PHY 410
Homework Assignment 9

Han Wen

Person No. 50096432

November 21, 2014

Abstract

The goal of this assignment is to get more familiar with PDE, as well as its application on typical mathematical physics equations

Contents

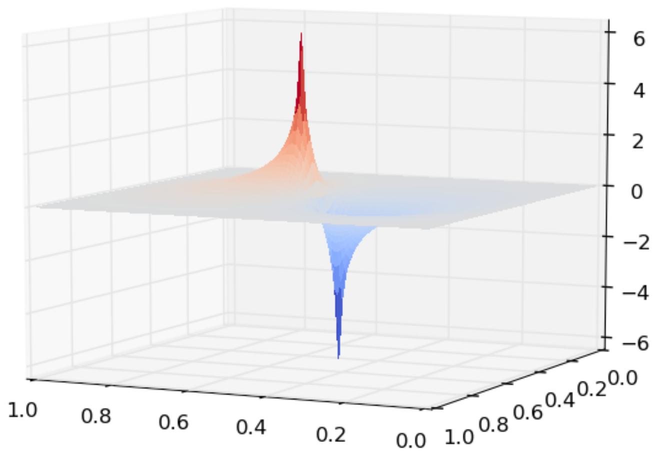

Figure 1: Electric potential for dipole

1 Problem 1

1.1 Description

Compute the electrostatic potential due to an electric dipole in a two-dimensional grounded metal box and compare with the expected exact solution (sum of the Coulomb potentials of two point charges) in a box of infinite size. Use at least two different methods (Jacobi/Gauss-Seidel, SOR, FFT, Multigrid).

1.2 Numerical Analysis

Change the charge density to that of a dipole, with number of interior points in x or y 100, desired accuracy 0.000001, for Jacobi method, it takes 9072 steps, 32.39s to finish, while using SOR method, it only takes 241 steps, 1.01s. Here is the result Fig 1

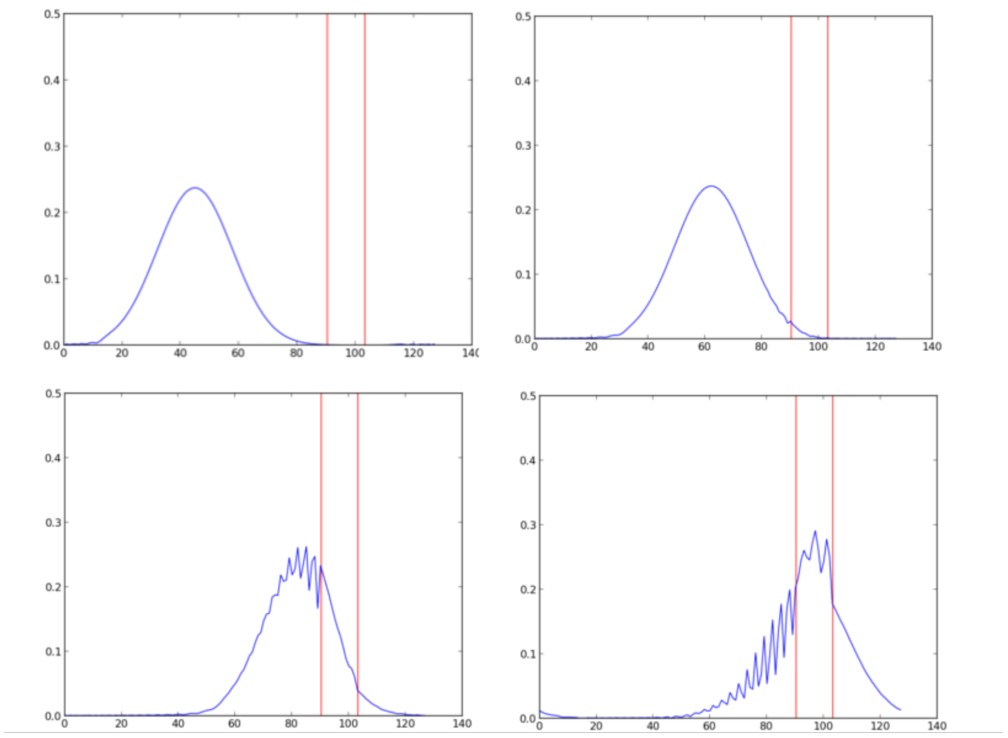

Figure 2: Wavepacket for E=1.0

2 Problem 2

2.1 Description

Use any wavepacket code to reproduce the movie frames in the article by Goldberg, Schey and Schwartz Am. J. Phys. 35, 177-186, 1967 or PDF copy. Modify the wavepacket code to study scattering from a potential of your choice. It might be most instructive to choose an example from your modern physics or quantum mechanics textbook. Describe the most interesting example you found.

2.2 Result

Firstly, I simulate the wavepacket when E=1.0 and 5.0. For E=1.0, the wavepacket frames: Fig. 2. For E=5.0, the frames: Fig. 3

Additionally, I chose the most common central potential, the column potential. And the result shows a wave run into the potential and then comes backwards, same as the real situation. Fig. 4

3 Problem 3

3.1 Description

In a wire chamber, several parallel wires are passed through a metal box. The wires are kept at a fixed potential V0, and the box edges are kept at ground.

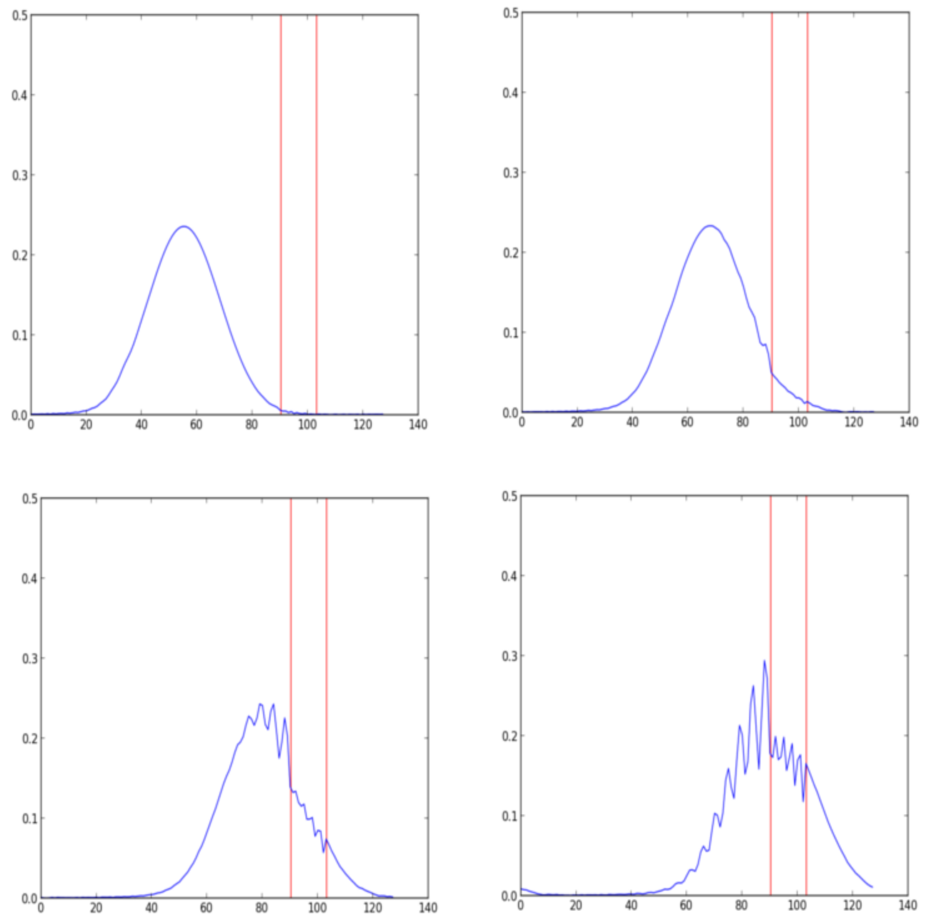

Figure 3: Wavepacket for E=5.0

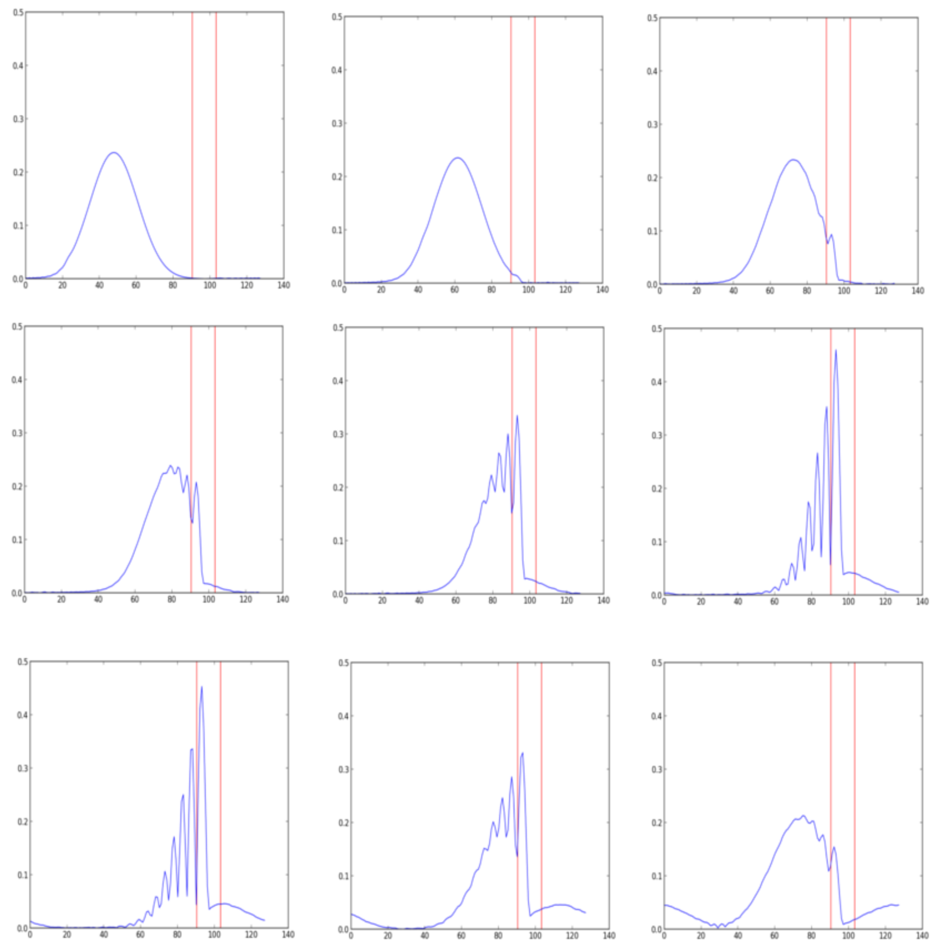

Figure 4: Wavepacket for column potential

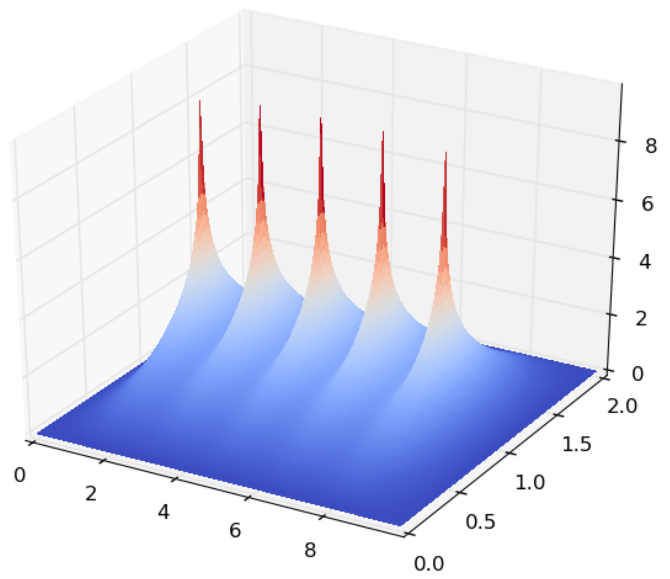

Figure 5: Potential of a wire chamber

Assuming an infinitely long z-direction (so this reduces to a two-dimensional problem), extend Problem 1 to compute the electrostatic potential for 5 wires equally spaced through a box of x-direction length Lx = 10 and y-direction length Ly=2. Then, compute the trajectory of a charged particle traveling at a forty-five degree angle to the bottom of the box numerically.

For the last part, you will have to pick one of the ODE methods (such as "kepler"), input the potential function that you have computed in this problem, and numerically take the gradient. This will be used as the derivative method.

3.2 Result

By modifying the code, change the geometric parameters, I generated the potential shown here Fig. 5

Consequently, the trajectory of the particle is shown here Fig. 6. For simplicity, I set the v=1 in both x and y direction and took the gradient directly as the acceleration.

For v=5 in both direction, its behaviour is shown here: Fig. 7

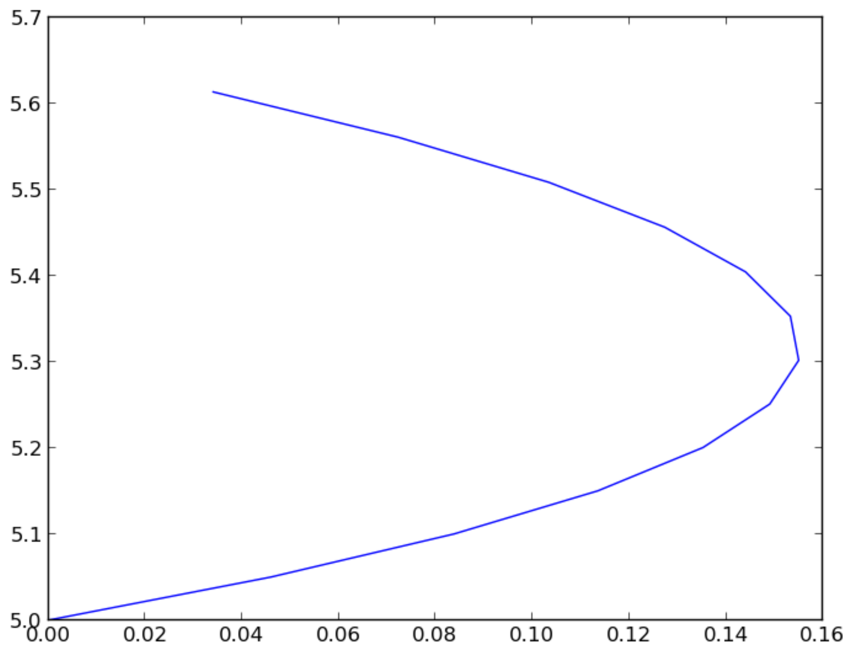

Figure 6: trajectory for v=1 in both direction

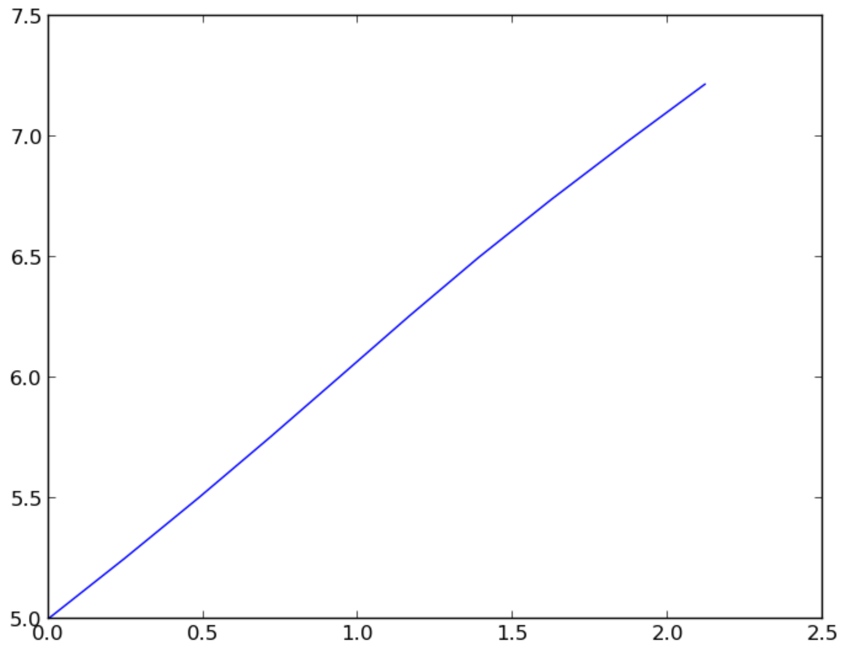

Figure 7: trajectory for v=5 in both direction

Acknowledgements

I discussed this assignment with my classmates and used material from the cited references, but this write-up is my own.

References

[1] PHY 410-505 Webpage, `http://www.physics.buffalo.edu/phy410-505`.

A Appendix

A.1 python code

The following python code was used to obtain the results in this report:

```
#include "cptstd.hpp"
#include "matrix.hpp"
using namespace cpt;

class Poisson {
public :
  Poisson( int iL = 50 )   :
    L(iL)
  {
    int N = L + 2;
    V = rho = V_new = Matrix<double,2>(N, N);


    h = 1 / double(L + 1);          // assume physical size in x and y = 1
    double q = 10;                  // point charge
    int i = N / 2;                  // center of lattice
    rho[i][i+5] = q / (h * h);        // charge density for dipole
    rho[i][i-5] = -q /(h * h);
    steps = 0;
  }

  void Jacobi() {
    // Jacobi algorithm for a single iterative step
    for (int i = 1; i <= L; i++)
      for (int j = 1; j <= L; j++)
        V_new[i][j] = 0.25 * (V[i - 1][j] + V[i + 1][j] +
                              V[i][j - 1] + V[i][j + 1] +
                              h * h * rho[i][j]);
  }
  double relative_error()
  {
    double error = 0;               // average relative error per lattice point
    int n = 0;                      // number of non-zero differences

    for (int i = 1; i <= L; i++)
      for (int j = 1; j <= L; j++) {
        if (V_new[i][j] != 0)
          if (V_new[i][j] != V[i][j]) {
            error += abs(1 - V[i][j] / V_new[i][j]);
            ++n;
          }
      }
    if (n != 0)
```

```
      error /= n;
    return error;
  }

  void Gauss_Seidel()
  {
    // copy V to V_new
    V_new = V;

    // Gauss-Seidel update in place
    for (int i = 1; i <= L; i++)
      for (int j = 1; j <= L; j++)
        V_new[i][j] = 0.25 * (V_new[i - 1][j] + V_new[i + 1][j] +
                              V_new[i][j - 1] + V_new[i][j + 1] +
                              h * h * rho[i][j]);
  }

  void successive_over_relaxation()   // using red-black checkerboard updating
  {
    // update even sites first
    for (int i = 1; i <= L; i++)
      for (int j = 1; j <= L; j++)
        if ((i + j) % 2 == 0)
          V_new[i][j] = (1 - omega) * V[i][j] + omega / 4 *
            (V[i - 1][j] + V[i + 1][j] +
             V[i][j - 1] + V[i][j + 1] +
             h * h * rho[i][j]);

    // update odd sites using updated even sites
    for (int i = 1; i <= L; i++)
      for (int j = 1; j <= L; j++)
        if ((i + j) % 2 != 0)
          V_new[i][j] = (1 - omega) * V[i][j] + omega / 4 *
            (V_new[i - 1][j] + V_new[i + 1][j] +
             V_new[i][j - 1] + V_new[i][j + 1] +
             h * h * rho[i][j]);
  }

  template< typename T>
  void iterate( T const & method,double accuracy)
  {
    clock_t t0 = clock();

    while (true) {
      (this->*method)();
      ++steps;
      double error = relative_error();
      if (error < accuracy)
```

```
        break;
      swap(V, V_new);           // use <algorithm> std::swap
    }
    cout << " Number of steps = " << steps << endl;

    clock_t t1 = clock();
    cout << " CPU time = " << double(t1 - t0) / CLOCKS_PER_SEC
         << " sec" << endl;
  }

  void set_omega ( double i ) { omega = i; }

  double get_h() const { return h; }
  int    get_L() const { return L; }
  double get_V( int i, int j) const { return V[i][j]; }

protected :
  int L ;                           // number of interior points in x and y
  Matrix<double,2> V,               // potential to be found
    rho,                            // given charge density
    V_new;                          // new potential after each step

  double h;                         // lattice spacing
  int steps;                        // number of iteration steps
  double accuracy;                  // desired accuracy in solution
  double omega;                     // overrelaxation parameter


};

int main() {

  int L;
    cout << " Iterative solution of Poisson's equation\n"
         << " ---------------------------------------------\n";
    cout << " Enter number of interior points in x or y: ";
    cin >> L;

    Poisson p(L);

    double accuracy;
    cout << " Enter desired accuracy in solution: ";
    cin >> accuracy;
    cout << " Enter 0 for Jacobi, 1 for Gauss Seidel, 2 for SOR: ";
    int choice;
    cin >> choice;

    void (Poisson::* ptfptr) (void);
    switch (choice) {
    case 0:
      ptfptr = &Poisson::Jacobi;
      p.iterate(ptfptr, accuracy);

      break;
    case 1:
      ptfptr = &Poisson::Gauss_Seidel;
      p.iterate(ptfptr, accuracy);

      break;
    case 2: default :
      ptfptr = &Poisson::successive_over_relaxation;
      double omega = 2 / (1 + 4 * atan(1.0) / double(L));
      p.set_omega( omega ) ;
      p.iterate(ptfptr, accuracy);

      break;
    }


    // write potential to file
    cout << " Potential in file poisson.data" << endl;
    ofstream date_file("poisson.data");
    for (int i = 0; i < p.get_L() + 2; i++) {
      double x = i * p.get_h();
      for (int j = 0; j < p.get_L() + 2; j++) {
        double y = j * p.get_h();
        char buff[1000];
        sprintf(buff, "%12.6f %12.6f %12.6f", x, y, p.get_V(i,j) );
        date_file << buff << endl;
      }
    }
    date_file.close();
}
```

```
import math
import cmath
import time
import matplotlib
matplotlib.use('TkAgg')
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation


class Wavepacket :
```

```
def __init__(self, N=600, L=100., dt=0.1, periodic=True):

    self.h_bar = 1.0               # Planck's constant / 2pi in natural units
    self.mass = 1.0                # particle mass in natural units

    # The spatial grid
    self.N = N                     # number of interior grid points
    self.L = L                     # system extends from x = 0 to x = L
    self.dx = L / float(N + 1)     # grid spacing
    self.dt = dt                   # time step
    self.x = []                    # vector of grid points
    self.periodic = periodic       # True = periodic, False = Dirichlet boundary

    # The potential V(x)
    self.V_0 = 0.5                 # height of potential barrier
    self.V_width = 10.0            # width of potential barrier
    self.V_center = 0.75 * L       # center of potential barrier
    self.gaussian = True          # True = Gaussian potential, False = step pote


    # Initial wave packet
    self.x_0 = L / 4.0             # location of center
    self.E = 5.0                   # average energy
    self.sigma_0 = L / 10.0        # initial width of wave packet
    self.psi_norm = 1.0            # norm of psi
    self.k_0 = 0.0                 # average wavenumber
    self.velocity = 0.0            # average velocity

    self.t = 0.0                   # time
    self.psi = []                  # complex wavefunction
    self.chi = []                  # wavefunction for simplified Crank-Nicholson
    self.a = []
    self.b = []                    # to represent tridiagonal elements of matrix
    self.c = []
    self.alpha = 0.0
    self.beta = 0.0                # corner elements of matrix Q

    # reset global vectors
    self.psi = [0 + 1j*0 for j in range(N)]
    self.chi = [0 + 1j*0 for j in range(N)]

    # reset time and the lattice
    self.t = 0.0
    self.dx = L / float(self.N + 1)
    self.x = [ float(j * self.dx) for j in range(self.N) ]

    # initialize the packet
    self.k_0 = math.sqrt(2*self.mass*self.E - self.h_bar**2 / 2 / self.sigma_0
    self.velocity = self.k_0 / self.mass
    self.psi_norm = 1 / math.sqrt(self.sigma_0 * math.sqrt(math.pi))
    for j in range(self.N):
        exp_factor = math.exp( - (self.x[j] - self.x_0)**2 / (2 * self.sigma_0
        self.psi[j] = (math.cos(self.k_0 * self.x[j]) + 1j * math.sin(self.k_0
        self.psi[j] *= exp_factor * self.psi_norm

    # elements of tridiagonal matrix Q = (1/2)(1 + i dt H / (2 hbar))
    for j in range(self.N):
        self.a.append( - 1j * self.dt * self.h_bar / (8 * self.mass * self.dx*
        self.b.append( 0.5 + 1j * self.dt / (4 * self.h_bar) *
                      (self.V(self.x[j]) + self.h_bar**2 / (self.mass * self.dx**2)
        self.c.append( - 1j * self.dt * self.h_bar / (8 * self.mass * self.dx*
    self.alpha = self.c[N-1]
    self.beta = self.a[0]




def V(self, x):
    half_width = abs(0.5 * self.V_width)
    if self.gaussian:
        #return self.V_0 * math.exp(-(x - self.V_center)**2 / (2 * self.half_w
        return self.V_0*(x-self.V_center)**(-1)
    else:
        if abs(x - self.V_center) <= half_width:
            return self.V_0
        else:
            return 0.0


def solve_tridiagonal(self, a, b, c, r, u):
    n = len(r)
    gamma = [ 0 + 1j*0 for j in range(n) ]
    beta = b[0]
    u[0] = r[0] / beta
    for j in range(1, n):
        gamma[j] = c[j-1] / beta
        beta = b[j] - a[j] * gamma[j]
        u[j] = (r[j] - a[j] * u[j-1]) / beta
    for j in range(n-2, -1, -1):
        u[j] -= gamma[j+1] * u[j+1]

def solve_tridiagonal_cyclic(self, a, b, c, alpha, beta, r, x):
    n = len(r)
    bb = [0 + 1j*0 for j in range(self.N)]
    u = [0 + 1j*0 for j in range(self.N)]
    z = [0 + 1j*0 for j in range(self.N)]
    gamma = -b[0]
    bb[0] = b[0] - gamma
    bb[n-1] = b[n-1] - alpha * beta / gamma
    for i in range(1, n-1):
        bb[i] = b[i]
    self.solve_tridiagonal(a, bb, c, r, x)
    u[0] = gamma
    u[n-1] = alpha
    for i in range(1, n-1):
        u[i] = 0
    self.solve_tridiagonal(a, bb, c, u, z)
    fact = x[0] + beta * x[n-1] / gamma
    fact /= 1.0 + z[0] + beta * z[n-1] / gamma
    for i in range(n):
        x[i] -= fact * z[i]

#     T = 5.0                   # time to travel length L


class Animator :


def __init__(self, periodic=True,wavepacket=None):
    self.avg_times = []
    self.periodic = periodic
    self.wavepacket = wavepacket
    self.t = 0.
    self.fig, self.ax = plt.subplots()

    self.myline = plt.axvline( x=(self.wavepacket.V_center - 0.5 * self.wavepa
                               color='r'
        )
    self.myline = plt.axvline( x=(self.wavepacket.V_center + 0.5 * self.wavepa
                               color='r'
        )
    self.ax.set_ylim(0,0.5)
    initvals = [ abs(ix) for ix in self.wavepacket.psi]
    self.line, = self.ax.plot(initvals)



def update(self, data) :
    self.line.set_ydata(data)
    return self.line,

def time_step(self):
    while True :
        start_time = time.clock()
        if self.periodic:
            self.wavepacket.solve_tridiagonal_cyclic(self.wavepacket.a, self.w
                                                      self.wavepacket.c, self.w
                                                      self.wavepacket.psi, self.
        else:
            self.wavepacket.solve_tridiagonal(self.wavepacket.a, self.wavepack
                                               self.wavepacket.c, self.wavepack
        for j in range(self.wavepacket.N):
            self.wavepacket.psi[j] = self.wavepacket.chi[j] - self.wavepacket.
        self.t += self.wavepacket.dt;
        end_time = time.clock()
        print 'Tridiagnonal_step_in_' + str(end_time - start_time)
        yield [abs(ix) for ix in self.wavepacket.psi]

def create_widgets(self):
    self.QUIT = Button(self, text="QUIT", command=self.quit)
    self.QUIT.pack(side=BOTTOM)

    self.draw = Canvas(self, width="600", height="400")
    self.draw.pack(side=TOP)

def animate(self) :
    self.ani = animation.FuncAnimation( self.fig,          # Animate our figure
                                        self.update,       # Update function drav
                                        self.time_step,    # "frames" function d
                                        interval=50,       # 50 ms between iterat
                                        blit=False         # don't blit anything
                                        )


wavepacket = Wavepacket(N=128)
animator = Animator(periodic=True,wavepacket=wavepacket)
animator.animate()
plt.show()

import math
import time
import cpt

import matplotlib.pyplot as plt
import numpy as np
class Poisson :
def __init__ (self, L=50):
    self.L = L                    # number of interior points in x and y
    self.omega = 1.88177          # over-relaxation parameter for L = 50
    self.N = L + 2                # interior plus two boundary points
    self.N1=5*L+2
    N=self.N
    N1=self.N1
    self.V = cpt.Matrix(N, N1)     # potential to be found
    self.rho = cpt.Matrix(N, N1)   # given charge density
    self.VNew = cpt.Matrix(N, N1) # new potential after each step
    self.h = 2.0 / (L + 1)        # lattice spacing assuming size in x and y =
    self.q = 10.0                 # point charge
    i = N / 2                     # center of lattice
    j = int(N1/6)
    self.rho[i][j] = self.q / self.h**2    # charge density
    self.rho[i][2*j] = self.q / self.h**2
    self.rho[i][3*j] = self.q / self.h**2
    self.rho[i][4*j] = self.q / self.h**2
    self.rho[i][5*j] = self.q / self.h**2

def Jacobi(self) :                # Jacobi algorithm for a single iterative ste
    VNew = self.VNew
    V    = self.V           #avoid lots of typing
    rho  = self.rho
    h    = self.h
    for i in range(1, self.L+1):
        for j in range(1, 5*self.L+1):
            VNew[i][j] = 0.25 * (V[i-1][j] + V[i+1][j] +
                                 V[i][j-1] + V[i][j+1] +
                                 h**2 * rho[i][j])

def GaussSeidel(self):           # Gauss-Seidel algorithm for one iterative ste
    L = self.L
    VNew = self.VNew
    V    = self.V           #avoid lots of typing
    rho  = self.rho
    h    = self.h

    # copy V to VNew
    for i in range(1, self.L+1):
        for j in range(1, 5*self.L+1):
            VNew[i][j] =  V[i][j]

    # perform Gauss-Seidel update
    for i in range(1, self.L):
        for j in range(1, 5*self.L+1):
            VNew[i][j] = 0.25 * (VNew[i-1][j] + VNew[i+1][j] +
                                 VNew[i][j-1] + VNew[i][j+1] +
                                 h**2 * rho[i][j])


def SuccessiveOverRelaxation(self):
    L = self.L
    VNew = self.VNew
    V    = self.V           #avoid lots of typing
    rho   = self.rho
    h     = self.h
    omega= self.omega

    # update even sites in red-black scheme
    for i in range(1, self.L+1):
        for j in range(1, 5*self.L+1):
            if (i + j) % 2 == 0:
                VNew[i][j] = (1 - omega) * V[i][j] + omega / 4 * (
                             V[i-1][j] + V[i+1][j] + V[i][j-1] +
                             V[i][j+1] + h**2 * rho[i][j] )

    # update odd sites in red-black scheme
    for i in range(1, self.L+1):
        for j in range(1, 5*self.L+1):
            if (i + j) % 2 != 0:
                VNew[i][j] = (1 - omega) * V[i][j] + omega / 4 * (
                             VNew[i-1][j] + VNew[i+1][j] + VNew[i][j-1] +
                             VNew[i][j+1] + h**2 * rho[i][j] )

def relativeError(self):
    L = self.L
    VNew = self.VNew
    V    = self.V           #avoid lots of typing
    rho  = self.rho
    h    = self.h
    omega= self.omega

    error = 0
    n = 0
    for i in range(1, self.L+1):
        for j in range(1, 5*self.L+1):
            if VNew[i][j] != 0 and VNew[i][j] != V[i][j]:
                error += abs(1 - V[i][j] / VNew[i][j])
                n += 1
    if n != 0:
        error /= n
    return error




print " Iterative solution of Poisson's equation"
print " --------------------------------------"
L = int(input(" Enter number of interior points in x or y: "))
poisson = Poisson(L=L)
accuracy = float(input(" Enter desired accuracy in solution: "))
```

```
choice = int( input(" Enter choice of algorithm , Jacobi (0) , Gauss-Seidel (1) or S

start_time = time.clock()

steps = 0

while True:
    if choice == 0 :
        poisson.Jacobi()
    elif choice == 1 :
        poisson.GaussSeidel()
    else :
        poisson.SuccessiveOverRelaxation()
    if poisson.relativeError() < accuracy:
        break
    for i in range(1, poisson.L+1):
        for j in range(1, 5*poisson.L+1):
            poisson.V[i][j] = poisson.VNew[i][j]
    steps += 1

print " Number of steps =", steps
print " CPU time =", time.clock() - start_time, "sec"
fp = open('p3.data','w')
for i in range(0, poisson.L+2):
    for j in range(0, 5*poisson.L+2):
        s = '{}  {} {} {}  {} \n'.format(i,j,i*poisson.h,j*poisson.h,poisson.V[i][j
        fp.write(s)
fp.write('{}'.format(poisson.h))
fp.close


#continue to generate the trajectory, taking gradient as acceleration

vx=0.2
vy=0.2
x=[]
y=[]
xp=0.0
yp=5.0
x.append(xp)
y.append(yp)
dt=0.05
ix = int(xp/poisson.h)
iy = int(yp/poisson.h)
#first step using two points method
ax = -(poisson.V[ix+1][iy]-poisson.V[ix][iy])/poisson.h
ay = -(poisson.V[ix][iy+1]-poisson.V[ix][iy])/poisson.h
xp=xp+vx*dt+0.5*ax*dt**2
```

```
yp=yp+vy*dt+0.5*ay*dt**2
vx=vx+ax*dt
vy=vy+ay*dt

x.append(xp)
y.append(yp)
dt=0.05
ix = int(xp/poisson.h)
iy = int(yp/poisson.h)
#first step using two points method
ax = -(poisson.V[ix+1][iy]-poisson.V[ix][iy])/poisson.h
ay = -(poisson.V[ix][iy+1]-poisson.V[ix][iy])/poisson.h
xp=xp+vx*dt+0.5*ax*dt**2
yp=yp+vy*dt+0.5*ay*dt**2
vx=vx+ax*dt
vy=vy+ay*dt

x.append(xp)
y.append(yp)
dt=0.05
ix = int(xp/poisson.h)
iy = int(yp/poisson.h)
#first step using two points method
ax = -(poisson.V[ix+1][iy]-poisson.V[ix][iy])/poisson.h
ay = -(poisson.V[ix][iy+1]-poisson.V[ix][iy])/poisson.h
xp=xp+vx*dt+0.5*ax*dt**2
yp=yp+vy*dt+0.5*ay*dt**2
vx=vx+ax*dt
vy=vy+ay*dt


det=True

while True:
    x.append(xp)
    y.append(yp)
    ix = int(xp/poisson.h)
    iy = int(yp/poisson.h)
    if ix > 99 or ix < 2 or iy > 499 or iy < 2:
        break
    ax = -(poisson.V[ix-2][iy]-8*poisson.V[ix-1][iy]+8*poisson.V[ix+1][iy]-poisson.
    ay = -(poisson.V[ix][iy-2]-8*poisson.V[ix][iy-1]+8*poisson.V[ix][iy+1]-poisson.
    xp=xp+vx*dt+0.5*ax*dt**2
    yp=yp+vy*dt+0.5*ay*dt**2
    vx=vx+ax*dt
    vy=vy+ay*dt

plt.plot(x,y)
```

```
# Convert x, y, V(x,y) to a surface plot
#from mpl_toolkits.mplot3d import Axes3D
#from matplotlib import cm
#from matplotlib.ticker import LinearLocator, FormatStrFormatter
#from matplotlib.mlab import griddata

# Define the axes
#x = np.arange(0, poisson.h*(5*poisson.L+2), poisson.h)
#y = np.arange(0, poisson.h*(poisson.L+2), poisson.h)
# Get the grid
#X, Y = np.meshgrid(x, y)
# Set Z to the poisson V[i][j]
#Z = np.array( poisson.V )

#fig = plt.figure()
#ax = fig.gca(projection='3d')
#scat = ax.plot_surface( X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
#                         linewidth=0, antialiased=False )

plt.show()
```