Homework 7

Varun Nair

April 5, 2019

```
In [21]: %precision %g
         %matplotlib inline
         %config InlineBackend.figure_format = 'retina'

         from math import sqrt, pi, sin, cos, floor, exp
         from cmath import exp as cexp
         import numpy as np
         from numpy import linalg as LA
         from scipy import constants as con
         import matplotlib.pyplot as plt
         from dcst import dst,idst
```

1 CP 9.2

The Jacobi method that the book described is quite slow. By their measure, it took about 10 minutes to calculate the voltages at a 100x100 grid with fixed voltage edges. It offers a different solution that speeds up the calculation in two ways: by "overrelaxing" and by continuously updating the values. This is known as the Gauss-Seidel method. Using the Laplace wave equation

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0,$$

we can solve for the values of the electric potential on the interior of a square region with side length L given the boundary conditions

$$\phi(x,L) = V \qquad \text{and} \qquad \phi(x,0) = \phi(0,y) = \phi(L,y) = 0.$$

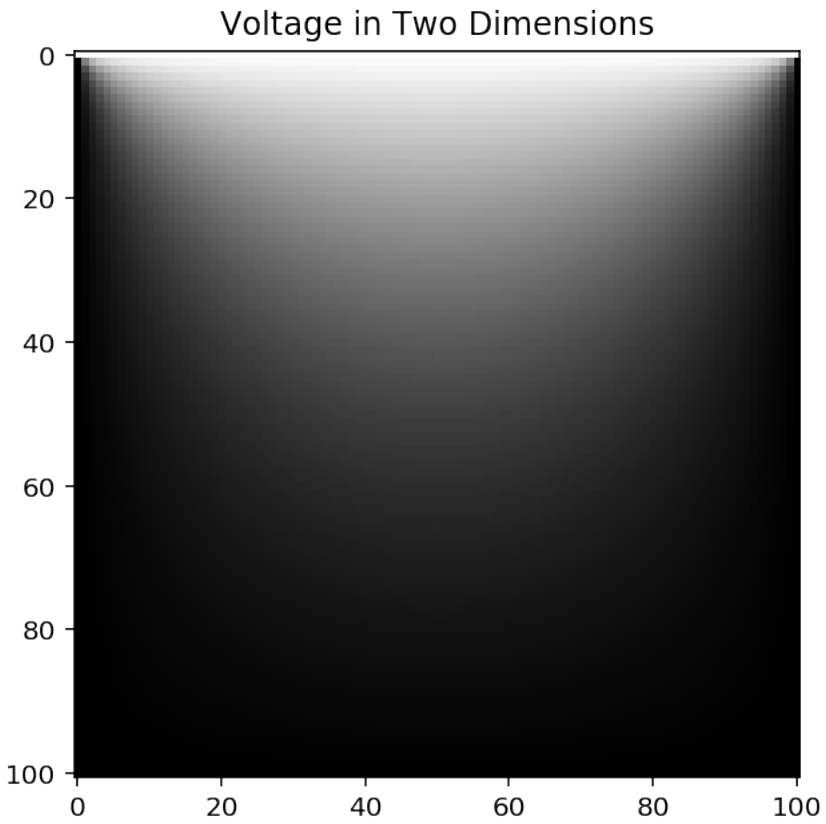
The Gauss-Seidel method iterates through and solves

$$\phi(x,y) \leftarrow \frac{1+\omega}{4}\left[\phi(x+a,y) + \phi(x-a,y) + \phi(x,y+a) + \phi(x,y-a)\right] - \omega\phi(x,y) \tag{1}$$

to within a target accuracy δ.

```
In [22]: %%time
```

```
        M = 100         #grid squares on a side
        V = 1.0         #voltage at top wall
        target = 1e-6   #target accuracy
        w = 0.94 #overrelaxation constant

        #create arrays to hold potential values
        phi = np.zeros([M+1,M+1],float)
        phi[0,:] = V
        d1 = np.empty([M+1,M+1],float)

        d = 1.0
        while d>target:

            #calculate new values of the potential
            for i in range(M+1):
                for j in range(M+1):
                    #because no phiprime, calculates differences
                    d1[i,j] = phi[i,j]

                    if i==0 or i==M or j==0 or j==M:
                        phi[i,j] = phi[i,j]
                    else:
                        phi[i,j] = (1+w)*(phi[i+1,j] + phi[i-1,j] \
                                        + phi[i,j+1] + phi[i,j-1])/4\
                                - w*phi[i,j]
                    d1[i,j] -= phi[i,j]

            # Calculate maximum difference from old values
            d = np.max(abs(d1))

        #makes density plot
        fig1, ax1 = plt.subplots(1, 1, figsize = (5, 5))

        ax1.imshow(phi, cmap='gray')
        ax1.set_title("Voltage in Two Dimensions")

CPU times: user 6.44 s, sys: 73.8 ms, total: 6.51 s
Wall time: 6.76 s
```



After ω is increased beyond 0.94, the time it takes for the program to run again begins to increase. However as promised, the 5.9s it took to solve and plot the original partial differential equation is far less than the original 10 minutes with the Jacobi method.

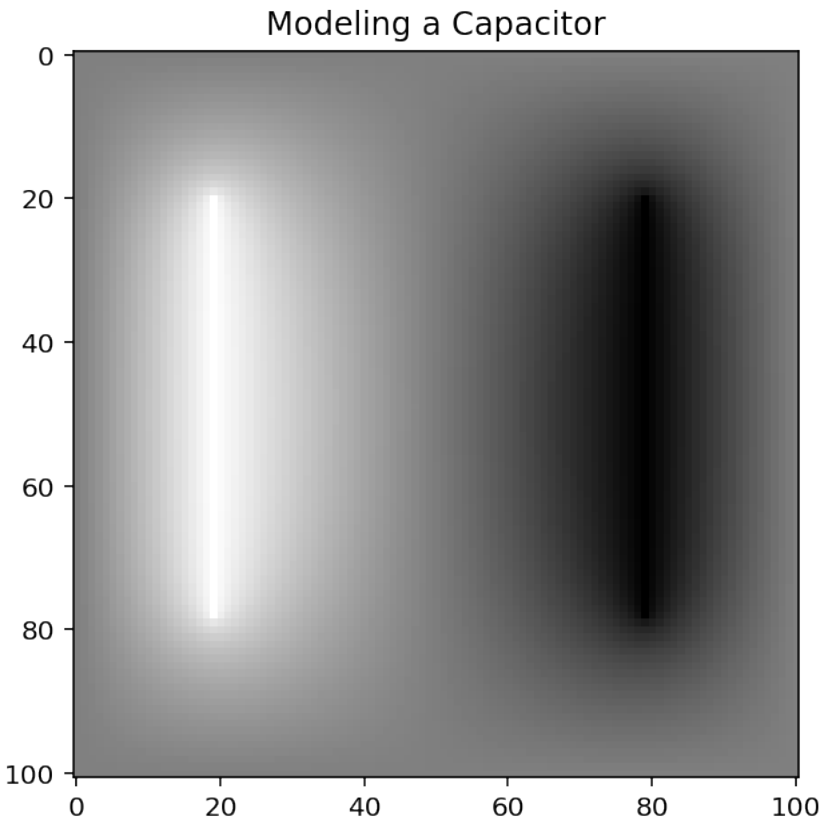
2 CP 9.3

We can solve this exercise with the same method as above, simply adjusting boundary conditions. The updated boundary conditions (in SI Units) are

$$\phi(x,0.1) = \phi(x,0) = \phi(0,y) = \phi(0.1,y) = 0, \qquad \phi(0.02, 0.02 < y < 0.08) = 1, \text{ and} \qquad \phi(0.08, 0.02 < y < 0.08) =$$

```
In [23]: %%time

         M = 100         #grid squares on a side
         V = 1.0         #voltage at top wall
         target = 1e-6   #target accuracy
         w = 0.94 #overrelaxation constant

         #create arrays to hold potential values
         phi = np.zeros([M+1,M+1],float)
```

```
        phi[19:79,19] = V
        phi[19:79,79] = -V
        d1 = np.empty([M+1,M+1],float)

        d = 1.0
        while d>target:

            #calculate new values of the potential
            for i in range(M+1):
                for j in range(M+1):
                    #because no phiprime, calculates differences
                    d1[i,j] = phi[i,j]

                    if i==0 or i==M or j==0 or j==M:
                        phi[i,j] = phi[i,j]
                    elif j==19 and i > 19 and i < 79:
                        phi[i,j] = phi[i,j]
                    elif j==79 and i > 19 and i < 79:
                        phi[i,j] = phi[i,j]
                    else:
                        phi[i,j] = (1+w)*(phi[i+1,j] + phi[i-1,j] \
                                          + phi[i,j+1] + phi[i,j-1])/4\
                                   - w*phi[i,j]
                    d1[i,j] -= phi[i,j]

            # Calculate maximum difference from old values
            d = np.max(abs(d1))

        #makes density plot
        fig2, ax2 = plt.subplots(1, 1, figsize = (5,5))

        ax2.imshow(phi, cmap='gray')
        ax2.set_title("Modeling a Capacitor")

CPU times: user 8.46 s, sys: 89.9 ms, total: 8.55 s
Wall time: 9.56 s
```



3 CP 9.4 Thermal diffusion in the Earth's crust

This problem has us solving a PDE initial value problem as opposed to a boundary value problem, using the FTCS method. Specifically one where the initial boundary condition is not constant. This problem looks at the heat that diffuses into the Earth's crust as temperature varies with the seasons according to

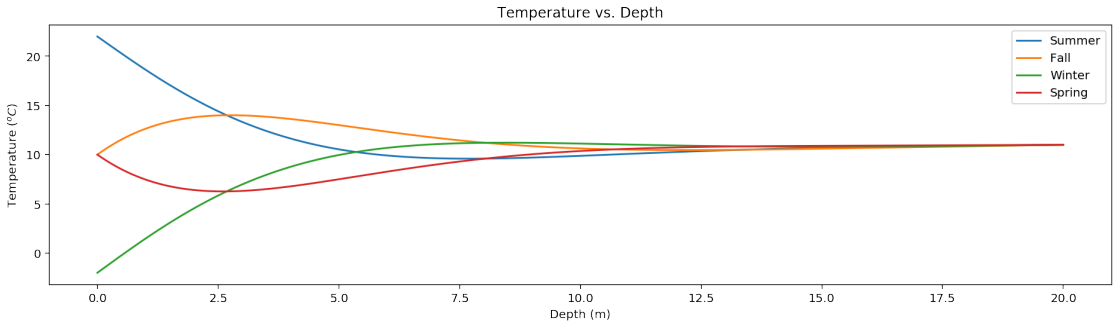
$$T_0(t) = A + B \sin \frac{2\pi t}{\tau}$$

for $\tau = 365$ days, $A = 10^oC$, and $B = 12^oC$. This model assumes that temperature 20m below the Earth's surface is a constant 11^oC and that the thermal diffusivity constant $D = 0.1$ m^{2}day^{-1}. Initially, the temperature everywhere between the surface and a depth of 20 m is at temperature 10^oC.

```
In [24]: %%time
         # Constants
         L = 20        # Thickness of crust in meters
         D = 0.1       # Thermal diffusivity
         N = 1000      # Number of divisions in grid
         depth = np.linspace(0,L,N+1)
```

```
a = L/N       # Grid spacing
h = 1e-3      # Time-step
epsilon = h/1000

#constant lower boundary temp in Celcius
Tlo = 11.0
#starting intermediate temps
Tmid = 10.0
#changing boundary temperature
    #constants
A = 10
B = 12
tau = 365
def T0(t):
    return A + B*sin(2*pi*t/tau)

t1 = 9.25*tau
t2 = 9.50*tau
t3 = 9.75*tau
t4 = 10.0*tau
tend = t4 + epsilon

# Create arrays
T = np.empty(N+1,float)
T[N] = Tlo
T[1:N] = Tmid
Tp = np.empty(N+1,float)
Tp[0] = T0(0)
Tp[N] = Tlo

fig3, ax3 = plt.subplots(1, 1, figsize = (16, 4))

t = 0.0
c = h*D/(a**2)
while t<tend:

    T[0] = T0(t)

    # Calculate the new values of T
    Tp[1:N] = T[1:N] + c*(T[0:N-1] + T[2:N+1] - 2*T[1:N])

    T,Tp = Tp,T
    t += h

    # Make plots at the given times
    if abs(t-t1) < epsilon:
        ax3.plot(depth,T,label='Summer')
    if abs(t-t2) < epsilon:
```

```
            ax3.plot(depth,T,label='Fall')
        if abs(t-t3) < epsilon:
            ax3.plot(depth,T,label='Winter')
        if abs(t-t4) < epsilon:
            ax3.plot(depth,T,label='Spring')

    ax3.set_xlabel("Depth (m)")
    ax3.set_ylabel("Temperature ($^oC$)")
    ax3.set_title("Temperature vs. Depth")
    ax3.legend()
    plt.show()
```




```
CPU times: user 35.7 s, sys: 104 ms, total: 35.8 s
Wall time: 36.2 s
```

We can see that the temperature near the surface of the Earth's crust is the most sensitive to seasonal temperature swings. As the depth approaches 2 m, the temperature of the crust is unaffected at this level of detail. Also the seasons in the key were labeled assuming this is in the Northern Hemisphere.

4 CP 9.5 FTCS solution of the wave equation

This problem looks at the behavior of a string after it's plucked and how the wave propagates along it, holding both ends fixed. Despite knowing that the FTCS method produces unstable solutions for anything longer than *short* time intervals, we'll solve this system using the method for the times $t = 2$ ms, 50 ms, and 100 ms.

To model this system, we start with the wave equation

$$\frac{\partial^2 \phi}{\partial t^2} = v^2 \frac{\partial^2 \phi}{\partial x^2}.$$

Then we split this into 2 first order differential equations for each grid point, so we can write the two relations

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = \psi(x,t) \qquad \frac{\mathrm{d}\psi}{\mathrm{d}t} = \frac{v^2}{a^2}\big[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)\big].$$

Then we can write the program to solve the equations

$$\phi(x,t+h) = \phi(x,t) + h\psi(x,t) \tag{2}$$

$$\psi(x,t+h) = \psi(x,t) + h\frac{v^2}{a^2}\left[\phi(x+a,t) + \phi(x-a,t) - 2\phi(x,t)\right].$$

Now for this specific problem, we define

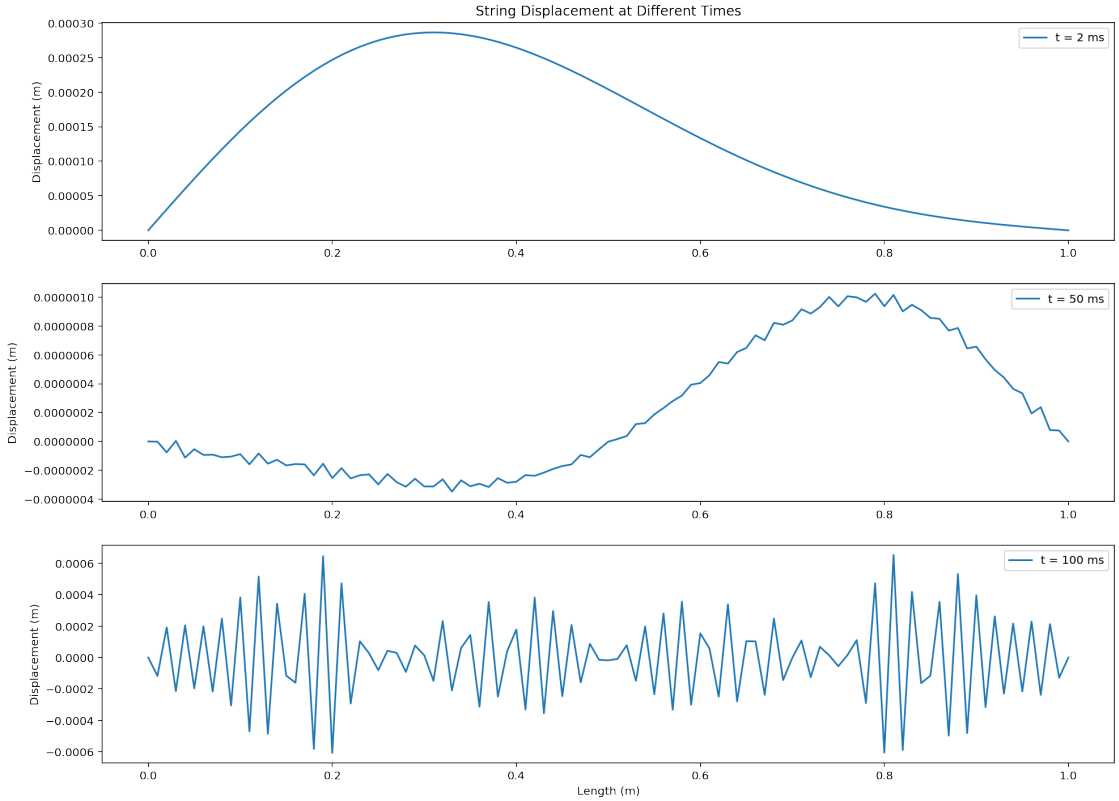
$$\psi(x) = C\frac{x(L-x)}{L^2}\exp\left[-\frac{(x-d)^2}{2\sigma^2}\right].$$

The initial condition for this problem is

$$\phi(x,0) = 0.$$

```
In [25]: %%time
         # Constants
         L = 1          #length of string (m)
         d = 0.1        #point of contact (m)
         v = 100        #wave speed (ms^-1)
         N = 100        # Number of divisions in grid
         length = np.linspace(0,L,N+1)
         a = L/N        # Grid spacing
         h = 1e-6       # Time-step

         epsilon = h/1000

         C = 1 #(ms^-1)
         s = 0.3 #sigma in the exercise (m)

         def wave(x):
             p1 = C*(x*(L-x)) / L**2
             p2 = exp(-(x-d)**2 / (2*s**2))
             return p1*p2

         t1 = 0.002
         t2 = 0.050
         t3 = 0.100
         tend = t3 + epsilon

         # Create arrays
         phi = np.empty(N+1,float)
         phi[:] = 0.0
         phip = np.empty(N+1,float)
         phip[:] = 0.0

         psi = np.empty(N+1,float)
         psi[:] = 0.0
```

```
psip = np.empty(N+1,float)
psip[:] = 0.0

fig4, ax4 = plt.subplots(3, 1, figsize = (16, 12))

#setting wave velocity
for i in range(len(psi)):
    psi[i] = wave(i*L/100)

t = 0.0
c = (h*v**2)/(a*a)
while t<tend:

    #hold ends fixed
    phi[0] = 0.0
    phi[N] = 0.0

    #solving differential equations
    phip[1:N] = phi[1:N] + h*psi[1:N]

    psip[1:N] = psi[1:N]\
              + c*(phi[0:N-1] + phi[2:N+1] - 2*phi[1:N])

    phi,phip = phip,phi
    psi,psip = psip,psi
    t += h

    #plots displacement
    if abs(t-t1) < epsilon:
        ax4[0].plot(length,phi,label="t = 2 ms")
    if abs(t-t2) < epsilon:
        ax4[1].plot(length,phi,label="t = 50 ms")
    if abs(t-t3) < epsilon:
        ax4[2].plot(length,phi,label="t = 100 ms")

ax4[2].set_xlabel("Length (m)")
ax4[0].set_ylabel("Displacement (m)")
ax4[1].set_ylabel("Displacement (m)")
ax4[2].set_ylabel("Displacement (m)")
ax4[0].set_title("String Displacement at Different Times")
ax4[0].legend()
ax4[1].legend()
ax4[2].legend()
plt.show()
```




```
CPU times: user 1.91 s, sys: 46.2 ms, total: 1.95 s
Wall time: 1.94 s
```

5 CP 9.9

Here we attempt to solve the time-dependent Schrodinger equation

$$-\frac{\hbar^2}{2M}\frac{\partial^2\psi}{\partial x^2} = \mathrm{i}\hbar\frac{\partial\psi}{\partial t}$$

for a particle in an infinite well of width L. One known unnormalized solution is

$$\psi_k(x,t) = \sin\left(\frac{\pi k x}{L}\right)\mathrm{e}^{\mathrm{i}Et/\hbar}, \qquad \text{where} \qquad E = \frac{\pi^2\hbar^2k^2}{2ML^2}.$$

We can take a linear combination of these to express the wavefunction as

$$\psi(x_n,t) = \frac{1}{N}\sum_{k=1}^{N-1} b_k \sin\left(\frac{\pi k n}{N}\right)\exp\left(\mathrm{i}\frac{\pi^2\hbar k^2}{2ML^2}t\right). \tag{3}$$

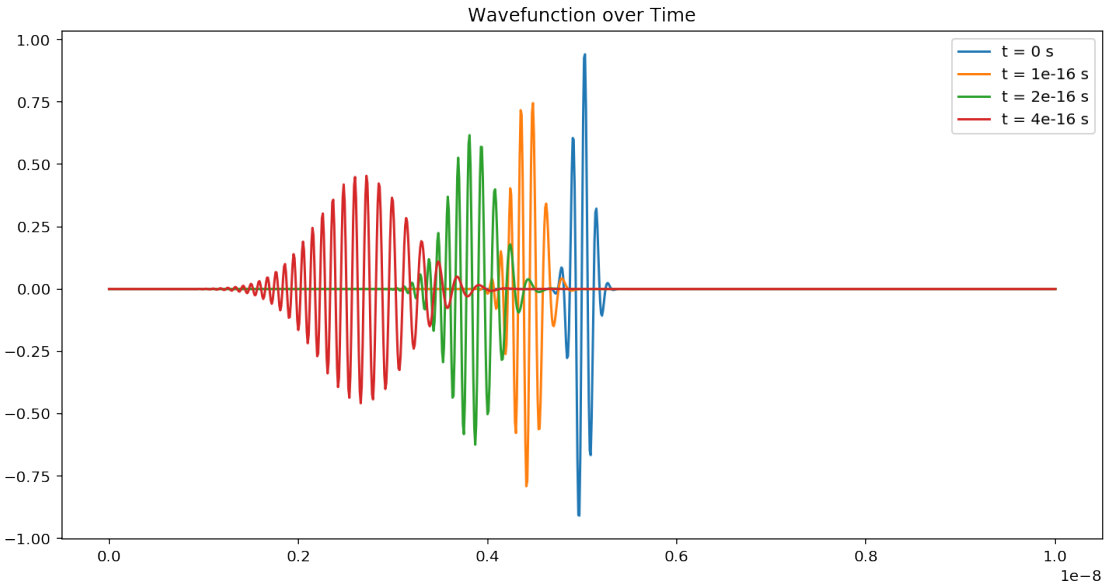
for all $x_n = \frac{nL}{N}$ along the length of the the well. Furthermore, the coefficients b_k are complex and can be expanded to separate the real and imaginary components such that $b_k = \alpha_k + \mathrm{i}\eta_k$. These

arrays of values (α_k and η_k) can be calculated using discrete sine transforms. Once these arrays are found, the real portion of the wavefunction (the part we're interested in) can be found by the relation

$$\text{Re}\,\psi(x_n,t) = \frac{1}{N}\sum_{k=1}^{N-1}\left[\alpha_k \cos\left(\frac{\pi^2\hbar k^2}{2ML^2}t\right) - \eta_k \sin\left(\frac{\pi^2\hbar k^2}{2ML^2}t\right)\right]\sin\left(\frac{\pi kn}{N}\right). \tag{4}$$

In [26]:
```
# Constants
L = 1e-8       #width of box (m)
M = con.m_e    #electron mass (m)
x0 = L/2       #wave speed (ms^-1)
N = 1000       #number of divisions in grid
kappa = 5e10   #(m^-1)
s = 1e-10      #sigma (m)
hbar = con.hbar  # Time-step
x = np.linspace(0,L,N+1)

def E(k):
    """Returns energy of particle based on k"""
    num = pi**2 * hbar**2 * k**2
    den = 2 * M * L**2
    return num / den

#initial condition wavefunction
def wave0(x):
    return np.exp(-(x-x0)**2/(2*s**2) + 1j*kappa*x)

psi0 = wave0(x)

alpha = dst(np.real(psi0))
eta = dst(np.imag(psi0))

t0 = 0.0
t1 = 1e-16
t2 = 2e-16
t3 = 4e-16

def psi(t):
    """Finds the real component of the wavefunction
    according to the above equation at time t"""
    y = np.zeros_like(x)
    for k in range(N+1):
        y[k] = alpha[k] * cos(E(k)*t/hbar)\
             - eta[k]   * sin(E(k)*t/hbar)

    return idst(y) #/ N

fig5, ax5 = plt.subplots(1, 1, figsize = (12, 6))
```

```
psi0 = psi(t0)
psi1 = psi(t1)
psi2 = psi(t2)
psi3 = psi(t3)
psi4 = psi(t4)

ax5.plot(x, psi0,label='t = {:1.0f} s'.format(t0))
ax5.plot(x, psi1,label='t = {:4.0e} s'.format(t1))
ax5.plot(x, psi2,label='t = {:4.0e} s'.format(t2))
ax5.plot(x, psi3,label='t = {:4.0e} s'.format(t3))
ax5.set_title("Wavefunction over Time")
ax5.legend()

plt.show()
```



From the graphs, we can see that over time the wavefunction disperses and initially travels leftward. It will do this until it hits the boundary of the infinite well at which point, its uncertainty will reduce and then it will reverse direction and travel rightwards.

In [ ]: