Emergence of a Stern Layer

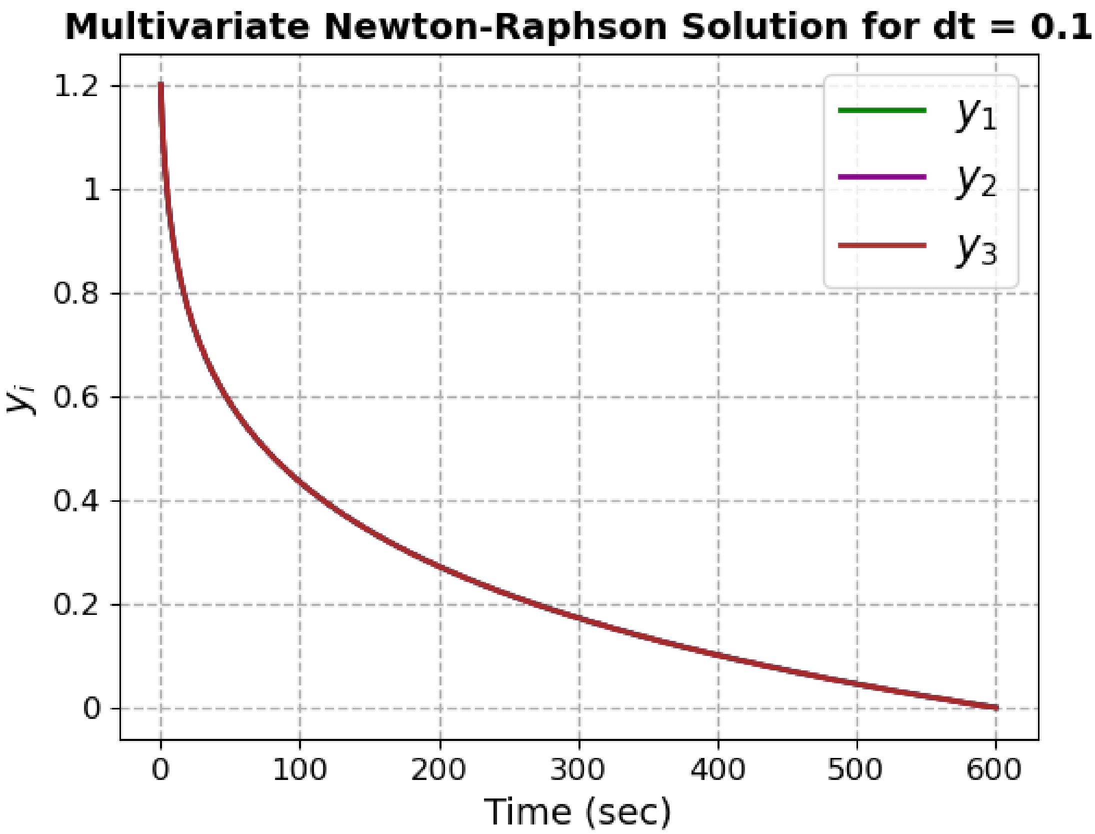
November 28, 2024

[3]:
```
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt

# Defining the root-solving functions
def f1(y1_n,y2_n,y3_n,y1_o,h):
    return y1_n + 0.04*h*y1_n - 1e4*h*y2_n*y3_n - y1_o

def f2(y1_n,y2_n,y3_n,y2_o,h):
    return y2_n - 0.04*h*y1_n + 1e4*h*y2_n*y3_n + 3e7*h*pow(y2_n,2) - y2_o

def f3(y1_n,y2_n,y3_n,y3_o,h):
    return y3_n - 3e7*h*pow(y2_n,2) - y3_o

# Calculating numerical Jacobian through spatial forward differencing
def jac(Y_n,Y_o,h):
    dy = 1e-8
    y1_n = Y_n[0]
    y2_n = Y_n[1]
    y3_n = Y_n[2]
    y1_o = Y_o[0]
    y2_o = Y_o[1]
    y3_o = Y_o[2]

    # Jacobian matrix for 3 equations and 3 unknowns
    J = np.zeros((3,3))
    J[0,0] = (f1(y1_n+dy,y2_n,y3_n,y1_o,h) - f1(y1_n,y2_n,y3_n,y1_o,h))/dy
    J[0,1] = (f1(y1_n,y2_n+dy,y3_n,y1_o,h) - f1(y1_n,y2_n,y3_n,y1_o,h))/dy
    J[0,2] = (f1(y1_n,y2_n,y3_n+dy,y1_o,h) - f1(y1_n,y2_n,y3_n,y1_o,h))/dy
    J[1,0] = (f2(y1_n+dy,y2_n,y3_n,y2_o,h) - f2(y1_n,y2_n,y3_n,y2_o,h))/dy
    J[1,1] = (f2(y1_n,y2_n+dy,y3_n,y2_o,h) - f2(y1_n,y2_n,y3_n,y2_o,h))/dy
    J[1,2] = (f2(y1_n,y2_n,y3_n+dy,y2_o,h) - f2(y1_n,y2_n,y3_n,y2_o,h))/dy
    J[2,0] = (f3(y1_n+dy,y2_n,y3_n,y3_o,h) - f3(y1_n,y2_n,y3_n,y3_o,h))/dy
    J[2,1] = (f3(y1_n,y2_n+dy,y3_n,y3_o,h) - f3(y1_n,y2_n,y3_n,y3_o,h))/dy
    J[2,2] = (f3(y1_n,y2_n,y3_n+dy,y3_o,h) - f3(y1_n,y2_n,y3_n,y3_o,h))/dy
    return J
```

