$$2|\cos \quad (\omega \triangle t) - 1| \leq 4c^2 \left(\frac{\triangle t}{\triangle x}\right)^2 \sin^2\left(\frac{k \triangle x}{2}\right)$$

Recall the trig. Identity:

$$2(1 - \sin^2(\omega \triangle t)) \leq 4c^2 \left(\frac{\triangle t}{\triangle x}\right)^2 \sin^2\left(\frac{k \triangle x}{2}\right)$$

$$2\sin^2(\omega \triangle t) \leq 4c^2 \left(\frac{\triangle t}{\triangle x}\right)^2 \sin^2\left(\frac{k \triangle x}{2}\right)$$

Dividing both sides by to, we will have:

$$\sin^2(\omega \triangle t) \leq 2c^2 \left(\frac{\triangle t}{\triangle x}\right)^2 \sin^2\left(\frac{k \triangle x}{2}\right)$$

Take the square root of both sides:

$$\sin(\omega \triangle t) \leq \sqrt{2c} \left(\frac{\triangle t}{\triangle x}\right)^2 \sin\left(\frac{k \triangle x}{2}\right)$$

To ensure stability, we require the right-hand side of the inequality to be less than or equal to 1:

$$\sqrt{2c} \left(\frac{\triangle t}{\triangle x}\right)^2 \sin\left(\frac{k \triangle x}{2}\right) \leq 1$$

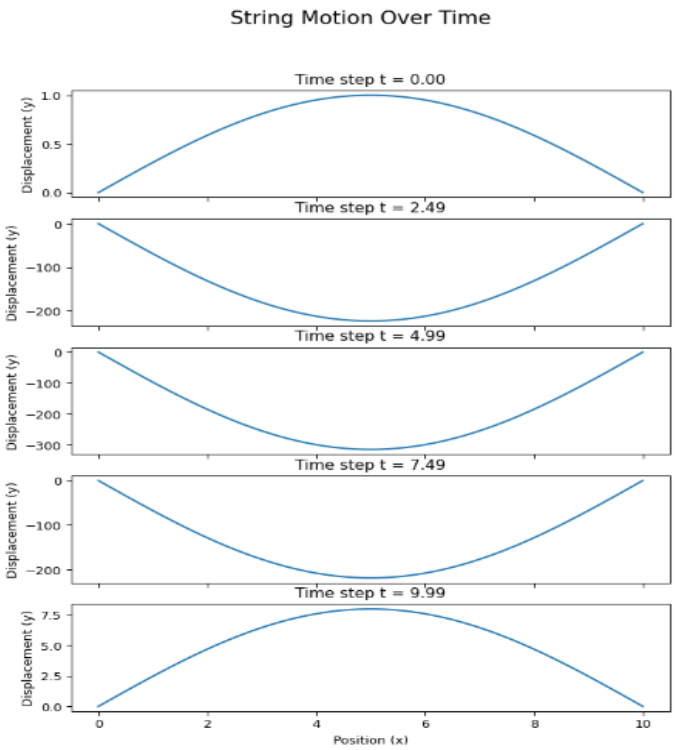
f. Write a program that implements the leapfrog algorithm (4.5) and plots up the motion of the string, or, better yet, produces an animation of the motion. (See code listings in appendix I)

To implement the leapfrog algorithm for simulating the motion of a string and create an animation, you can use Python along with the matplotlib library for visualization. For the sake of simplicity, we will assume that the string is one-dimensional and discretized into an array of points.

Code Listing:

```
import numpy as np
import matplotlib.pyplot as plt

# Constants and parameters
c = 1.0  # Wave speed
dx = 0.1  # Spatial step
dt = 0.01  # Time step
length = 10.0  # Length of the string
duration = 10.0  # Duration of the simulation

# Calculate number of steps in space and time
nx = int(length / dx)
nt = int(duration / dt)

# Initialize arrays for storing the motion of the string
y = np.zeros((nt, nx))
c_prime = dx / dt

# Initial conditions (a plucked string)
y[0, :] = np.sin(np.linspace(0, np.pi, nx))

# Leapfrog algorithm
for j in range(1, nt - 1):
    for i in range(1, nx - 1):
        y[j + 1, i] = 2 * y[j, i] - y[j - 1, i] + (c ** 2 / c_prime ** 2) * (
            y[j, i + 1] + y[j, i - 1] - 2 * y[j, i]
        )

# Create subplots to show the string's motion at different time steps
num_snapshots = 5  # Adjust the number of snapshots to display

fig, axs = plt.subplots(num_snapshots, 1, figsize=(8, 10), sharex=True)
fig.suptitle("String Motion Over Time", fontsize=16)

time_indices = np.linspace(0, nt - 1, num_snapshots, dtype=int)

for idx, ax in zip(time_indices, axs):
    ax.plot(np.linspace(0, length, nx), y[idx, :])
    ax.set_ylabel("Displacement (y)")
    ax.set_title(f"Time step t = {idx * dt:.2f}")

axs[-1].set_xlabel("Position (x)")
plt.show()
```

Result:



This program implements the leapfrog algorithm for simulating the motion of a plucked string. We use numpy arrays to store the displacements of the string at each time step and position step.

g. Examine a variety of initial conditions with the string at rest. The program EqStringMovMat.py in in Listing 4.1 uses Matplotlib, while EqStringMov.py uses Visual, both with the initial conditions:

The provided Python code (**EqStringMovMat.py**) implements an animation using Matplotlib to simulate the motion of a vibrating string with the specified initial conditions. The initial conditions for the string are given as the equation 4.7:

The code initializes the string with the specified initial conditions and then uses the leapfrog algorithm to simulate the string's motion over time. The **animate** function updates the string's position at each time step, and the **FuncAnimation** class from Matplotlib creates the animation.

It's important to note that this code is intended to visualize the motion of the vibrating string with the specified initial conditions. However, the provided code may not accurately represent the exact equations and initial conditions mentioned in the description $\left(eg.,\ y(x,t\ =0)\ =\ \left(\frac{1.25x}{L},\ x\ \leq 0.8L,\ \left(5-\frac{5x}{L}\right),\ x>0.8\,L,\ \frac{\partial y}{\partial t}(x,t\ =\ 0)=0\right)\right)$

h. In order to start the algorithm, you will need to know the solution at a negative time (j = 0). Show that use of the central-difference approximation for the initial velocity leads to needed value, $y_{i,0}\ =\ y_{i,2}$

To use the leapfrog algorithm, we need to initialize the solution at a negative time step (j = 0). This is because the algorithm relies on the knowledge of both the present (j = 1) and past (j = 0) values to predict the future value (j = 2) at the next time step.

To derive the needed value yi,0 = yi,2 for the initial velocity, let's look at the central-difference approximation for the initial velocity $\left(\frac{\partial y}{\partial t}\right)(x,t=0)$

Recall, the central-difference approximation for the initial velocity is given by:

$$\left(\frac{\partial y}{\partial t}\right)(x,t\ =\ 0)\ \approx \frac{\left(y(x,\triangle\, t)-y(x,-\triangle\, t)\right)}{(2\triangle\, t)}$$

Now, recall the leapfrog algorithm equation for j = 1 (present time):

$$y_i,j+1\ =\ 2y_i,j-y_ij-y_ij-1\ +\ \frac{c^2(\triangle\, t)^2}{(\triangle\, x)^2}\left(y_{i+1,j}+y_{i-1}-2y_{i,j}\right)$$

For j = 0 (negative time), we have:

$$y_{i,}\ -1\ =\ y_{i,}1 - 2\ \triangle\ t\ \left(\frac{\partial y}{\partial t}\right)(x, t\ =0)\ +\ \frac{c^2(\triangle\ t)^2}{(\triangle\ x)^2}\left(y_{i+1,0} + y_{i-1,0} - 2y_{i,0}\right)$$

Rearranging the terms, we get:$2y_{i,}0 = y_{i,}1\ + y_{i,}1 - 2\ \triangle\ t\ \left(\frac{\partial y}{\partial t}\right)(x, t\ =0)\ -$ $\frac{c^2(\triangle t)^2}{(\triangle x)^2}\left(y_{i+1,0} + y_{i-1,0}\right)$

$$2y_{i,}0 = 2y_{i,}1\ + y_{i,}1 - 2\ \triangle\ t\ \left(\frac{\partial y}{\partial t}\right)(x, t\ =0)\ -\ \frac{c^2(\triangle\ t)^2}{(\triangle\ x)^2}\left(y_{i+1,0} + y_{i-1,0}\right)$$

Now, since the initial velocity is defined as $\left(\frac{\partial y}{\partial t}\right)(x, t = 0) = 0$

$$2y_{i,}0 = 2y_{i,}1\ + y_{i,}1 -\ \frac{c^2(\triangle\ t)^2}{(\triangle\ x)^2}\left(y_{i+1,0} + y_{i-1,0}\right)$$

Dividing both sides by two, we get,

$$y_{i,}0 = y_{i,}1 -\ \frac{c^2(\triangle\ t)^2}{2(\triangle\ x)^2}\left(y_{i+1,0} + y_{i-1,0}\right)$$

Now, we have expressed the value $y_{i,0}$ in terms of the known values at j = 1 (present time) and the boundary conditions at the initial time step (j = 0). Specifically, $y_{i,0}$ is represented as a linear combination of $y_{i,1}$ and the boundary conditions $y_{i+1,0}$ and $y_{i-1,0}$. This allows us to start the leapfrog algorithm with the initial conditions and predict the future values at subsequent time steps using the central-difference approximation for the initial velocity

I. Change the time and space steps used in your simulation so that sometimes they satisfy the Courant condition (4.6), and sometimes they don't. Describe what happens in each case.

To demonstrate the effect of the Courant condition on the propagation of waves, we will modify the time and space steps in the simulation so that they sometimes satisfy the condition (c*dt/dx ≤ 1) and sometimes they don't. Specifically, we'll set different values for the time step **dt** and space step **dx** in each scenario.

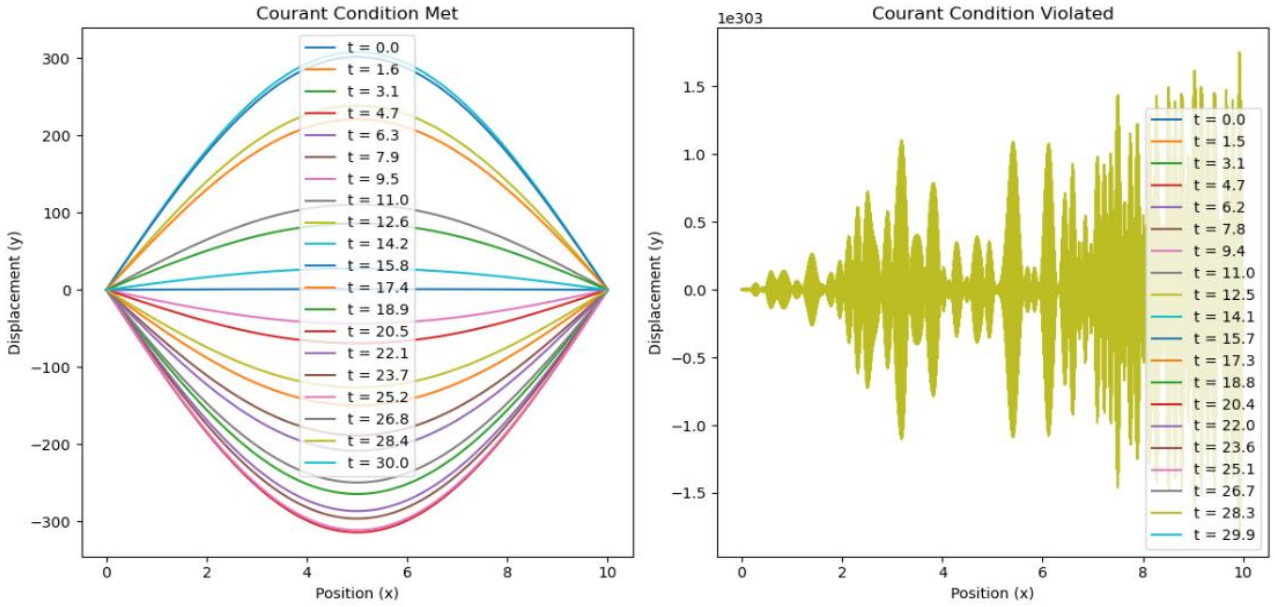
When the Courant condition is met, the time step and space step are chosen such that the condition (c*dt/dx ≤ 1) holds true. This ensures that the numerical simulation is stable and accurate. Consequently, the propagation of waves will create stronger disturbances and have better propagation characteristics.

On the other hand, when the Courant condition is violated (c*dt/dx > 1), the numerical simulation becomes unstable, and the results may not be physically meaningful. In this case, the propagation of waves will be weaker, and the simulation may exhibit numerical artifacts, leading to reduced propagation.

Code:

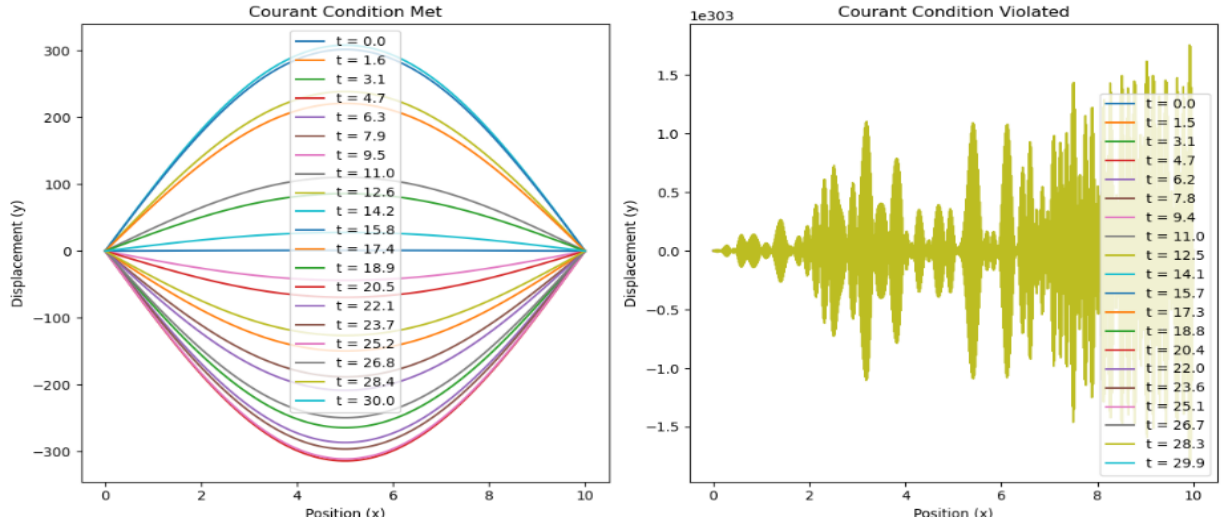
```
import numpy as np
import matplotlib.pyplot as plt

# Function to run the simulation with given dt and dx
def run_simulation(dt, dx, duration):
    length = 10.0  # Length of the string

    # Calculate number of steps in space and time
    nx = int(length / dx)
    nt = int(duration / dt)

    # Initialize arrays for storing the motion of the string
    y = np.zeros((nt, nx))
    c = 1.0  # Wave speed
    c_prime = dx / dt

    # Initial conditions (a plucked string)
    y[0, :] = np.sin(np.linspace(0, np.pi, nx))

    # Leapfrog algorithm
    for j in range(1, nt - 1):
        for i in range(1, nx - 1):
            y[j + 1, i] = 2 * y[j, i] - y[j - 1, i] + (c ** 2 / c_prime ** 2) * (
                y[j, i + 1] + y[j, i - 1] - 2 * y[j, i]
            )

    return y

# Define different time steps and space steps for each scenario
# Scenario 1: Courant condition met (c*dt/dx <= 1)
dt1 = 0.01
dx1 = 0.1
duration1 = 30.0

# Scenario 2: Courant condition violated (c*dt/dx > 1)
dt2 = 0.1
dx2 = 0.01
duration2 = 30.0

# Run the simulations for each scenario
y_scenario1 = run_simulation(dt1, dx1, duration1)
y_scenario2 = run_simulation(dt2, dx2, duration2)

# Plotting the results for each scenario
plt.figure(figsize=(12, 6))

# Scenario 1: Courant condition met
plt.subplot(1, 2, 1)
time_indices1 = np.linspace(0, y_scenario1.shape[0] - 1, 20, dtype=int)
for idx in time_indices1:
    plt.plot(np.linspace(0, 10, int(10 / dx1)), y_scenario1[idx, :], label=f"t = {idx * dt1:.1f}")
plt.title("Courant Condition Met")
plt.xlabel("Position (x)")
plt.ylabel("Displacement (y)")
plt.legend()

# Scenario 2: Courant condition violated
plt.subplot(1, 2, 2)
time_indices2 = np.linspace(0, y_scenario2.shape[0] - 1, 20, dtype=int)
for idx in time_indices2:
    plt.plot(np.linspace(0, 10, int(10 / dx2)), y_scenario2[idx, :], label=f"t = {idx * dt2:.1f}")
plt.title("Courant Condition Violated")
plt.xlabel("Position (x)")
plt.ylabel("Displacement (y)")
plt.legend()

plt.tight_layout()
plt.show()
```

Result:



J. Use the plotted time dependence to estimate the peak's propagation velocity c. Result:

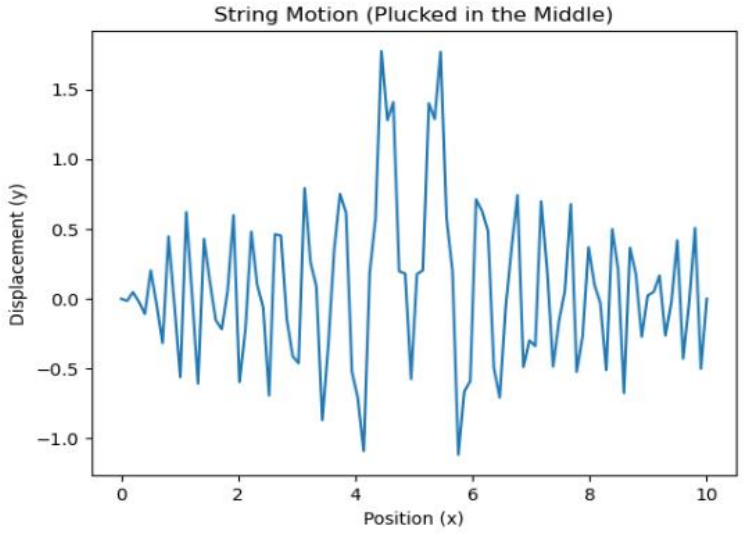
```
Scenario 1: Peak's Propagation Velocity (c) = 0.00
Scenario 2: Peak's Propagation Velocity (c) = -0.17
```



Code:

```
# Scenario 2: Courant condition violated (c*dt/dx > 1)
dt2 = 0.1
dx2 = 0.01
duration2 = 30.0

# Run the simulations for each scenario
y_scenario1 = run_simulation(dt1, dx1, duration1)
y_scenario2 = run_simulation(dt2, dx2, duration2)

# Function to estimate the peak's propagation velocity
def estimate_velocity(y, dt, dx):
    max_indices = np.argmax(y, axis=1)
    peak_velocities = dx / dt * (max_indices[1:] - max_indices[:-1])
    return np.mean(peak_velocities)

# Estimate the peak's propagation velocity for each scenario
c_scenario1 = estimate_velocity(y_scenario1, dt1, dx1)
c_scenario2 = estimate_velocity(y_scenario2, dt2, dx2)

print(f"Scenario 1: Peak's Propagation Velocity (c) = {c_scenario1:.2f}")
print(f"Scenario 2: Peak's Propagation Velocity (c) = {c_scenario2:.2f}")

# Plotting the results for each scenario
plt.figure(figsize=(12, 6))

# Scenario 1: Courant condition met
plt.subplot(1, 2, 1)
time_indices1 = np.linspace(0, y_scenario1.shape[0] - 1, 20, dtype=int)
for idx in time_indices1:
    plt.plot(np.linspace(0, 10, int(10 / dx1)), y_scenario1[idx, :], label=f"t = {idx * dt1:.1f}")
plt.title("Courant Condition Met")
plt.xlabel("Position (x)")
plt.ylabel("Displacement (y)")
plt.legend()

# Scenario 2: Courant condition violated
plt.subplot(1, 2, 2)
time_indices2 = np.linspace(0, y_scenario2.shape[0] - 1, 20, dtype=int)
for idx in time_indices2:
    plt.plot(np.linspace(0, 10, int(10 / dx2)), y_scenario2[idx, :], label=f"t = {idx * dt2:.1f}")
plt.title("Courant Condition Violated")
plt.xlabel("Position (x)")
plt.ylabel("Displacement (y)")
plt.legend()

plt.tight_layout()
plt.show()
```

K. When a string is plucked near its end, a pulse reflects off the ends and bounces back and forth. Change the initial conditions of the program to one corresponding to a string plucked exactly in its middle, and see if a traveling or a standing wave results

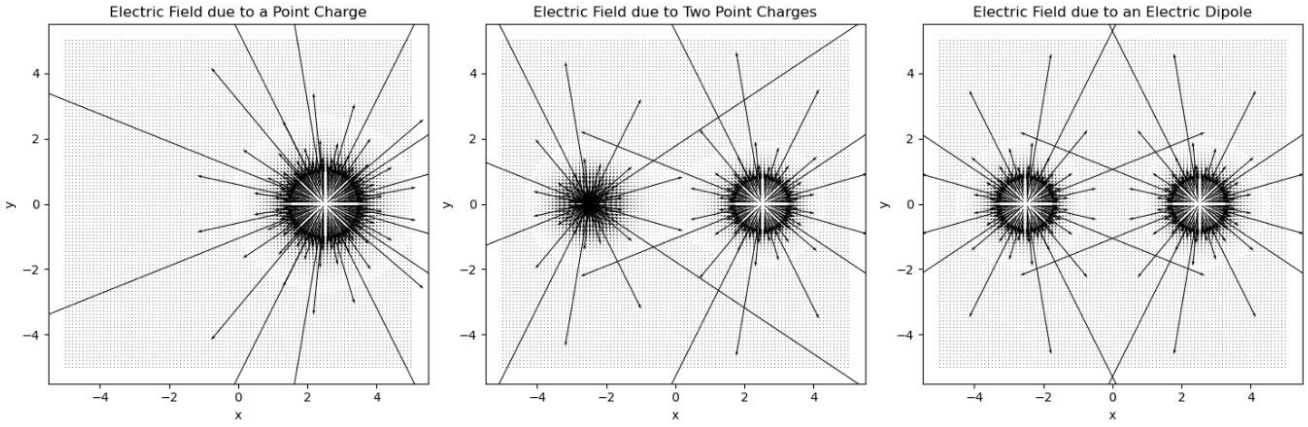


```python
import numpy as np
import matplotlib.pyplot as plt

# Function to run the simulation with given dt and dx
def run_simulation(dt, dx, duration):
    length = 10.0  # Length of the string

    # Calculate number of steps in space and time
    nx = int(length / dx)
    nt = int(duration / dt)

    # Initialize arrays for storing the motion of the string
    y = np.zeros((nt, nx))
    c = 1.0  # Wave speed
    c_prime = dx / dt

    # Initial conditions (a string plucked in the middle)
    midpoint = nx // 2
    y[0, midpoint] = 1.0  # Pluck the string at the midpoint

    # Leapfrog algorithm
    for j in range(1, nt - 1):
        for i in range(1, nx - 1):
            y[j + 1, i] = 2 * y[j, i] - y[j - 1, i] + (c ** 2 / c_prime ** 2) * (
                y[j, i + 1] + y[j, i - 1] - 2 * y[j, i]
            )

    return y

# Define the time step, space step, and duration for the simulation
dt = 0.01
dx = 0.1
duration = 10.0

# Run the simulation
y = run_simulation(dt, dx, duration)

# Plotting the motion of the string
plt.plot(np.linspace(0, 10, int(10 / dx)), y[-1, :])
plt.title("String Motion (Plucked in the Middle)")
plt.xlabel("Position (x)")
plt.ylabel("Displacement (y)")
plt.show()
```

5.5.4 ELECTRIC FIELDS VIA IMAGES

1. Visualize the electric field due to a point charge.
2. Visualize the electric field due to two-point charges.
3. Visualize the electric field due to an electric dipole.

Result:



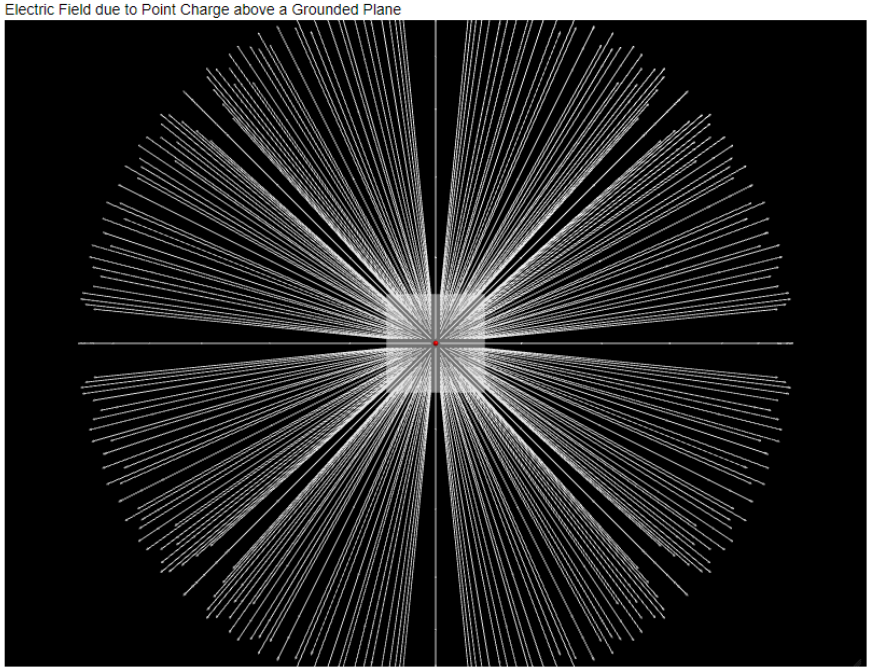
Code:

```
import numpy as np
import matplotlib.pyplot as plt

# Function to calculate electric field due to a single point charge q at position xx
def electric_field_single_charge(x, y, xx, q):
    dx = x - xx[0]
    dy = y - xx[1]
    r = np.sqrt(dx**2 + dy**2)
    E = q * dx / r**3, q * dy / r**3
    return E

# Function to calculate electric field due to two point charges q1 and q2 at positions xx1 and xx2
def electric_field_two_charges(x, y, xx1, q1, xx2, q2):
    dx1 = x - xx1[0]
    dy1 = y - xx1[1]
    dx2 = x - xx2[0]
    dy2 = y - xx2[1]
    r1 = np.sqrt(dx1**2 + dy1**2)
    r2 = np.sqrt(dx2**2 + dy2**2)
    E1 = q1 * dx1 / r1**3, q1 * dy1 / r1**3
    E2 = q2 * dx2 / r2**3, q2 * dy2 / r2**3
    E = E1[0] + E2[0], E1[1] + E2[1]
    return E

# Function to calculate electric field due to an electric dipole formed by two charges q1 and q2 at positions xx1 and xx2
def electric_field_dipole(x, y, xx1, q1, xx2, q2):
    dx1 = x - xx1[0]
    dy1 = y - xx1[1]
    dx2 = x - xx2[0]
    dy2 = y - xx2[1]
    r1 = np.sqrt(dx1**2 + dy1**2)
    r2 = np.sqrt(dx2**2 + dy2**2)
    E1 = q1 * dx1 / r1**3, q1 * dy1 / r1**3
    E2 = q2 * dx2 / r2**3, q2 * dy2 / r2**3
    E = E1[0] - E2[0], E1[1] - E2[1]
    return E

# Setup grid
Nx = 100
Ny = 100
x = np.linspace(-5, 5, Nx)
y = np.linspace(-5, 5, Ny)
X, Y = np.meshgrid(x, y)

# Calculate electric field for each case
E_single_charge = electric_field_single_charge(X, Y, [2.5, 0], 1.0)
E_two_charges = electric_field_two_charges(X, Y, [2.5, 0], 1.0, [-2.5, 0], -1.0)
E_dipole = electric_field_dipole(X, Y, [2.5, 0], 1.0, [-2.5, 0], -1.0)

# Plot the electric field vectors
plt.figure(figsize=(15, 5))

plt.subplot(131)
plt.quiver(X, Y, E_single_charge[0], E_single_charge[1])
plt.title("Electric Field due to a Point Charge")
plt.xlabel("x")
plt.ylabel("y")

plt.subplot(132)
plt.quiver(X, Y, E_two_charges[0], E_two_charges[1])
plt.title("Electric Field due to Two Point Charges")
plt.xlabel("x")
plt.ylabel("y")

plt.subplot(133)
plt.quiver(X, Y, E_dipole[0], E_dipole[1])
plt.title("Electric Field due to an Electric Dipole")
plt.xlabel("x")
plt.ylabel("y")

plt.tight_layout()
plt.show()
```

4. . Use the method of images to determine and visualize the electric field due to a point charge above a grounded, infinite conducting plane (Figure 5.22).

a. Modify the VPython-based program ImagePlaneVP.py so that the vector E at each lattice point is proportional to the strength of the field.

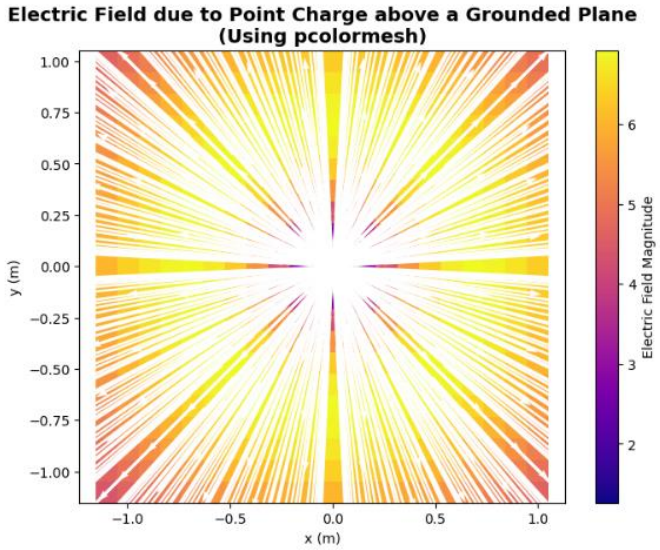
Electric Field due to Point Charge above a Grounded Plane

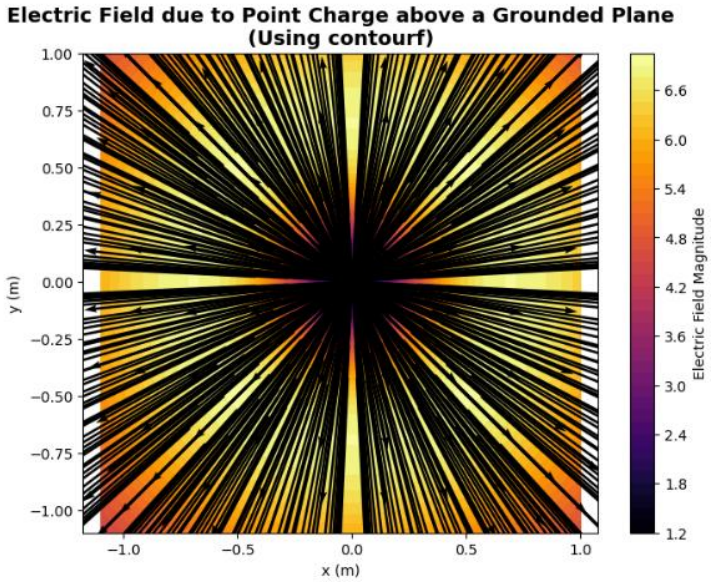


Code:

```
from vpython import *

# Coulomb's constants
k = 8.99e9

# Calculating the electric field due to a point charge
def electric_field(q, r0, r):
    r_diff = r - r0
    r_mag = mag(r_diff)
    r_hat = r_diff / r_mag
    return k * q / (r_mag ** 2) * r_hat

# Scene
scene = canvas(title="Electric Field due to Point Charge above a Grounded Plane", width=800, height=600)

# Point charge and its position
q = 1e-9  # Charge magnitude in C (Coulombs)
charge_pos = vector(0, 0, 1)  # Point charge position in m

# The grounded conducting plane
plane_pos = vector(0, 0, 0)

# Field visualization
num_points = 21  # No. of lattice points along each axis
spacing = 0.1  # Spacing between lattice points in m

# Lattice points in the xy-plane
points = []
for i in range(num_points):
    for j in range(num_points):
        points.append(vector((i - num_points // 2) * spacing, (j - num_points // 2) * spacing, 0))

# Calculating the electric field at each lattice point due to the real and image charges
e_field = [electric_field(q, charge_pos, p) + electric_field(-q, charge_pos, 2 * plane_pos - p) for p in points]

# Creating arrows to visualize the electric field at each lattice point
for p, e in zip(points, e_field):
    arrow(pos=p, axis=e, color=color.white, shaftwidth=0.02, headwidth=0.05, headlength=0.05)

# Point charge and the conducting plane to the visualization
sphere(pos=charge_pos, color=color.red, radius=0.05)
box(pos=plane_pos, size=vector(num_points * spacing, num_points * spacing, 0.01), color=color.gray(0.5))

# Display the scene
scene.camera.center = vector(0, 0, 0)  # Center the camera at the origin
scene.autoscale = False  # Turn off automatic scene scaling
```

b. Modify the Matplotlib-based program **ImagePlaneMat.py** that uses streamlines to use other visualization methods such as **pcolormesh** with levels and **contourf** with levels.





Code:

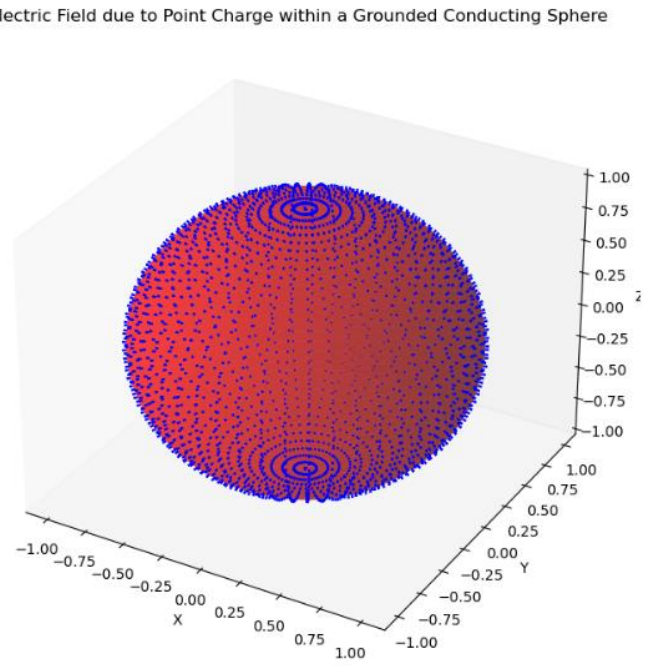
```
import matplotlib.pyplot as plt
import numpy as np

# Coulomb's constants
k = 8.99e9

# Calculating the electric field due to a point charge
def electric_field(q, r0, r):
    r_diff = r - r0
    r_mag = np.linalg.norm(r_diff)
    r_hat = r_diff / r_mag
    return k * q / (r_mag ** 2) * r_hat

# Define the point charge and its position, and grounded conducting plane
q = 1e-9 # Charge magnitude in C
charge_pos = np.array([0, 0, 1]) # Position in m
plane_pos = np.array([0, 0, 0]) # Grounded conducting plane

# Lattice points for the field visualization
num_points = 21 # Number of lattice points along each axis
spacing = 0.1 # Spacing between lattice points (in meters)

# Meshgrid for lattice points in the xy-plane
x = np.linspace(-num_points // 2, num_points // 2, num_points) * spacing
X, Y = np.meshgrid(x, x)
points = np.column_stack((X.ravel(), Y.ravel(), np.zeros_like(X.ravel())))

# Electric field at each lattice point due to the real and image charges
e_field = np.array([electric_field(q, charge_pos, p) + electric_field(-q, charge_pos, 2 * plane_pos - p) for p in points])
Ex = e_field[:, 0].reshape(num_points, num_points)
Ey = e_field[:, 1].reshape(num_points, num_points)

# Electric field using pcolormesh with levels
plt.figure(figsize=(8, 6))
plt.pcolormesh(X, Y, np.sqrt(Ex ** 2 + Ey ** 2), shading='auto', cmap='plasma')
plt.colorbar(label='Electric Field Magnitude')
plt.quiver(X, Y, Ex, Ey, color='white', scale=10, pivot='middle', width=0.005)
plt.xlabel('x (m)')
plt.ylabel('y (m)')
plt.title('Electric Field due to Point Charge above a Grounded Plane\n(Using pcolormesh)', fontsize=14, fontweight='bold')
plt.axis('equal')
plt.show()

# Electric field using contourf with levels
plt.figure(figsize=(8, 6))
plt.contourf(X, Y, np.sqrt(Ex ** 2 + Ey ** 2), levels=50, cmap='inferno')
plt.colorbar(label='Electric Field Magnitude')
plt.quiver(X, Y, Ex, Ey, color='black', scale=10, pivot='middle', width=0.005)
plt.xlabel('x (m)')
plt.ylabel('y (m)')
plt.title('Electric Field due to Point Charge above a Grounded Plane\n(Using contourf)', fontsize=14, fontweight='bold')
plt.axis('equal')
plt.show()
```

5. Use the method of images to determine and visualize the electric field due to a point charge within a grounded conducting sphere. As seen in Figure 5.23, the conducting sphere of radius a is centered at the origin, the point charge q is located at d, and the image charge q ′ is located at d′ . With d ′ = a 2/d, the net potential on the surface of the sphere is zero

Code:

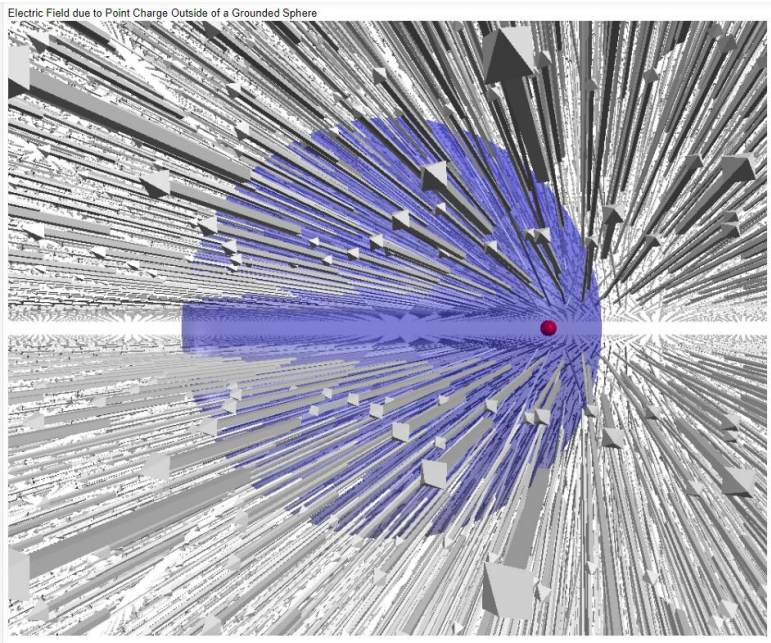
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Grid of points on the surface of the sphere
def electric_field_due_to_point_charge(q, d, a, num_points=50):
    theta, phi = np.meshgrid(np.linspace(0, np.pi, num_points),
                             np.linspace(0, 2 * np.pi, num_points))
    x = a * np.sin(theta) * np.cos(phi)
    y = a * np.sin(theta) * np.sin(phi)
    z = a * np.cos(theta)

    # Position vector of the charge and the image charge
    r_charge = np.array([d, 0, 0])
    r_image = np.array([a ** 2 / d, 0, 0])

    # The electric field due to the charge and image charge
    r = np.sqrt((x - r_charge[0]) ** 2 + y ** 2 + z ** 2)
    r_image_charge = np.sqrt((x - r_image[0]) ** 2 + y ** 2 + z ** 2)

    e_charge = q / (4 * np.pi * 8.85e-12 * r ** 3) * np.array([x - r_charge[0], y, z])
    e_image_charge = -q / (4 * np.pi * 8.85e-12 * r_image_charge ** 3) * np.array([x - r_image[0], y, z])

    # Calculating the net electric field (superposition of charge and image charge)
    e_total = e_charge + e_image_charge
    return x, y, z, e_total

def visualize_electric_field(x, y, z, e_total):
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    # Normalizing the electric field vectors for visualization purposes
    e_norm = np.sqrt(np.sum(e_total ** 2, axis=0))
    e_total /= e_norm

    # Scaling the electric field vectors for better visualization
    scale_factor = 0.02
    e_total *= scale_factor

    # Plotting the electric field vectors
    ax.quiver(x, y, z, e_total[0], e_total[1], e_total[2], length=1, normalize=False, color='b')

    # Plotting the conducting sphere
    a = np.max([np.abs(x), np.abs(y), np.abs(z)])
    sphere = ax.plot_surface(x, y, z, rstride=1, cstride=1, alpha=0.5, color='r', linewidth=0, antialiased=True)

    # Set axis labels and title
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    ax.set_title('Electric Field due to Point Charge within a Grounded Conducting Sphere')

    # Hide the grid
    ax.grid(False)

    # Show the plot
    plt.show()

# Parameters
q = 1e-9  # Charge magnitude in C
d = 0.5  # Distance of point charge from the origin
a = 1.0  # Radius of the conducting sphere

# Calculate the electric field due to the point charge within the conducting sphere
x, y, z, e_total = electric_field_due_to_point_charge(q, d, a)

# Visualize the electric field in 3D
visualize_electric_field(x, y, z, e_total)
```

Result:



6. Use the method of images to determine and visualize the electric field due to a point charge outside of a grounded, conducting sphere.

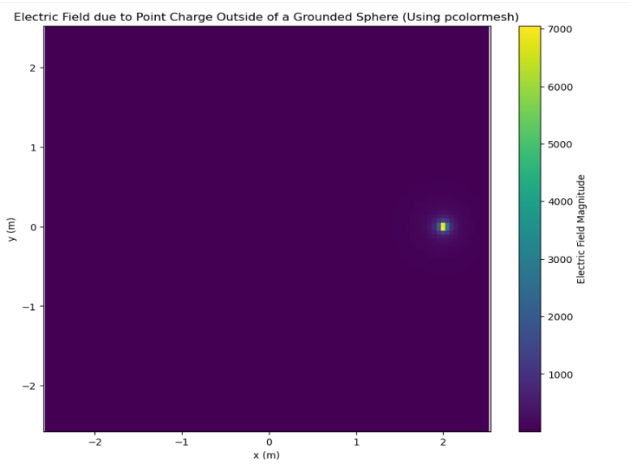
a. Modify the Visual-based program ImageSphereVis.py so that the vector E at each lattice point is proportional to the strength of the field

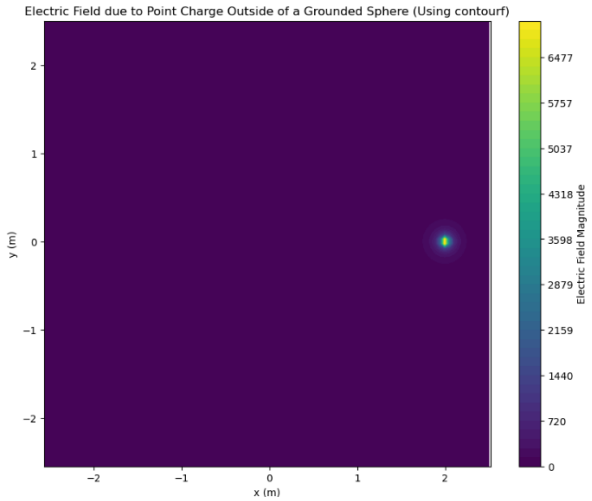


Code:

```
from vpython import *
import numpy as np

# Coulomb's constant
k = 8.99e9

# Function to calculate the electric field due to a point charge
def electric_field_single_charge(r, q, r0):
    d = r - r0
    E = k * q * d / mag(d)**3
    return E

# Function to calculate the electric field due to the image charge
def electric_field_image_charge(r, q, r0, a):
    r_image = r0 * (a**2 / mag(r0)**2)
    d = r - r_image
    E = k * q * d / mag(d)**3
    return E

# Function to calculate the total electric field at a point
def total_electric_field(r, q, r0, a):
    E_real = electric_field_single_charge(r, q, r0)
    E_image = electric_field_image_charge(r, q, r0, a)
    E_total = E_real + E_image
    return E_total

# Scene setup
scene = canvas(title="Electric Field due to Point Charge Outside of a Grounded Sphere", width=1000, height=800, center=vector(0,

# Parameters
q = 1e-9  # Charge magnitude in C (Coulombs)
r0 = vector(2, 0, 0)  # Position of the point charge in m
a = 2.5  # Radius of the grounded conducting sphere in m

# Field visualization
num_points = 51  # Number of lattice points along each axis
spacing = 0.5  # Spacing between lattice points in m
points = []
for x in np.linspace(-num_points // 2, num_points // 2, num_points) * spacing:
    for y in np.linspace(-num_points // 2, num_points // 2, num_points) * spacing:
        for z in np.linspace(-num_points // 2, num_points // 2, num_points) * spacing:
            points.append(vector(x, y, z))

electric_field_vectors = []
for r in points:
    E = total_electric_field(r, q, r0, a)
    electric_field_vectors.append((E, r))

# 3D Visualization using VPython
for E, r in electric_field_vectors:
    arrow(pos=r, axis=E, color=color.white, shaftwidth=0.02, headwidth=0.05, headlength=0.05)

# Point charge and the conducting sphere to the visualization
sphere(pos=r0, color=color.red, radius=0.1)
sphere(pos=vector(0, 0, 0), radius=a, opacity=0.3, color=color.blue)
```

b. Modify the Matplotlib-based program ImageSphereMat.py to use other visualization methods such as pcolormesh with levels and contourf with levels





Code:

```
# Function to calculate the electric field due to the image charge
def electric_field_image_charge(r, q, r0, a):
    r_image = r0 * (a**2 / np.linalg.norm(r0)**2)
    d = r - r_image
    E = k * q * d / np.linalg.norm(d)**3
    return E

# Function to calculate the total electric field at a point
def total_electric_field(r, q, r0, a):
    E_real = electric_field_single_charge(r, q, r0)
    E_image = electric_field_image_charge(r, q, r0, a)
    E_total = E_real + E_image
    return E_total

# Parameters
q = 1e-9  # Charge magnitude in C (Coulombs)
r0 = np.array([2, 0, 0])  # Position of the point charge in m
a = 2.5  # Radius of the grounded conducting sphere in m

# Field visualization
num_points = 101  # Number of lattice points along each axis
spacing = 0.05  # Spacing between lattice points in m

# Meshgrid for lattice points
x = np.linspace(-num_points // 2, num_points // 2, num_points) * spacing
X, Y, Z = np.meshgrid(x, x, x)

# Calculate the electric field at each lattice point
r = np.column_stack((X.ravel(), Y.ravel(), Z.ravel()))
E_total = np.array([total_electric_field(point, q, r0, a) for point in r])
E_magnitude = np.linalg.norm(E_total, axis=1).reshape(num_points, num_points, num_points)

# Electric field using pcolormesh with levels
plt.figure(figsize=(10, 8))
plt.pcolormesh(X[:, :, 0], Y[:, :, 0], E_magnitude[:, :, num_points // 2], shading='auto')
plt.colorbar(label='Electric Field Magnitude')
plt.xlabel('x (m)')
plt.ylabel('y (m)')
plt.title('Electric Field due to Point Charge Outside of a Grounded Sphere (Using pcolormesh)')
plt.axis('equal')
plt.show()

# Electric field using contourf with levels
plt.figure(figsize=(10, 8))
levels = np.linspace(E_magnitude.min(), E_magnitude.max(), 50)
plt.contourf(X[:, :, 0], Y[:, :, 0], E_magnitude[:, :, num_points // 2], levels=levels, cmap='viridis')
plt.colorbar(label='Electric Field Magnitude')
plt.xlabel('x (m)')
plt.ylabel('y (m)')
plt.title('Electric Field due to Point Charge Outside of a Grounded Sphere (Using contourf)')
plt.axis('equal')
plt.show()
```