Numerical Assignment Todd Blacklaw H00368671

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

#  Part (a) Electric Field due to Two Point Charges
print("Part A")

# Physical Constants
epsilon_0 = 8.854e-12

k = 1 / (4 * np.pi * epsilon_0)

q = 1e-9

# Define positions
r1 = np.array([0.0, 0.0, 0.0])  # Charge at the origin
r2 = np.array([1.0, 0.0, 0.0])  # Charge at (1, 0, 0)

def electric_field_point_charge(r, q, rq):
    """
    Compute the electric field at point r due to a point charge q at position rq.

    Parameters:
    r : numpy array
        The position vector where the electric field is calculated.
    q : float
        The charge of the point charge.
    rq : numpy array
        The position vector of the point charge.

    Returns:
    E : numpy array
```

```
    The electric field vector at point r.
  """
  r_rel = r - rq
  r_norm = np.linalg.norm(r_rel)
  if r_norm == 0:
    return np.array([0.0, 0.0, 0.0])  # Avoid division by zero at the location of the charge
  E = k * q * r_rel / r_norm**3
  return E

def total_electric_field(r):
  """
  Compute the total electric field at point r due to both point charges.

  Parameters:
  r : numpy array
    The position vector where the electric field is calculated.

  Returns:
  E_total : numpy array
    The total electric field vector at point r.
  """
  E1 = electric_field_point_charge(r, q, r1)
  E2 = electric_field_point_charge(r, q, r2)
  E_total = E1 + E2
  return E_total

# Testing with symmetry considerations
# Arrangement 1 Charges along the x-axis at (-a, 0, 0) and (a, 0, 0)

# Update charge positions for symmetry
a = 1.0
```

Numerical Assignment Todd Blacklaw H00368671

```
r1 = np.array([-a, 0.0, 0.0])
r2 = np.array([a, 0.0, 0.0])

# Test point along the y-axis
test_point = np.array([0.0, 0.5, 0.0])

# Compute electric field at test point
E_total = total_electric_field(test_point)

print("Electric field at point {} due to two charges at positions {} and {}:".format(test_point, r1, r2))
print("E_total = [{}] V/m".format(E_total))

#The x-components should cancel out, and E_total should be along the y-axis.

# Arrangement 2 Charges along the y-axis at (0, -a, 0) and (0, a, 0)

# Update charge positions for second symmetry test
r1 = np.array([0.0, -a, 0.0])
r2 = np.array([0.0, a, 0.0])

# Test point along the x-axis
test_point = np.array([0.5, 0.0, 0.0])

# Compute electric field at test point
E_total = total_electric_field(test_point)

print("\nElectric field at point {} due to two charges at positions {} and {}:".format(test_point, r1,
r2))
print("E_total = [{}] V/m".format(E_total))

# The y-components should cancel out, and E_total should be along the x-axis.
```

Numerical Assignment Todd Blacklaw H00368671

```
#-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------
# Part (b) Generating a Uniform Distribution of Point Charges in a Sphere

print("Part B")

Q_total = 1e-6

N = 100000

# Generate random positions in the cube
np.random.seed(0)
x_cube = np.random.uniform(-1, 1, N)
y_cube = np.random.uniform(-1, 1, N)
z_cube = np.random.uniform(-1, 1, N)

# Compute radial distances from the origin
radii = np.sqrt(x_cube**2 + y_cube**2 + z_cube**2)

# Select points inside the unit sphere (r <= 1)
inside_sphere = radii <= 1.0
x_sphere = x_cube[inside_sphere]
y_sphere = y_cube[inside_sphere]
z_sphere = z_cube[inside_sphere]

# Number of point charges inside the sphere
M = len(x_sphere)

q = Q_total / M
```

Numerical Assignment Todd Blacklaw H00368671

```
print(f"Total number of point charges generated: {N}")
print(f"Number of point charges inside the sphere: {M}")
print(f"Charge per point charge: {q} C")

# Plotting the positions of the point charges inside the sphere
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x_sphere, y_sphere, z_sphere, s=1, color='blue')

# Setting plot labels and title
ax.set_title('Uniform Distribution of Point Charges in a Sphere')
ax.set_xlabel('X (m)')
ax.set_ylabel('Y (m)')
ax.set_zlabel('Z (m)')
ax.set_xlim([-1, 1])
ax.set_ylim([-1, 1])
ax.set_zlim([-1, 1])

plt.show()

total_charge = q * M
print(f"Total charge of the sphere: {total_charge} C")

#-------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------
# Part (c) Computing the Electric Field Along a Radial Line from a Uniformly Charged Spheret

print("Part C")

try:
   x_sphere
```

```
except NameError:

    # Re-run the code from part (b)

    Q_total = 1e-6

    N = 100000

    # Generate random positions in the cube
    np.random.seed(0)
    x_cube = np.random.uniform(-1, 1, N)
    y_cube = np.random.uniform(-1, 1, N)
    z_cube = np.random.uniform(-1, 1, N)

    # Compute radial distances from the origin
    radii = np.sqrt(x_cube**2 + y_cube**2 + z_cube**2)

    # Select points inside the unit sphere (r <= 1)
    inside_sphere = radii <= 1.0
    x_sphere = x_cube[inside_sphere]
    y_sphere = y_cube[inside_sphere]
    z_sphere = z_cube[inside_sphere]

    M = len(x_sphere)

    q = Q_total / M

# Positions of the point charges
r_charges = np.vstack((x_sphere, y_sphere, z_sphere)).T

# Function to compute the electric field at a point r due to multiple point charges
```

Numerical Assignment Todd Blacklaw H00368671

```
def electric_field_multiple_charges(r, q_array, r_charges):
    """
    Compute the electric field at point r due to multiple point charges.

    Parameters:
    r : numpy array
        The position vector where the electric field is calculated.
    q_array : numpy array
        Array of point charges.
    r_charges : numpy array
        Positions of the point charges.

    Returns:
    E_total : numpy array
        The total electric field vector at point .
    """
    r_vectors = r - r_charges
    r_norms = np.linalg.norm(r_vectors, axis=1)
    # Avoid division by zero
    zero_mask = r_norms == 0
    r_norms[zero_mask] = np.inf  # Temporarily set zero distances to infinity to avoid division by zero
    r_vectors[zero_mask] = 0  # Zero vector where distance is zero

    E_vectors = k * q_array[:, np.newaxis] * r_vectors / r_norms[:, np.newaxis]**3
    E_total = np.sum(E_vectors, axis=0)
    return E_total

num_points = 100
r_values = np.linspace(0, 5, num_points)
```

Numerical Assignment Todd Blacklaw H00368671

```
# Initialize arrays
E_magnitudes = np.zeros(num_points)

# Charges and their positions
q_array = np.full(len(x_sphere), q)

# Compute the electric field at each point along the line
for i, r in enumerate(r_values):
    position = np.array([r, 0.0, 0.0])
    E_total = electric_field_multiple_charges(position, q_array, r_charges)
    E_magnitude = np.linalg.norm(E_total)
    E_magnitudes[i] = E_magnitude

# Plotting the magnitude of the electric field as a function of distance
plt.figure(figsize=(8, 6))
plt.plot(r_values, E_magnitudes, label='Numerical Electric Field')

# compute the theoretical electric field of a uniformly charged sphere
# Inside the sphere (r <= 1 m): E = (1 / (4 * pi * epsilon_0)) * (Q_total * r) / R^3
# Outside the sphere (r > 1 m): E = (1 / (4 * pi * epsilon_0)) * (Q_total) / r^2

E_theoretical = np.zeros(num_points)
for i, r in enumerate(r_values):
    if r <= 1.0:
        # Inside the sphere
        E_theoretical[i] = (1 / (4 * np.pi * epsilon_0)) * (Q_total * r) / (1.0)**3
    else:
        # Outside the sphere
        E_theoretical[i] = (1 / (4 * np.pi * epsilon_0)) * Q_total / r**2

plt.plot(r_values, E_theoretical, 'r--', label='Theoretical Electric Field')
```

Numerical Assignment Todd Blacklaw H00368671

```
# Setting plot labels and title
plt.title('Magnitude of Electric Field vs. Distance from Centre of Sphere')
plt.xlabel('Distance from Centre (m)')
plt.ylabel('Electric Field Magnitude (V/m)')
plt.legend()
plt.grid(True)
plt.show()

# Adjusting the Number of Point Charges to Achieve Desired Accuracy

Q_total = 1e-6

# Desired accuracy
tolerance = 0.01

r_check = np.linspace(1.1, 5, 50)

# Theoretical electric field due to a point charge at the center
def theoretical_electric_field(r):
    return (1 / (4 * np.pi * epsilon_0)) * Q_total / r**2

E_theoretical = theoretical_electric_field(r_check)

# Function to compute the electric field components at a point r due to multiple point charges
def electric_field_components(r, q_array, r_charges):
    """
    Compute the electric field vector at point r due to multiple point charges.

    Parameters:
    r : numpy array
```

```
    The position vector where the electric field is calculated.
  q_array : numpy array
    Array of point charges .
  r_charges : numpy array
    Positions of the point charges.

  Returns:
  E_total : numpy array
    The total electric field vector at point r.
  """
  r_vectors = r - r_charges
  r_norms = np.linalg.norm(r_vectors, axis=1)
  # Avoid division by zero
  zero_mask = r_norms == 0
  r_norms[zero_mask] = np.inf  # Set zero distances to infinity to avoid division by zero
  r_vectors[zero_mask] = 0  # Zero vector where distance is zero

  E_vectors = k * q_array[:, np.newaxis] * r_vectors / r_norms[:, np.newaxis]**3
  E_total = np.sum(E_vectors, axis=0)
  return E_total

# Function to generate point charges inside the sphere
def generate_point_charges(N_total):
  """
  Generate N_total point charges uniformly distributed inside a unit sphere.

  Parameters:
  N_total : int
    Total number of random points to generate in the cube before filtering.

  Returns:
```

```
    q_array : numpy array
        Array of point charges (shape (M,)).
    r_charges : numpy array
        Positions of the point charges.
    """
    # Generate random positions in the cube
    x_cube = np.random.uniform(-1, 1, N_total)
    y_cube = np.random.uniform(-1, 1, N_total)
    z_cube = np.random.uniform(-1, 1, N_total)

    # Compute radial distances from the origin
    radii = np.sqrt(x_cube**2 + y_cube**2 + z_cube**2)

    # Select points inside the unit sphere (r <= 1)
    inside_sphere = radii <= 1.0
    x_sphere = x_cube[inside_sphere]
    y_sphere = y_cube[inside_sphere]
    z_sphere = z_cube[inside_sphere]

    M = len(x_sphere)

    q = Q_total / M

    # Positions of the point charges
    r_charges = np.vstack((x_sphere, y_sphere, z_sphere)).T
    q_array = np.full(M, q)

    return q_array, r_charges

# Function to compute the maximum relative error for r > 1.1 m
def compute_max_relative_error(q_array, r_charges):
```

```
    E_magnitudes = []
    for r in r_check:
        position = np.array([r, 0.0, 0.0])
        E_total = electric_field_components(position, q_array, r_charges)
        E_magnitude = np.linalg.norm(E_total)
        E_magnitudes.append(E_magnitude)
    E_magnitudes = np.array(E_magnitudes)
    relative_errors = np.abs(E_magnitudes - E_theoretical) / E_theoretical
    max_relative_error = np.max(relative_errors)
    return max_relative_error, E_magnitudes

# Main loop to adjust N until the desired accuracy is achieved
N_total = 10000
max_iterations = 10
for iteration in range(max_iterations):
    np.random.seed(0)
    q_array, r_charges = generate_point_charges(N_total)
    max_error, E_magnitudes = compute_max_relative_error(q_array, r_charges)
    print(f"Iteration {iteration + 1}: N_total = {N_total}, Max Relative Error = {max_error:.4f}")
    if max_error <= tolerance:
        print("Desired accuracy achieved.")
        break
    else:
        N_total *= 2
else:
    print("Maximum iterations reached without achieving desired accuracy.")

# Compute electric field components along the line from r = 0 to r = 5 m
num_points = 100
r_values = np.linspace(0, 5, num_points)
```

Numerical Assignment Todd Blacklaw H00368671

```
E_components = np.zeros((num_points, 3))
E_magnitudes_full = np.zeros(num_points)

for i, r in enumerate(r_values):
    position = np.array([r, 0.0, 0.0])
    E_total = electric_field_components(position, q_array, r_charges)
    E_components[i, :] = E_total
    E_magnitudes_full[i] = np.linalg.norm(E_total)

# Plotting the electric field components and magnitude as functions of r
plt.figure(figsize=(10, 6))
plt.plot(r_values, E_components[:, 0], label='Ex')
plt.plot(r_values, E_components[:, 1], label='Ey')
plt.plot(r_values, E_components[:, 2], label='Ez')
plt.plot(r_values, E_magnitudes_full, 'k--', label='|E|')

plt.title('Electric Field Components vs. Distance from Centre of Sphere')
plt.xlabel('Distance from Centre (m)')
plt.ylabel('Electric Field (V/m)')
plt.legend()
plt.grid(True)
plt.show()

#-----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------

# Part (d) Normalizing Position Vectors to Sample Points Uniformly on the Surface of the Sphere
# Assuming we have the point charges from part (c)

print("Part D")
```

Numerical Assignment Todd Blacklaw H00368671

```
try:
    x_sphere
except NameError:
    # Re-run the code from part (c) to generate the point charges inside the sphere

    Q_total = 1e-6
    N = 100000
    np.random.seed(0)
    x_cube = np.random.uniform(-1, 1, N)
    y_cube = np.random.uniform(-1, 1, N)
    z_cube = np.random.uniform(-1, 1, N)
    radii = np.sqrt(x_cube**2 + y_cube**2 + z_cube**2)
    inside_sphere = radii <= 1.0
    x_sphere = x_cube[inside_sphere]
    y_sphere = y_cube[inside_sphere]
    z_sphere = z_cube[inside_sphere]
    M = len(x_sphere)
    q = Q_total / M

# Combine the coordinates into a single array for easier processing
positions = np.vstack((x_sphere, y_sphere, z_sphere)).T

# Normalize the position vectors
norms = np.linalg.norm(positions, axis=1)
positions_normalized = positions / norms[:, np.newaxis]

# Plotting the normalized positions on the surface of the sphere
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(positions_normalized[:, 0], positions_normalized[:, 1], positions_normalized[:, 2],
s=1, color='green')
```

```
# Setting plot labels and title
ax.set_title('Distribution of Point Charges on the Surface of the Unit Sphere')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_xlim([-1, 1])
ax.set_ylim([-1, 1])
ax.set_zlim([-1, 1])
ax.set_box_aspect([1,1,1])  # Equal aspect ratio

plt.show()

#-------------------------------------------------------------------------------------------------------------------------
# Part (e) Adjusting the Number of Point Charges on the Sphere's Surface to Achieve Desired Accuracy

print("Part E")

Q_total = 1e-6

tolerance_relative = 0.01
tolerance_absolute = 0.01 * (1 / (4 * np.pi * epsilon_0)) * Q_total / (1.1)**2

# Radial distances where we check the electric field
r_check_inside = np.linspace(0.1, 0.9, 10)
r_check_outside = np.linspace(1.1, 5, 20)
r_check = np.concatenate((r_check_inside, r_check_outside))

# Theoretical electric field for a uniformly charged spherical shell
def theoretical_electric_field_shell(r):
```

```
    E = np.zeros_like(r)
    # Inside the shell (r < R): E = 0
    # Outside the shell (r >= R): E = (1 / (4 * pi * epsilon_0)) * (Q_total) / r^2
    mask_outside = r >= 1.0
    E[mask_outside] = (1 / (4 * np.pi * epsilon_0)) * Q_total / r[mask_outside]**2
    return E

E_theoretical = theoretical_electric_field_shell(r_check)

# Function to compute the electric field at a point r due to multiple point charges
def electric_field_multiple_charges(r, q_array, r_charges):
    r_vectors = r - r_charges
    r_norms = np.linalg.norm(r_vectors, axis=1)
    # Avoid division by zero
    zero_mask = r_norms == 0
    r_norms[zero_mask] = np.inf
    r_vectors[zero_mask] = 0

    E_vectors = k * q_array[:, np.newaxis] * r_vectors / r_norms[:, np.newaxis]**3
    E_total = np.sum(E_vectors, axis=0)
    return E_total

# Function to generate N point charges uniformly distributed on the sphere's surface
def generate_surface_point_charges(N):
    np.random.seed(0)
    phi = np.random.uniform(0, 2 * np.pi, N)      # Azimuthal angle
    cos_theta = np.random.uniform(-1, 1, N)       # Cosine of polar angle
    theta = np.arccos(cos_theta)                  # Polar angle

    x = np.sin(theta) * np.cos(phi)
    y = np.sin(theta) * np.sin(phi)
```

```
    z = cos_theta

    r_charges = np.vstack((x, y, z)).T

    q = Q_total / N
    q_array = np.full(N, q)

    return q_array, r_charges

# Function to compute maximum errors
def compute_max_errors(q_array, r_charges):
    E_magnitudes = []
    errors_relative = []
    errors_absolute = []

    for r_value, E_th in zip(r_check, E_theoretical):
        position = np.array([r_value, 0.0, 0.0])
        E_total = electric_field_multiple_charges(position, q_array, r_charges)
        E_magnitude = np.linalg.norm(E_total)
        E_magnitudes.append(E_magnitude)

        if E_th != 0:
            # Outside the shell
            error = np.abs(E_magnitude - E_th) / E_th
            errors_relative.append(error)
        else:
            # Inside the shell
            error = np.abs(E_magnitude)
            errors_absolute.append(error)

    max_relative_error = max(errors_relative) if errors_relative else 0
```

```
    max_absolute_error = max(errors_absolute) if errors_absolute else 0


    return max_relative_error, max_absolute_error, E_magnitudes


# Main loop N until desired accuracy is achieved
N = 1000
max_iterations = 10
for iteration in range(max_iterations):
    q_array, r_charges = generate_surface_point_charges(N)
    max_rel_error, max_abs_error, E_magnitudes = compute_max_errors(q_array, r_charges)


    print(f"Iteration {iteration + 1}: N = {N}, Max Relative Error (outside) = {max_rel_error:.4f}, Max
Absolute Error (inside) = {max_abs_error:.4e}")


    if max_rel_error <= tolerance_relative and max_abs_error <= tolerance_absolute:
        print("Desired accuracy achieved.")
        break
    else:
        N *= 2
else:
    print("Maximum iterations reached without achieving desired accuracy.")


# Determine the required order of magnitude for N
order_of_magnitude = int(np.floor(np.log10(N)))
print(f"\nUp to the nearest order of magnitude, the number of sample points required is
10^{order_of_magnitude}.")


# Compute electric field magnitude along the line
num_points = 200
r_values = np.linspace(0, 5, num_points)


E_magnitudes_full = []
```

Numerical Assignment Todd Blacklaw H00368671

```
E_theoretical_full = theoretical_electric_field_shell(r_values)

for r_value in r_values:
    position = np.array([r_value, 0.0, 0.0])
    E_total = electric_field_multiple_charges(position, q_array, r_charges)
    E_magnitude = np.linalg.norm(E_total)
    E_magnitudes_full.append(E_magnitude)

E_magnitudes_full = np.array(E_magnitudes_full)

# Plotting the magnitude of the electric field
plt.figure(figsize=(10, 6))
plt.plot(r_values, E_magnitudes_full, label='Numerical |E|')
plt.plot(r_values, E_theoretical_full, 'r--', label='Theoretical |E|')

plt.title(f'Electric Field Magnitude vs. Distance (N = {N})')
plt.xlabel('Distance from Centre (m)')
plt.ylabel('Electric Field Magnitude (V/m)')
plt.legend()
plt.grid(True)
plt.show()

#-------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------
# Part F Plot Hollow Sphere Charge Desnity and Plot Electric Field along Z-Axis
print("Part F")

Q_total = 1e-6

N = 100000
```

Numerical Assignment Todd Blacklaw H00368671

```
# Generate random angles for uniform distribution on sphere surface
np.random.seed(0)
phi = np.random.uniform(0, 2 * np.pi, N)      # Azimuthal angle
cos_theta = np.random.uniform(-1, 1, N)       # Cosine of polar angle
theta = np.arccos(cos_theta)                  # Polar angle

# Convert spherical coordinates to Cartesian coordinates
x = np.sin(theta) * np.cos(phi)
y = np.sin(theta) * np.sin(phi)
z = cos_theta

# Positions of the point charges
r_charges_full = np.vstack((x, y, z)).T

# Discard points belonging to the upper hemisphere (z > 0
mask_lower_hemisphere = z <= 0
x_hemisphere = x[mask_lower_hemisphere]
y_hemisphere = y[mask_lower_hemisphere]
z_hemisphere = z[mask_lower_hemisphere]
r_charges = np.vstack((x_hemisphere, y_hemisphere, z_hemisphere)).T

# Number of point charges in the lower hemisphere
N_hemisphere = len(x_hemisphere)

# Total charge of the hemisphere
Q_hemisphere = Q_total / 2

# Charge per point charge
q = Q_hemisphere / N_hemisphere  #
q_array = np.full(N_hemisphere, q)
```

```
print(f"Number of point charges in the hemisphere: {N_hemisphere}")
print(f"Charge per point charge: {q} C")

# Plotting the distribution of point charges on the hemisphere
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(r_charges[:, 0], r_charges[:, 1], r_charges[:, 2], s=1, color='blue')

# Setting plot labels and title
ax.set_title('Point Charges on the Lower Hemisphere')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_xlim([-1, 1])
ax.set_ylim([-1, 1])
ax.set_zlim([-1, 1])
ax.set_box_aspect([1,1,1])

plt.show()

# Function to compute the electric field at a point r due to multiple point charges
def electric_field_multiple_charges(r, q_array, r_charges):
    """
    Compute the electric field at point r due to multiple point charges.

    Parameters:
    r : numpy array
        The position vector where the electric field is calculated (shape (3,)).
    q_array : numpy array
        Array of point charges (shape (N,)).
    r_charges : numpy array
```

Numerical Assignment Todd Blacklaw H00368671

```
    Positions of the point charges (shape (N, 3)).

  Returns:
  E_total : numpy array
    The total electric field vector at point r (shape (3,)).
  """
  r_vectors = r - r_charges
  r_norms = np.linalg.norm(r_vectors, axis=1)
  # Avoid division by zero
  zero_mask = r_norms == 0
  r_norms[zero_mask] = np.inf
  r_vectors[zero_mask] = 0



  E_vectors = k * q_array[:, np.newaxis] * r_vectors / r_norms[:, np.newaxis]**3
  E_total = np.sum(E_vectors, axis=0)
  return E_total

# Define the range along the z-axis
z_values = np.linspace(-0.9, 3.0, 200)
E_z_values = []

# Compute the electric field at points along the z-axis

for z_point in z_values:
  position = np.array([0.0, 0.0, z_point])
  E_total = electric_field_multiple_charges(position, q_array, r_charges)
  E_magnitude = np.linalg.norm(E_total)
  E_z = E_total[2]
  E_z_values.append(E_z)
```

```
E_z_values = np.array(E_z_values)


# Plotting the electric field along the z-axis
plt.figure(figsize=(10, 6))
plt.plot(z_values, E_z_values, label='E_z (Numerical)')


plt.title('Electric Field Produced by a Uniformly Charged Hemispherical Surface')
plt.xlabel('z (m)')
plt.ylabel('Electric Field E_z (V/m)')
plt.legend()
plt.grid(True)
plt.show()
```

```
[Running] python -u "c:\Users\toddb\Desktop\Uni\test2.py"
Part A
Electric field at point [0.  0.5 0. ] due to two charges at positions [-1.  0.  0.] and [1. 0. 0.]:
E_total = [[0.         6.43110498 0.        ]] V/m

Electric field at point [0.5 0.  0. ] due to two charges at positions [ 0. -1.  0.] and [0. 1. 0.]:
E_total = [[6.43110498 0.         0.        ]] V/m
Part B
Total number of point charges generated: 100000
Number of point charges inside the sphere: 52173
Charge per point charge: 1.9167002089203226e-11 C
Total charge of the sphere: 1e-06 C
```

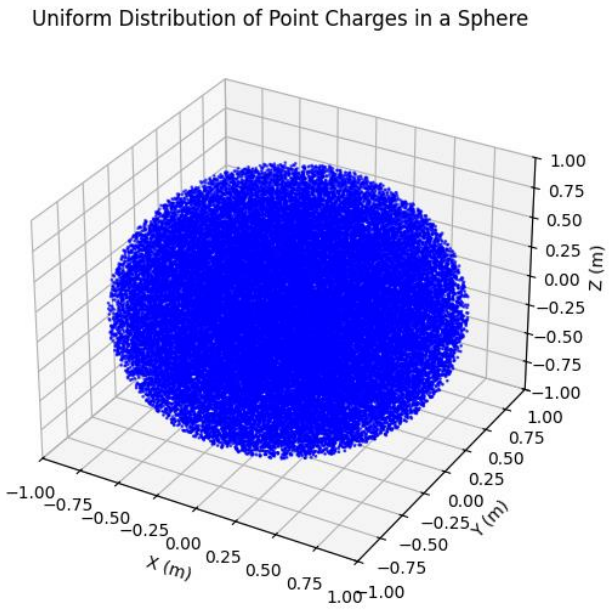

Figure 1 Part B

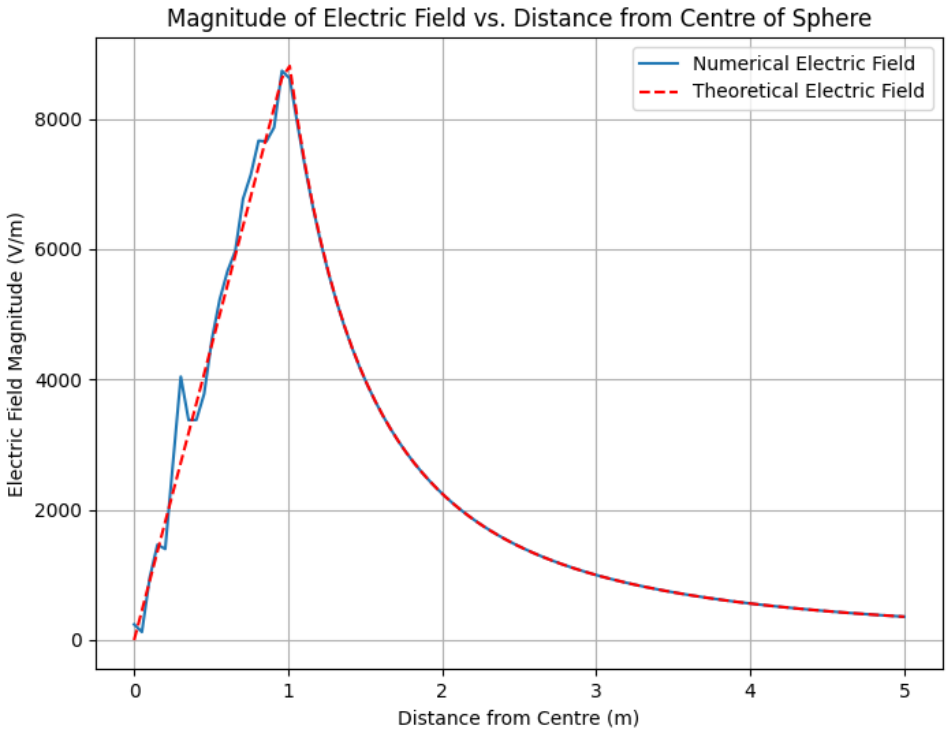

Figure 2 Part C.1

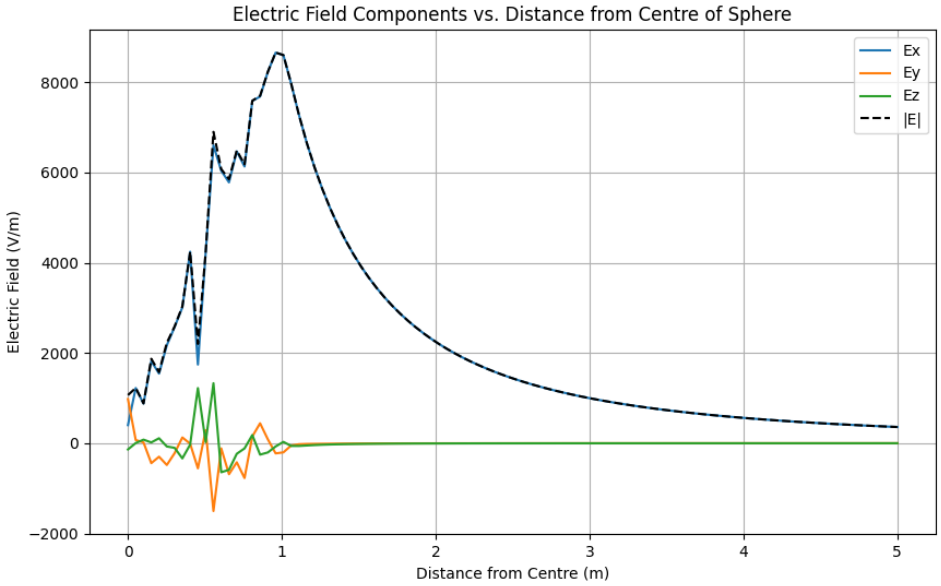

Figure 3 Part C.2

```
Part C
Iteration 1: N_total = 10000, Max Relative Error = 0.0210
Iteration 2: N_total = 20000, Max Relative Error = 0.0165
Iteration 3: N_total = 40000, Max Relative Error = 0.0103
Iteration 4: N_total = 80000, Max Relative Error = 0.0007
Desired accuracy achieved.
Part D
```

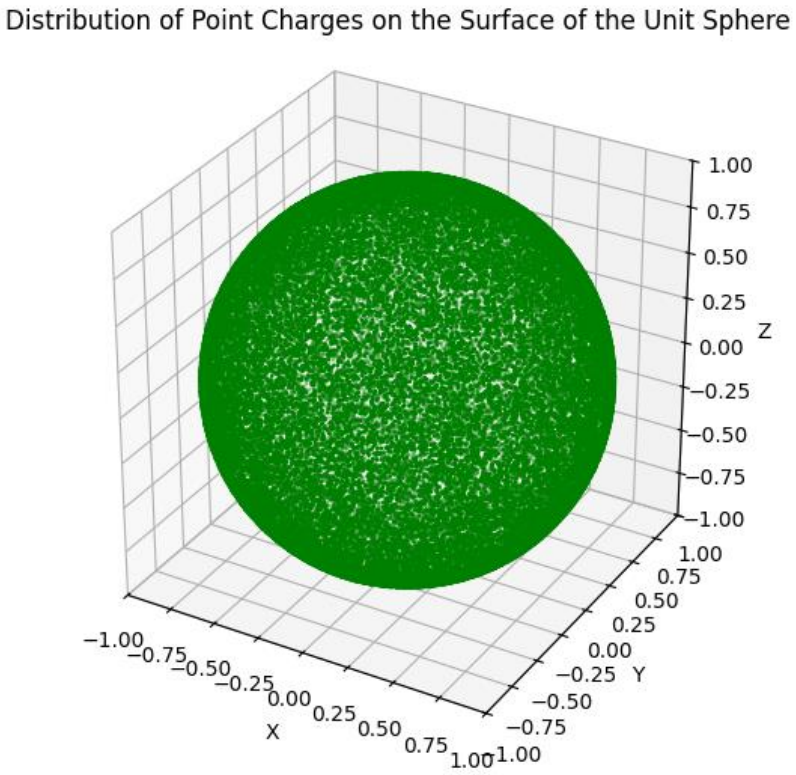

Figure 4 Part D

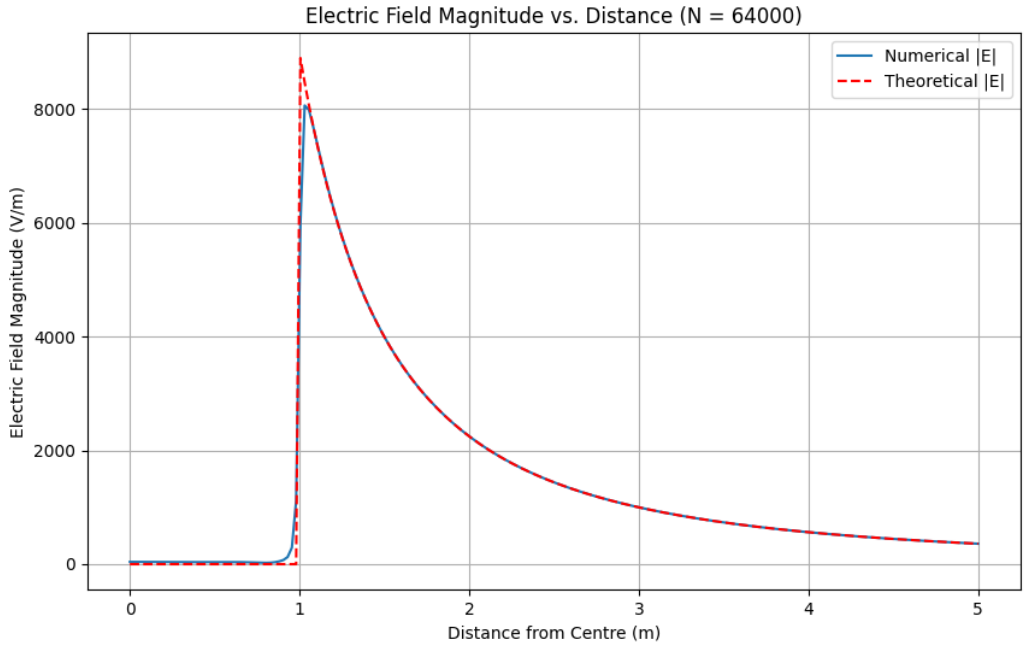

Figure 5 Part E

```
Part E
Iteration 1: N = 1000, Max Relative Error (outside) = 0.1017, Max Absolute Error (inside) = 1.1492e+03
Iteration 2: N = 2000, Max Relative Error (outside) = 0.0835, Max Absolute Error (inside) = 1.0143e+03
Iteration 3: N = 4000, Max Relative Error (outside) = 0.0200, Max Absolute Error (inside) = 7.4815e+02
Iteration 4: N = 8000, Max Relative Error (outside) = 0.0260, Max Absolute Error (inside) = 2.5572e+02
Iteration 5: N = 16000, Max Relative Error (outside) = 0.0156, Max Absolute Error (inside) = 3.5919e+02
Iteration 6: N = 32000, Max Relative Error (outside) = 0.0285, Max Absolute Error (inside) = 2.2064e+02
Iteration 7: N = 64000, Max Relative Error (outside) = 0.0017, Max Absolute Error (inside) = 6.7115e+01
Desired accuracy achieved.

Up to the nearest order of magnitude, the number of sample points required is 10^4.
Part F
Number of point charges in the hemisphere: 49568
Charge per point charge: 1.0087153001936732e-11 C

[Done] exited with code=0 in 69.544 seconds
```

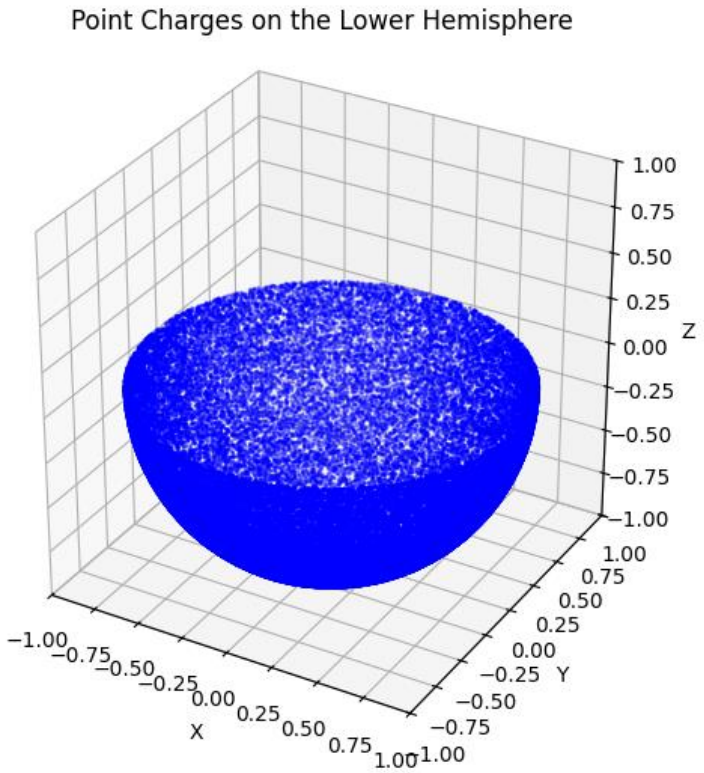

Figure 6 Part F.1

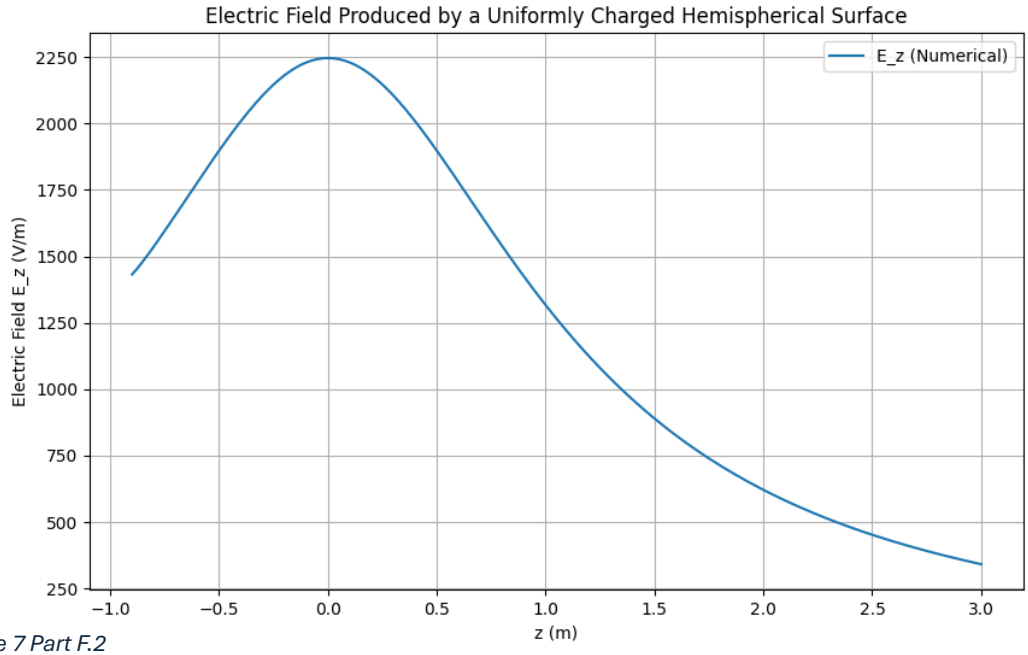

Figure 7 Part F.2