<double> &Vec, std::string filename
    )
{
    using namespace std;
    cout << "Entering \"Write_T_Vec_to_file\"... " << endl;
    fstream w_File;
    w_File.open(filename, fstream::out);
    for (int i = 0; i < size(Vec); i++)
    {
        if(i < (size(Vec) - 1)) w_File << Vec[i] << endl;
        else                    w_File << Vec[i];
    }
}

int main(int argc, const char** argv) {
    const int gridX    = 201; /* How should I choose these values? 2^i,
    3^i, etc. ?*/
    const int gridY    = 201;
    const double dx    = 1./(gridX - 1);
    const double dy    = 1./(gridY - 1);
    const double dkx   = 2.*M_PI/1.;
    const double dky   = 2.*M_PI/1.;
    const double tMax  = 100.;
    const double dt    = 0.15;
    double       norm  = 0.;
    int          ind   = 0;
    bool         Flag  = true;
    static std::complex<double> psiGrid[gridX][gridY];
    std::vector<double> FreqVec;
    std::vector<std::complex<double>> psiXY;

    // Testing
    const std::complex<double> imU(0., 1.);

    /* Note: 'm' indexes x (rows); 'n' indexes y (columns) */
    for (size_t m = 0; m < gridX; m++)
    {
        if (fabs(-0. + m*dx - 0.1) < 0.01 && Flag == true)
        {
            ind = m;
            std::cout << "ind = " << m << std::endl;
```

```
            Flag = false;
        }
        for (size_t n = 0; n < gridY; n++)
        {
                psiGrid[m][n] = 1; // Trying particle in a 2D box.
                // psiGrid[m][n] = 1./sqrt(M_PI) * exp(-(-0. + m*dx-1)
*(-0. + m*dx-1)-(-0. + n*dy-1)*(-0. + n*dy-1)); /* Initial condition
 */
                if(m == 0)       {psiGrid[m][n] = 0;};
                if(m == gridX-1) {psiGrid[m][n] = 0;};
                if(n == 0)       {psiGrid[m][n] = 0;};
                if(n == gridY-1) {psiGrid[m][n] = 0;};
        }
    }

    norm = Get_Norm(psiGrid, dx, dy);
    for (size_t m = 0; m < gridX; m++)
        for (size_t n = 0; n < gridY; n++)
            psiGrid[m][n] /= sqrt(norm); /* Primitive normalization */

    std::cout << "Before normalization: " << norm << "\nAfter
normalization : " << Get_Norm(psiGrid, dx, dy) << std::endl;

    /* Specific point: x = 0.1, y = 0; */
    std::cout << "x = 0.1: " << -0. + ind*dx << std::endl;
    std::cout << "y = 0  : " << -0. + gridY/2*dy << std::endl;

    Update_psiGrid(psiGrid, dx, dy, dkx, dky, tMax, dt, ind, psiXY);
    Write_Vec_to_file(psiXY, "PsiXY.txt");

    /* Final step: FFT psiXY (in time domain) */
    const int gridT = int(tMax/dt);
    const double dw = 2 * M_PI / tMax;
    std::vector<std::complex<double>> psiXY_FT;
    fftw_complex *outT;
    fftw_plan pT;
    outT = (fftw_complex *) fftw_malloc(gridT * sizeof(fftw_complex));
    pT   = fftw_plan_dft_1d(gridT, reinterpret_cast<fftw_complex*>(&
psiXY[0]), outT, FFTW_FORWARD, FFTW_ESTIMATE);

    fftw_execute(pT);
    fftw_destroy_plan(pT);
    for (size_t i = 0; i < gridT; i++)
    {
            psiXY_FT.push_back(std::complex(outT[i][0], outT[i][1])/
double(sqrt(gridT)));
            if(i < gridT/2)
                FreqVec.push_back(dw * i);
            else
```

```
                FreqVec.push_back(dw * (int(i) - gridT));
    }
    fftw_free(outT);
    Write_Vec_to_file(psiXY_FT, "PsiXY_FT.txt");
    Write_k_Vec_to_file(FreqVec, "wVec.txt");

    return 0;
}
```

Listing 2: C++ code utilized in Problem 3 for the particle in a box model. For ease of copying: `https://codeshare.io/9OkqlB`.

Python script: `P3.py`

```
import matplotlib.pyplot as plt
import numpy as np
plt.style.use('fast')

# After talking with the professor, I think this is the best way of
    doing this.
def SqrAbs(Psi):
    absPsi = []
    for i in Psi:
        absPsi.append(i[0]**2 + i[1]**2)
    return absPsi

# Definitions/Reading results
wVec       = np.genfromtxt(r'wVec.txt', delimiter=',')
Psi        = np.genfromtxt(r'PsiXY.txt', delimiter=',')
PsiFT      = np.genfromtxt(r'PsiXY_FT.txt', delimiter=',')
AbsSqPsiFT = SqrAbs(PsiFT)

Spectrum = [AbsSqPsiFT[0]]
for i in range(1, int(len(wVec)/2.)):
    Spectrum.append(AbsSqPsiFT[i] + AbsSqPsiFT[len(wVec) - 1 - (i-1)])
# AbsSq.append(AbsSqPsiFT[int(len(wVec)/2.)]) # Ignoring the last \
    omega value, so our vector is symmetric.

# Comment/uncomment depending on what you want to plot. In order: Abs
    ^2/Real/imaginary part/.
plt.plot(wVec[:int(len(wVec)/2)], Spectrum, label = r"Spectrum: $|C_{\
    omega}|^2 + |C_{-\omega}|^2$")

# Real part
# plt.plot(wVec[:int(len(wVec)/2)], [PsiFT[i][0] for i in range(0, len(
    PsiFT))][:int(len(wVec)/2)], label = "Real part")
# plt.plot(wVec[int(len(wVec)/2):][::-1], [PsiFT[i][0] for i in range
    (0, len(PsiFT))][int(len(wVec)/2):][::-1], color = 'C0')

# Imaginary part
```

```
# plt.plot(wVec[:int(len(wVec)/2)], [PsiFT[i][1] for i in range(0, len(
    PsiFT))][:int(len(wVec)/2)], label = "Imaginary part")
# plt.plot(wVec[int(len(wVec)/2):][::-1], [PsiFT[i][1] for i in range
    (0, len(PsiFT))][int(len(wVec)/2):][::-1], color = 'C1')

plt.xlabel(r"Frequency: $\omega$")
# plt.ylabel(r"$\mathscr{F}(|\psi(0.1, 0.0)|^2)$")
# plt.ylabel(r"$\mathscr{F}(\psi(0.1, 0.0))$")
plt.ylabel(r"$|C_{\omega}|^2 + |C_{-\omega}|^2$")
plt.title(r"Spectrum of $\psi(t)$ at $(x, y) = (0.1, 0.0)$")
plt.grid(True)
plt.legend()
# plt.xlim(0., 5.)
# plt.ylim(0., 1.)
plt.tight_layout()
plt.savefig('Spectrum.png', dpi = 500)

print(f"Fundamental frequency = wVec[{np.argmax(Spectrum)}] = {wVec[np.
    argmax(Spectrum)]}")
```

Listing 3: Python code utilized in Problem 3 to plot the spectrum. For ease of copying: `https://codeshare.io/K8OqJY`.