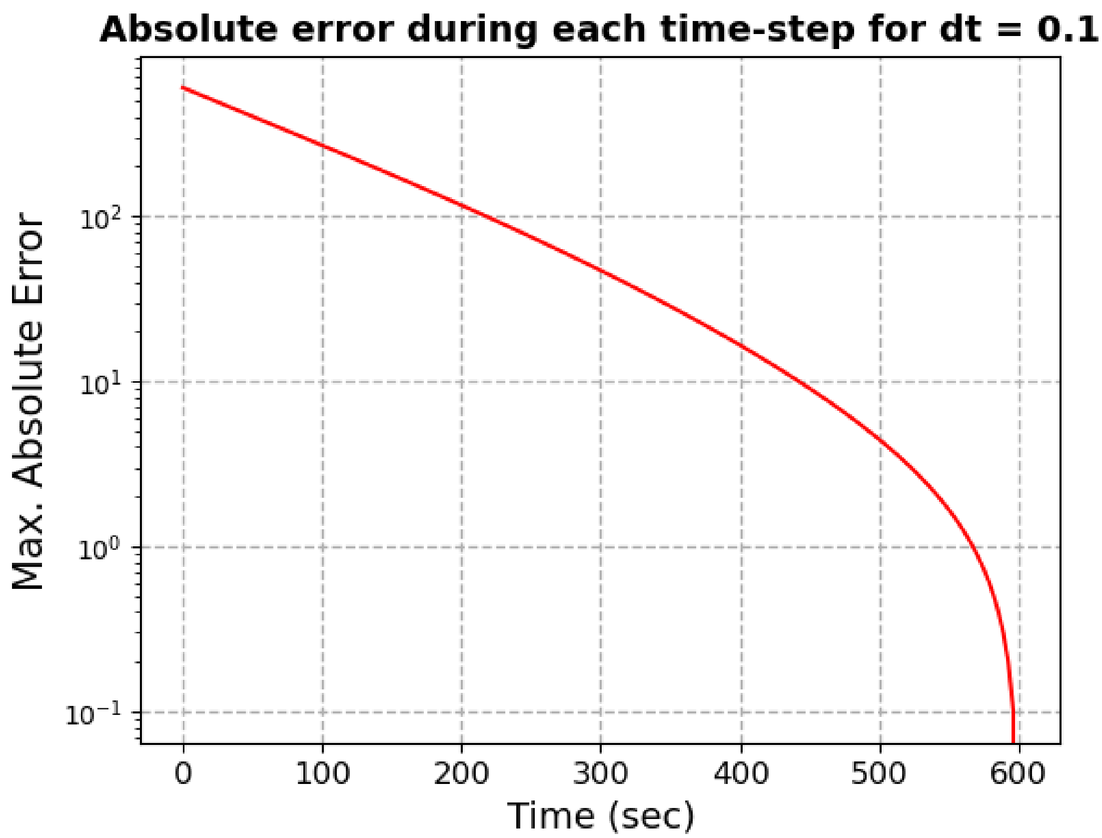
```python
# Defining the initial conditions
Y_o = np.zeros((3,1))
Y_o[0] = 1
Y_n = np.zeros((3,1))
F = np.copy(Y_o)

# Defining the Newton-Raphson solver parameters
err = 1e9
alpha = 1
tol = 1e-12
count = 0
t = np.arange(0,600.00001,step=0.1)
h = t[1] - t[0]
y1 = [1]*len(t)
y2 = [0]*len(t)
y3 = [0]*len(t)
e = [1e-6]*len(t)

# Outer time loop
for k in range(1,len(t)):
    y1_o = Y_o[0]
    y2_o = Y_o[1]
    y3_o = Y_o[2]
    Y_g = Y_o

    # Inner iterative solver loop
    while err >= tol:
        J = jac(Y_n,Y_o,h)
        y1_g = Y_g[0]
        y2_g = Y_g[1]
        y3_g = Y_g[2]
        F[0] = f1(y1_g,y2_g,y3_g,y1_o,h)
        F[1] = f2(y1_g,y2_g,y3_g,y2_o,h)
        F[2] = f3(y1_g,y2_g,y3_g,y3_o,h)
        Y_n = Y_g - alpha*np.matmul(inv(J),F)
        err = max(abs(Y_n - Y_g))

        # Updating the guess values for a new iteration
        Y_g = Y_n
        count += 1

    # Updating the new time-step values
    e[k] = err
    Y_o = Y_n
    y1[k] = Y_n[0]
    y2[k] = Y_n[1]
    y3[k] = Y_n[2]
```

```python
    log_message = 'Time = {0} sec, y1 = {1}, y2 = {2}, y3 = {3}'.format
    (round(t[k],1),round(Y_n[0][0],3),round(Y_n[1][0],3),round(Y_n[2][0],3))
    #print(log_message(round(t[k],1), round(Y_n[0][0], 3), round(Y_n[1][0], 3),␣
 ↪round(Y_n[2][0], 3)))

    # Resetting the error criteria
    count = 1
    err = 1e9
y1 = np.array([item[0] if isinstance(item, np.ndarray) else item for item in␣
 ↪y1], dtype=float)
y2 = np.array([item[0] if isinstance(item, np.ndarray) else item for item in␣
 ↪y1], dtype=float)
y3 = np.array([item[0] if isinstance(item, np.ndarray) else item for item in␣
 ↪y1], dtype=float)
e = np.array([item[0] if isinstance(item, np.ndarray) else item for item in␣
 ↪y1], dtype=float)

# Plotting the variables at each time-step
fig1, ax1 = plt.subplots()
plt.plot(t,y1,color='green',linewidth=2,label='$y_1$')
plt.plot(t,y2,color='purple',linewidth=2,label='$y_2$')
plt.plot(t,y3,color='brown',linewidth=2,label='$y_3$')
plt.grid('both',linestyle='--',linewidth=1)
xticks = [-100, 0, 100, 200, 300, 400, 500, 600, 700]
ax1.set_xticklabels(xticks,rotation=0,fontsize=12)
yticks = [-0.2, 0, 0.2, 0.4, 0.6, 0.8, 1, 1.2]
ax1.set_yticklabels(yticks,rotation=0,fontsize=12)
plt.xlabel('Time (sec)',fontsize=14)
plt.ylabel('$y_i$',fontsize=15)
plt.title('Multivariate Newton-Raphson Solution for dt =␣
 ↪'+str(h),fontsize=14,fontweight='bold')
plt.legend(fontsize=16)
plt.show()

# Plotting the converged error for each time-step
fig2, ax2 = plt.subplots()
plt.semilogy(e,t,color='red')
plt.grid('both',linestyle='--',linewidth=1)
plt.xlabel('Time (sec)',fontsize=14)
plt.ylabel('Max. Absolute Error',fontsize=15)
plt.title('Absolute error during each time-step for dt =␣
 ↪'+str(h),fontsize=14,fontweight='bold')
xticks = [-100, 0, 100, 200, 300, 400, 500, 600, 700]
ax2.set_xticklabels(xticks,rotation=0,fontsize=12)
plt.show()
```

```
C:\Users\Katha\AppData\Local\Temp\ipykernel_15860\2554164812.py:105:
UserWarning: FixedFormatter should only be used together with FixedLocator
  ax1.set_xticklabels(xticks,rotation=0,fontsize=12)
C:\Users\Katha\AppData\Local\Temp\ipykernel_15860\2554164812.py:107:
UserWarning: FixedFormatter should only be used together with FixedLocator
  ax1.set_yticklabels(yticks,rotation=0,fontsize=12)
```



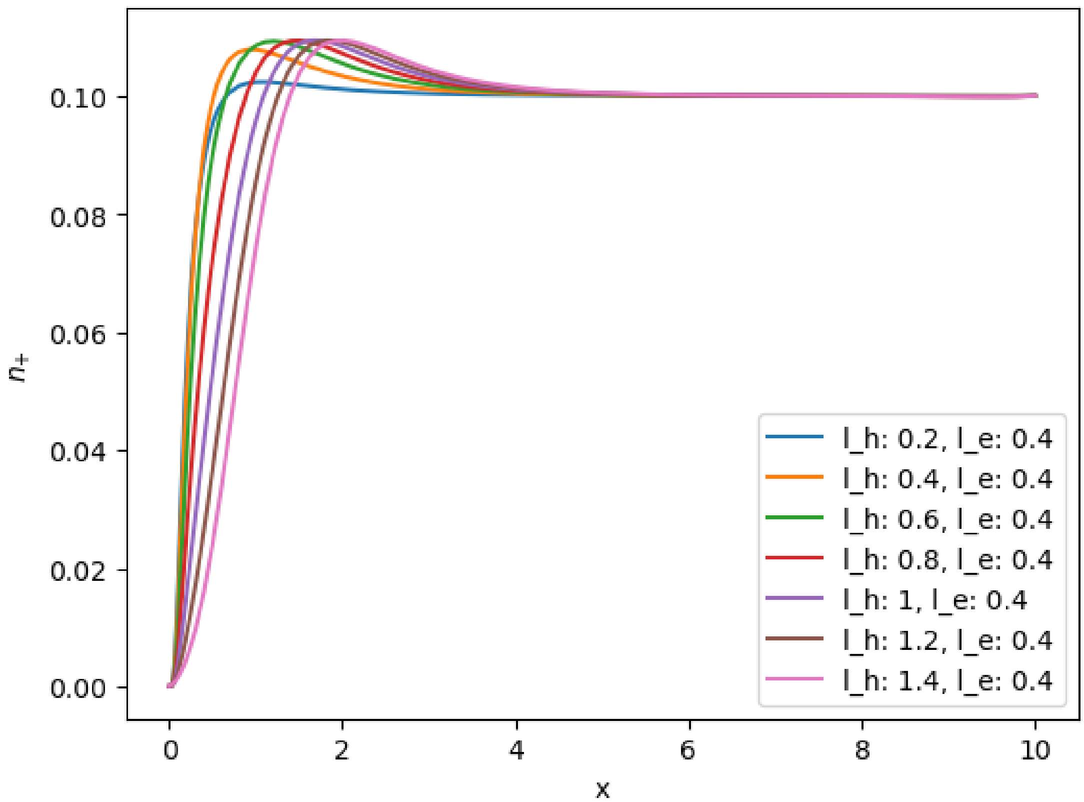

```
C:\Users\Katha\AppData\Local\Temp\ipykernel_15860\2554164812.py:122:
UserWarning: FixedFormatter should only be used together with FixedLocator
  ax2.set_xticklabels(xticks,rotation=0,fontsize=12)
```



[5]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy. integrate import solve_bvp
import warnings
warnings.filterwarnings("ignore")
```

[38]:
```
# Define the differential equations
def fun(x, y):
  k_e = 1
  k = 1/0.3
  n0 = 0.1
  #l_h = 0.4  # assuming a fixed value here for demonstration
  #l_e = 0.4  # assuming a fixed value here for demonstration

  k_h_squared = 8*np.pi*l_h*np.exp(k*l_h)*n0

  dy1_dx = y[1]
  dy2_dx = (k_e**2/2)*(np.exp(y[0]) - np.exp(-y[0]-y[2]))
  dz1_dx = y[3]
  dz2_dx = (k**2)*(y[2]) + (k_h_squared/2)*(1 - np.exp(-y[0]-y[2]))
```

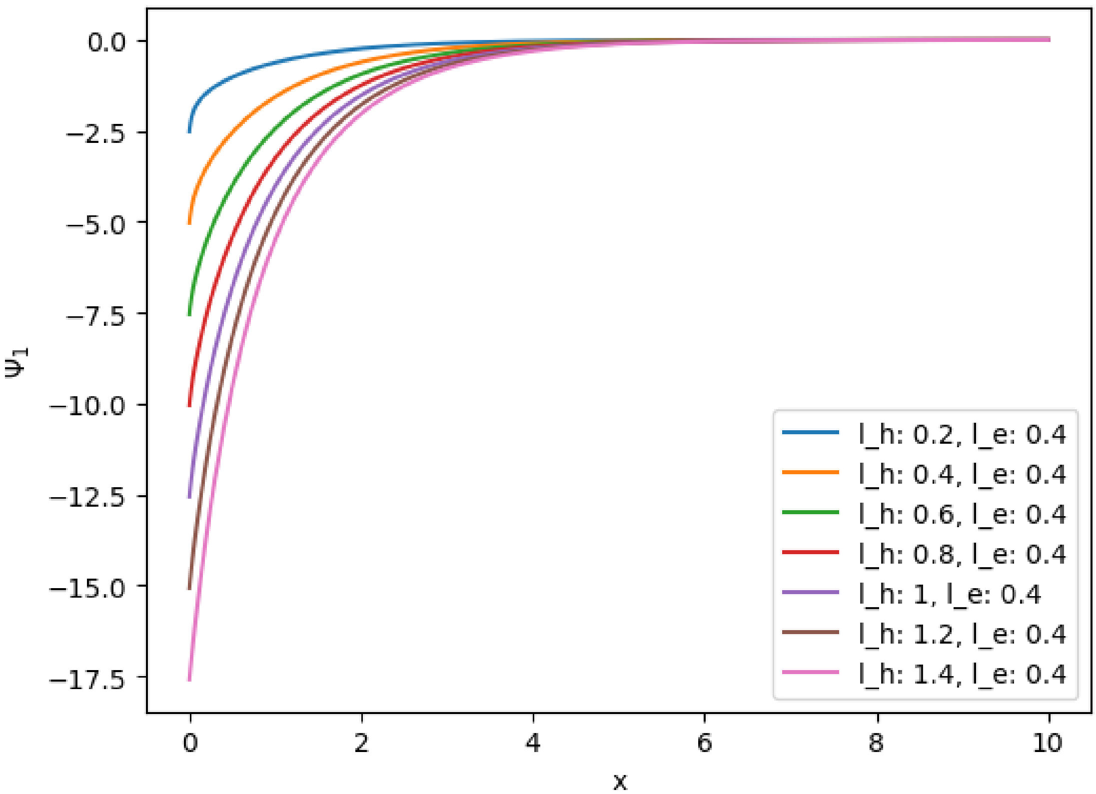
```python
  return np.vstack((dy1_dx, dy2_dx, dz1_dx, dz2_dx))

# Define function to calculate initial and boundary conditions
def get_initial_and_boundary_conditions(k_e, n_o, k, l_e, sig_e, sig_h):
  k1 = 1 + (2*k**2)/(8*np.pi*l_e*np.exp(k*l_e)*n_o)
  dy_dx_0 = -4*np.pi*l_e*sig_e
  dz_dx_0 = -4*np.pi*l_h*sig_h

  def bc(ya, yb):
    return np.array([ya[0]+dy_dx_0, yb[0], ya[1]+dz_dx_0, yb[1]])

  return bc

# Define list of l_h and l_e values
l_h_vals = [0.4, 0.4, 0.4,0.4,0.4,0.4]
l_e_vals = [0.2,0.4,0.6,0.8,1,1.2,1.4]
l_h_vals = list(set(l_h_vals))
l_e_vals = list(set(l_e_vals))
# Loop through different values and plot
for l_h in l_h_vals:
  for l_e in l_e_vals:
    k_e = 1
    n_o = 0.1
    k = 1/0.3
    sig_e = -1
    sig_h = 5

    bc = get_initial_and_boundary_conditions(k_e, n_o, k, l_e, sig_e, sig_h)
    x = np.linspace(0,10,100)
    y0 = np.zeros((4, x.size))

    sol = solve_bvp(fun, bc, x, y0)  # Removed the 'args' argument
    psi_1 = sol.y[0]
    psi_2 = sol.y[1]
    x_val = sol.x
    n_plux = n_o*np.exp(-psi_1 - psi_2)
    plt.plot(x_val, n_plux, label=f"l_h: {l_e}, l_e: {l_h}")

# Add labels and title
plt.xlabel("x")
plt.ylabel("$n_{+}$")
plt.legend()
plt.show()
```



[36]:

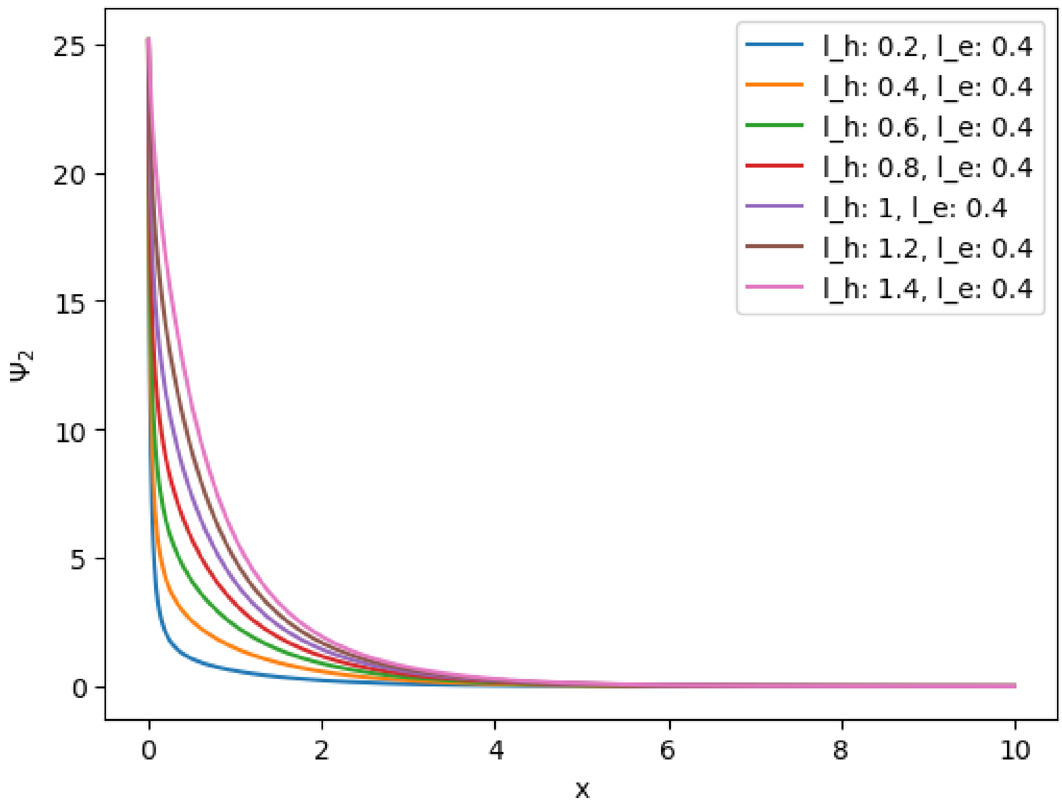
```
def fun(x, y):
  k_e = 1
  k = 1/0.3
  n0 = 0.1
  #l_h = 0.4  # assuming a fixed value here for demonstration
  #l_e = 0.4  # assuming a fixed value here for demonstration

  k_h_squared = 8*np.pi*l_h*np.exp(k*l_h)*n0

  dy1_dx = y[1]
  dy2_dx = (k_e**2/2)*(np.exp(y[0]) - np.exp(-y[0]-y[2]))
  dz1_dx = y[3]
  dz2_dx = (k**2)*(y[2]) + (k_h_squared/2)*(1 - np.exp(-y[0]-y[2]))

  return np.vstack((dy1_dx, dy2_dx, dz1_dx, dz2_dx))

# Define function to calculate initial and boundary conditions
def get_initial_and_boundary_conditions(k_e, n_o, k, l_e, sig_e, sig_h):
  k1 = 1 + (2*k**2)/(8*np.pi*l_e*np.exp(k*l_e)*n_o)
  dy_dx_0 = -4*np.pi*l_e*sig_e
  dz_dx_0 = -4*np.pi*l_h*sig_h
```

```
  def bc(ya, yb):
    return np.array([ya[0]+dy_dx_0, yb[0], ya[1]+dz_dx_0, yb[1]])

  return bc

# Define list of l_h and l_e values
l_h_vals = [0.4, 0.4, 0.4,0.4,0.4,0.4]
l_e_vals = [0.2,0.4,0.6,0.8,1,1.2,1.4]
l_h_vals = list(set(l_h_vals))
l_e_vals = list(set(l_e_vals))
# Loop through different values and plot
for l_h in l_h_vals:
  for l_e in l_e_vals:
    k_e = 1
    n_o = 0.1
    k = 1/0.3
    sig_e = -1
    sig_h = 5

    bc = get_initial_and_boundary_conditions(k_e, n_o, k, l_e, sig_e, sig_h)
    x = np.linspace(0,10,100)
    y0 = np.zeros((4, x.size))

    sol = solve_bvp(fun, bc, x, y0)  # Removed the 'args' argument
    psi_1 = sol.y[0]
    psi_2 = sol.y[1]
    x_val = sol.x
    n_plux = n_o*np.exp(-psi_1 - psi_2)
    plt.plot(x_val, psi_1, label=f"l_h: {l_e}, l_e: {l_h}")

# Add labels and title
plt.xlabel("x")
plt.ylabel("$\Psi_1$")
plt.legend()
plt.show()
```



[37]:
```
def fun(x, y):
  k_e = 1
  k = 1/0.3
  n0 = 0.1
  #l_h = 0.4  # assuming a fixed value here for demonstration
  #l_e = 0.4  # assuming a fixed value here for demonstration

  k_h_squared = 8*np.pi*l_h*np.exp(k*l_h)*n0

  dy1_dx = y[1]
  dy2_dx = (k_e**2/2)*(np.exp(y[0]) - np.exp(-y[0]-y[2]))
  dz1_dx = y[3]
  dz2_dx = (k**2)*(y[2]) + (k_h_squared/2)*(1 - np.exp(-y[0]-y[2]))

  return np.vstack((dy1_dx, dy2_dx, dz1_dx, dz2_dx))

# Define function to calculate initial and boundary conditions
def get_initial_and_boundary_conditions(k_e, n_o, k, l_e, sig_e, sig_h):
  k1 = 1 + (2*k**2)/(8*np.pi*l_e*np.exp(k*l_e)*n_o)
  dy_dx_0 = -4*np.pi*l_e*sig_e
  dz_dx_0 = -4*np.pi*l_h*sig_h
```

```python
  def bc(ya, yb):
    return np.array([ya[0]+dy_dx_0, yb[0], ya[1]+dz_dx_0, yb[1]])

  return bc

# Define list of l_h and l_e values
l_e_val = [0.4, 0.4, 0.4,0.4,0.4,0.4]
l_h_val = [0.2,0.4,0.6,0.8,1,1.2,1.4]
l_h_vals = list(set(l_e_val))
l_e_vals = list(set(l_h_val))
# Loop through different values and plot
for l_h in l_h_vals:
  for l_e in l_e_vals:
    k_e = 1
    n_o = 0.1
    k = 1/0.3
    sig_e = -1
    sig_h = 5

    bc = get_initial_and_boundary_conditions(k_e, n_o, k, l_e, sig_e, sig_h)
    x = np.linspace(0,10,100)
    y0 = np.zeros((4, x.size))

    sol = solve_bvp(fun, bc, x, y0)  # Removed the 'args' argument
    psi_1 = sol.y[0]
    psi_2 = sol.y[1]
    x_val = sol.x
    n_plux = n_o*np.exp(-psi_1 - psi_2)
    plt.plot(x_val, psi_2, label=f"l_h: {l_e}, l_e: {l_h}")

# Add labels and title
plt.xlabel("x")
plt.ylabel("$\Psi_2$")
plt.legend()
plt.show()
```


l_h: 0.2, l_e: 0.4
l_h: 0.4, l_e: 0.4
l_h: 0.6, l_e: 0.4
l_h: 0.8, l_e: 0.4
l_h: 1, l_e: 0.4
l_h: 1.2, l_e: 0.4
l_h: 1.4, l_e: 0.4
Ψ_2
x
0
2
4
6
8
10
5
15
20
25

[ ]: