Project 1 (ME490)

Ariana Quek

1 Question 1

Figure 1 shows the evolution of mass with time; where mass is defined as

$$M(t) = \int_0^L c(x,t)dx \tag{1}$$

where $x \in [0, L]$ and $t \in [0, T]$. The slight deviation of the mean from the initial mass is attributed to the approximation error resulting from the use of the trapezoidal rule. The spatial step size, denoted as d_x, is set to 0.1 (as shown in the Python code in the Appendix) to ensure a more accurate representation of the integral. The accuracy of the trapezoidal rule approximation increases as the partition size, d_x, decreases. The mass is expected to remain constant throughout the iteration due to zero flux at the boundaries, which implies no outflow or inflow of mass at either end. As d_x approaches zero, $M(T)$ approaches $M(0)$, where T represents the final time step in a staggered scheme.

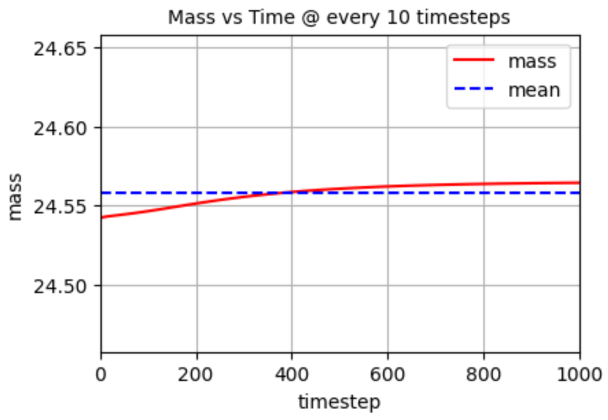


Figure 1: Total mass of contaminant in the domain as a function of time.

2 Question 2

2.1 Derivation of Finite Difference (FD) expression

The forward difference method were used to enforce the boundary condition below.

$$D\frac{\partial c}{\partial x} - c\nu = 0 \tag{2}$$

Expressing it using forward difference:

$$D\frac{C_{I+1}^n - C_I^n}{\triangle x} - C_I^n \nu = 0 \tag{3}$$

The coefficients for the A matrix at the boundary are expressed as $\frac{D}{\Delta x} - \nu$ and $\frac{D}{\Delta x}$. This is expression is used for spatial nodes corresponding to $x = 0$ and $x = L$.

For nodes not at the boundary, for points $(x_I,\, t^n)$, using the backward difference method results in the expression:

$$\frac{C_I^n - C_I^{n-1}}{\triangle t} = D\frac{C_{I+1}^n - 2C_I^n + C_{I-1}^n}{\triangle x^2} - \nu\frac{C_{I+1}^n - C_I^n}{\triangle x} \tag{4}$$

Rearranging the equation with t^n on the LHS and t^{n-1} on the RHS gives

$$-\sigma C_{I-1}^n + (2\sigma - \lambda + 1)C_I^n + (\lambda - \sigma)C_{I+1}^n = C_I^{n-1} \tag{5}$$

where $\sigma = D\frac{\triangle t}{\triangle x^2}$ and $\lambda = \nu\frac{\triangle t}{\triangle x}$.

2.2 Mass Conservation for velocity, $\nu = 10$ and $\nu = 100$

For configuration where $\nu = 10$, the spatial discretization with $d_x = 1.0$ resulted in a relative error of mass difference equal to 0.919%. The error is defined as:

$$Err(\%) = \frac{abs(M(T) - M(0))}{M(0)} * 100 \tag{6}$$

where $M(t)$ is calculated using Eq. (1).

However, for configurations where $\nu = 100$, it struggles to achieve a relative error of less than 2%. Figure 2 illustrates the relative error plot vs. dx. A value of $dx = 0.0333$ is needed to achieve an error of 1.862%.

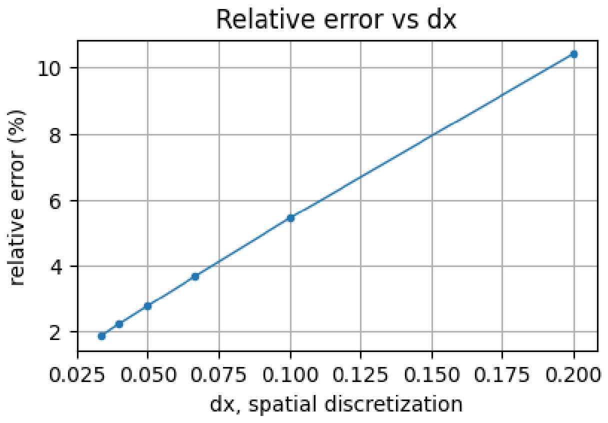


Figure 2: Plot of relative error against dx for $\nu = 100$.

Figure 3 illustrates the evolution of mass at every 25 timesteps for both $\nu = 10$ and $\nu = 100$. With the added velocity term in the PDE equation, air moves from left to right at a given velocity magnitude of ν. As shown in Figure 3, the evolution of concentration over time differs for the two velocity magnitudes. At the higher velocity magnitude, concentration diffuses more on the left end and exhibits a sharp increase on the right end. In contrast, with the lower velocity, there isn't as sharp of a gradient (change in concentration) on the right end. This difference explains why a higher order of spatial discretization is needed to obtain a good approximation using the trapezoidal rule. Increasing the number of elements ($d_x \to 0$) is necessary to achieve a better approximation due to the sharp gradient (higher concentration of contaminant) at the right boundary.

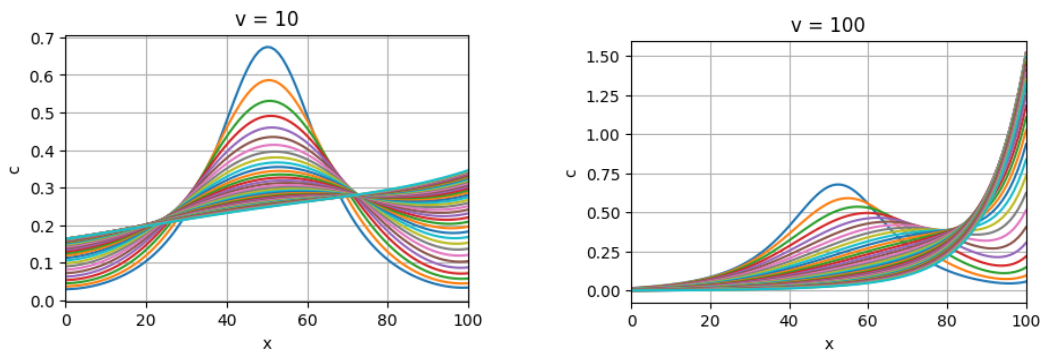


Figure 3: Evolution of concentration at every 25 timestep.

3 Question 3

3.1 Jacobian with Finite Difference

Figure 4 compares the error in computing the Jacobian matrix using automatic differentiation (AD) with the TensorFlow package and finite difference (FD) schemes. The error decreases up to a certain value of ε_{opt}

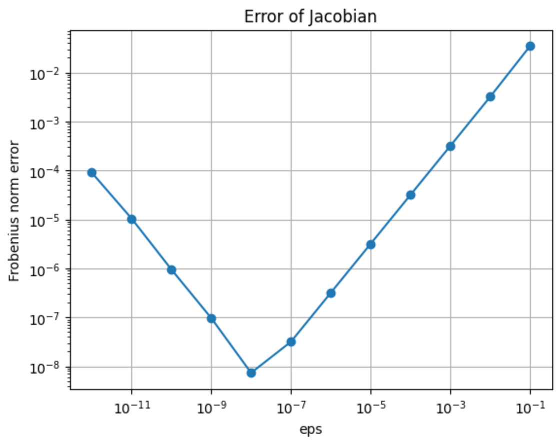


Figure 4: Plot of error vs $\log(\varepsilon)$

and then increases. This loss of accuracy occurs when the step size, ε, becomes very small ($\varepsilon \leq \varepsilon_{\text{opt}} \to 0$). This is due to the loss of significant digits resulting from round-off errors, which are further compounded by dividing by a small number when ε approaches $\varepsilon_{\text{mach}}$. The numerical value we actually compute is given by:

$$\bar{J} = J_{FD}(1 + K(\varepsilon) \cdot \varepsilon_{\text{mach}}),$$

where

$$J_{FD} = \frac{f(x+\varepsilon h) - f(x)}{\varepsilon h} \quad ,$$

and

$$K(\varepsilon) = \frac{\max\{|f(x+\varepsilon h)|, |f(x)|\}}{|f(x+\varepsilon h) - f(x)|}$$

Here, J_{FD} represents the finite difference approximation, and $K(\varepsilon)$ is the condition number for computing $|f(x+\varepsilon h) - f(x)|$. When performing computations using the finite differences method of order m, especially in the presence of round-off errors, there is an optimal node spacing, where $\varepsilon \approx \varepsilon_{\text{mach}}^{1/(m+1)}$.

3.2 Newton Raphson's Method

The Newton's algorithm is employed to compute the Jacobian matrix and solve the catenary problem. Figure 5 presents two subplots, showing the initial and final configurations of the catenary after 8 iterations, with an error convergence of 1×10^{-12} for both methods. Relative error is utilized as a metric to assess the performance of both methods (refer to Eqn. (7)).

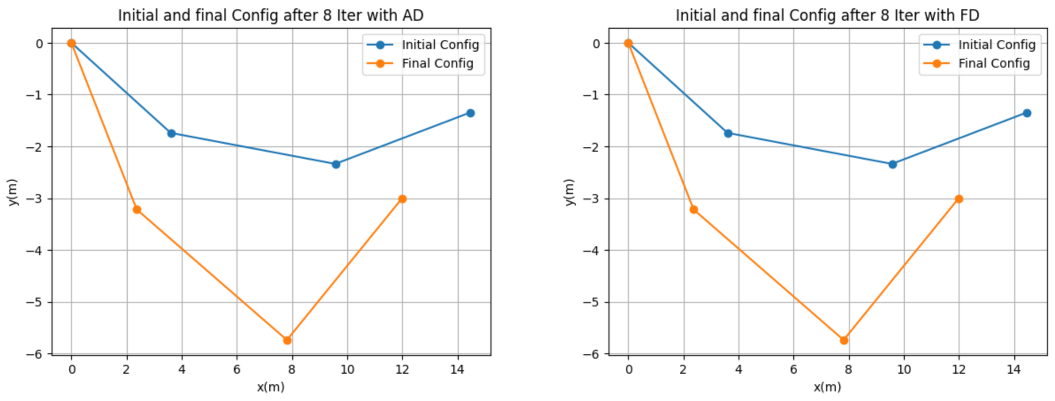


Figure 5: Evolution of concentration at every 25 timestep.

$$\text{Rel. Err} = \frac{|F(x_c)|}{|F(x_0)|} \tag{7}$$

Here, x_c represents the current guess of the solution, and x_0 is the initial guess.

The maximum number of iterations was set to 100 for both methods, but convergence was achieved with an error of less than 1×10^{-12} after only 8 iterations. On average, the runtime required to compute a 4×4 Jacobian matrix using automatic differentiation (AD) and finite difference (FD) methods is 0.670 and 0.031, respectively.

4 Appendix

Code to generate results are attached below.

```
In [ ]: import matplotlib.pyplot as plt
        import math
        import numpy as np
        from numpy import arange, array
        from numpy.linalg import inv,solve
In [ ]: def initial_func(x,c, L, decayL):
          for i in range(0,len(x)):
             dis = abs(x[i] - L/2)
             c[i] = math.exp(-dis/decayL)

          return c
In [ ]: L = 100.0
        N = 1000

        #diffusion constant
        D = 1000

        #velocity
        v = 100.0

        decayL = L/8

        #time steps
        n = 1000
        Tfinal = 1.0
        dt = Tfinal/n

        dx = L/N
In [ ]: x = arange(0.0,L+dx,dx)
        c0 = np.zeros_like(x)

        #initialize right side
        b = np.zeros_like(x)
        initial_func(x,c0,L,decayL)
```

```
Out[ ]: array([0.01831564, 0.01846275, 0.01861105, ..., 0.01861105, 0.01846275,
               0.01831564])
```

```
In [ ]: s = (N+1,N+1)
        Amat = np.zeros(s)

        #set up first and last rows
        Amat[0,0] = -1/dx
        Amat[0,1] = 1/dx

        Amat[N,N-1] = -1/dx
        Amat[N,N] = 1/dx

        #diffusive piece
        for i in range (1,len(Amat)-1):
           Amat[i,i] = 1.0 + 2*D*dt/(dx*dx)
           Amat[i,i-1] = -D*dt/(dx*dx)
           Amat[i,i+1] = -D*dt/(dx*dx)
In [ ]: # we want to write a function to calculate the mass of the contaminant in the domain
        Loading [MathJax]/extensions/Safe.js oncentration over the domain
```

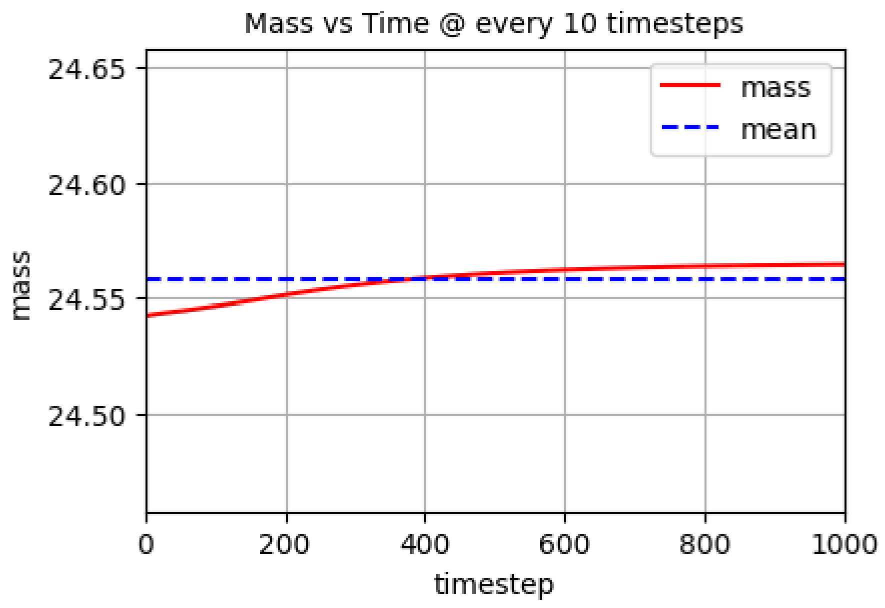
```
def mass(c,dx):
    """
    Computes mass at a particular time step
    """

    # integrate the concentration over the domain
    mass = 0
    for i in range (1,len(c)-1):
        mass = mass + c[i]*dx

    mass = mass + c[0]*dx/2 + c[len(c)-1]*dx/2

    return mass
```

In [ ]:
```
#initialize b
for i in range (1,len(c0)-1):
   b[i] = c0[i]

#initialize c
c = np.zeros_like(x)

t = 0
tsteps = np.zeros(n+1)

mass_ss_list = [mass(c0,dx)]
# print("t = ", 0, " ", "mass = ", mass(c0,dx,L))

for k in range (1,n+1):
    t = t + dt
    tsteps[k] = t
    c = solve(Amat,b)

    # computes mass at every 100 time steps
    if k%10 == 0:
        mass_ss_list.append(mass(c,dx))
        # print("t = ", round(t, 3), "mass = ", mass(c,dx,L))
    for i in range (1,len(c)-1):
       b[i] = c[i]
```

In [ ]:
```
# plot a graph of mass vs time and take also plot the mean
mass_mean = np.mean(mass_ss_list)
print("mean mass = ", mass_mean)
print("mass at t = 0 = ", mass_ss_list[0])
print("max deviation from mean = ", max(abs(mass_ss_list - mass_mean)))

iter = np.arange(0,n+1,10)
# print(iter)

fig = plt.figure(figsize=(4.5,3))
plt.plot(iter, mass_ss_list, 'r-', label='mass')
plt.plot(iter, mass_mean*np.ones(len(iter)), 'b--', label='mean')
plt.grid()
plt.xlabel('timestep')
plt.ylabel('mass')
plt.legend()
plt.xlim(0,1000)
```

```
    plt.ylim(mass_mean-0.1,mass_mean+0.1)
    plt.title('Mass vs Time @ every 10 timesteps', fontsize=10)
```

```
mean mass =  24.557831423322668
mass at t = 0 =  24.542239918890164
max deviation from mean =  0.015591504432503456
Out[ ]:Text(0.5, 1.0, 'Mass vs Time @ every 10 timesteps')
```



In [ ]:
```
import matplotlib.pyplot as plt
import math
import numpy as np
from numpy import arange, array
from numpy.linalg import inv,solve
```
In [ ]:
```
def initial_func(x,c, L, decayL):
  for i in range(0,len(x)):
     dis = abs(x[i] - L/2)
     c[i] = math.exp(-dis/decayL)

  return c
```
In [ ]:
```
def mass(c,dx):
    """
    Computes mass at a particular time step
    """

    # integrate the concentration over the domain
    mass = 0
    for i in range (1,len(c)-1):
        mass = mass + c[i]*dx

    mass = mass + c[0]*dx/2 + c[len(c)-1]*dx/2

    return mass
```
In [ ]:
```
def solve_c_mass(v, dx, dt, N, D, L, n):
    # define auxiliary variables
    sigma = D*dt/(dx*dx)
    lmda = v*dt/dx

    x = arange(0.0,L+dx,dx)
    c0 = np.zeros_like(x)

    #initialize right side
    b = np.zeros_like(x)
    decayL = L/8
    initial_func(x,c0,L,decayL)

    s = (N+1,N+1)
    Amat = np.zeros(s)

    #set up first and last rows
    Amat[0,0] = -D/dx - v
    Amat[0,1] = D/dx

    Amat[N,N-1] = -D/dx - v
    Amat[N,N] = D/dx

    #diffusive piece
    for i in range (1,len(Amat)-1):
        Amat[i,i] = 2*sigma - lmda + 1
        Amat[i,i-1] = -sigma
        Amat[i,i+1] = -sigma + lmda
```
Loading [MathJax]/extensions/Safe.js

```
        #initialize b
        for i in range (1,len(c0)-1):
            b[i] = c0[i]

        #initialize c
        c = np.zeros_like(x)

        t = 0
        tsteps = np.zeros(n+1)

        #store initial mass
        initial_m = mass(c0,dx)

        for k in range (1,n+1):
            t = t + dt
            tsteps[k] = t
            c = solve(Amat,b)

            for i in range (1,len(c)-1):
                b[i] = c[i]

        final_m = mass(c,dx)

        return initial_m, final_m
```

In [ ]:

```
#diffusion constant
D = 1000

#time steps
n = 1000
Tfinal = 1.0
dt = Tfinal/n

# domain
L = 100.0

### change parameters here
v = 10
N_list = [100]

for N in N_list:
    dx = L/N
    print("dx = ", dx)
    print(initial_m)
    print(final_m)
    initial_m, final_m = solve_c_mass(v, dx, dt, N, D, L, n)
    print("rel error (%) = ", abs(final_m - initial_m)/initial_m * 100, "%")
    print("")
```

```
dx =  1.0
24.542239918890164
23.21003250376308
rel error (%) =  0.91919324231313 %
```

In [ ]:

```
#diffusion constant
```

```
D = 1000

#time steps
n = 1000
Tfinal = 1.0
dt = Tfinal/n

# domain
L = 100.0

### change parameters here
v = 100
N_list = [500, 1000, 1500, 2000, 2500, 3000, 3500, 4000]
rel_err_list = []

for N in N_list:
    dx = L/N
    print("dx = ", dx)
    # print(initial_m)
    # print(final_m)
    initial_m, final_m = solve_c_mass(v, dx, dt, N, D, L, n)
    rel_err = abs(final_m - initial_m)/initial_m * 100
    rel_err_list.append(rel_err)
        # if error is less than 2 percent, break
    if rel_err < 2:
        break
    print("rel error (%) = ", rel_err, "%")
    print("")
```

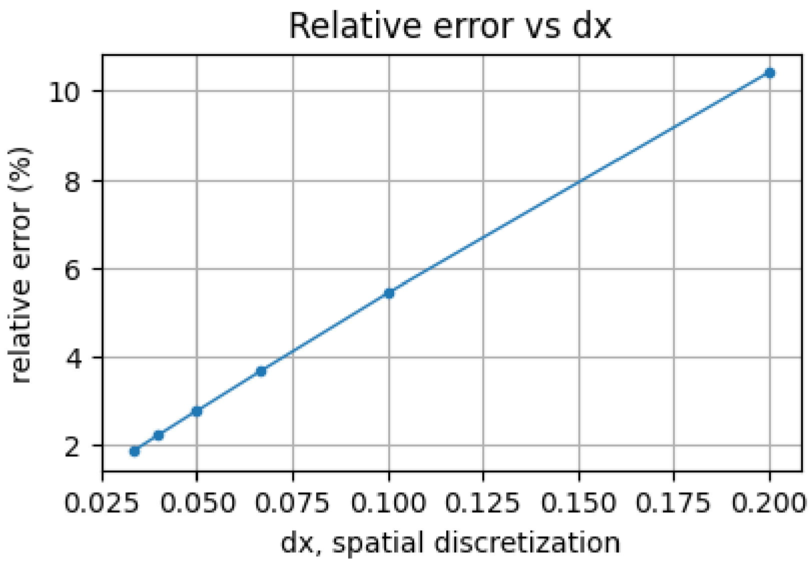
```
dx =  0.2
rel error (%) =  10.413372308091686 %

dx =  0.1
rel error (%) =  5.428222605311926 %

dx =  0.06666666666666667
rel error (%) =  3.6704198340590155 %

dx =  0.05
rel error (%) =  2.77251662984471 %

dx =  0.04
rel error (%) =  2.227561459260509 %

dx =  0.03333333333333333
```

In [ ]:
```
print("N_list = ", N_list[:len(rel_err_list)])
dx_list = [L/N for N in N_list[:len(rel_err_list)]]
print("dx_list = ", dx_list)
print("rel_err_list = ", rel_err_list)
```

```
N_list =  [500, 1000, 1500, 2000, 2500, 3000]
dx_list =  [0.2, 0.1, 0.06666666666666667, 0.05, 0.04, 0.03333333333333333]
rel_err_list =  [10.413372308091686, 5.428222605311926, 3.6704198340590155, 2.77251662984471, 2.227561459260509,
1.861637785934784]
```

In [ ]:
```
# plot a graph of the relative error vs dx
```

```
plt.figure(figsize = (4.5, 2.7))
plt.plot(dx_list, rel_err_list, 'o-', markersize = 3, linewidth = 1)
plt.xlabel("dx, spatial discretization")
plt.ylabel("relative error (%)")
plt.title("Relative error vs dx")
plt.grid(True)
```

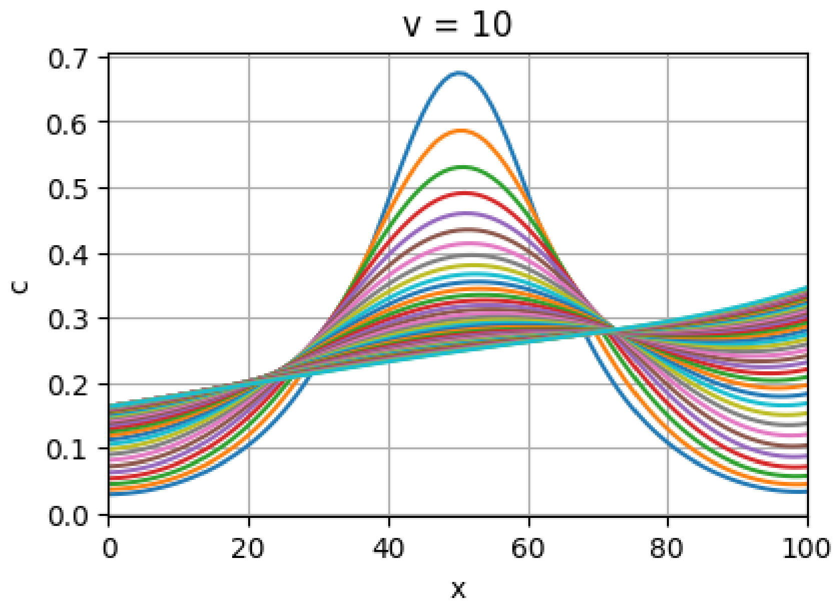


In [ ]:
```
def solve_c_mass(v, dx, dt, N, D, L, n):
    # define auxiliary variables
    sigma = D*dt/(dx*dx)
    lmda = v*dt/dx

    x = arange(0.0,L+dx,dx)
    c0 = np.zeros_like(x)

    #initialize right side
    b = np.zeros_like(x)
    decayL = L/8
    initial_func(x,c0,L,decayL)

    s = (N+1,N+1)
    Amat = np.zeros(s)

    #set up first and last rows
    Amat[0,0] = -D/dx - v
    Amat[0,1] = D/dx

    Amat[N,N-1] = -D/dx - v
    Amat[N,N] = D/dx

    #diffusive piece
    for i in range (1,len(Amat)-1):
        Amat[i,i] = 2*sigma - lmda + 1
        Amat[i,i-1] = -sigma
        Amat[i,i+1] = -sigma + lmda

    #initialize b
    for i in range (1,len(c0)-1):
        b[i] = c0[i]

    #initialize c
    c = np.zeros_like(x)

    t = 0
    tsteps = np.zeros(n+1)

    #store initial mass
    initial_m = mass(c0,dx)

    for k in range (1,n+1):
        t = t + dt
        tsteps[k] = t
        c = solve(Amat,b)

        # for plotting
        if (k % 25) == 0:
            # plt.clf()
            plt.plot(x,c)
            # plt.suptitle("Time = %1.3f" % t)
        plt.grid(True)
        plt.xlabel("x")
        plt.ylabel("c")
        plt.xlim(0,100)
        # legend
        # plt.legend(loc = "upper right")

        for i in range (1,len(c)-1):
            b[i] = c[i]

    final_m = mass(c,dx)

    return initial_m, final_m
```

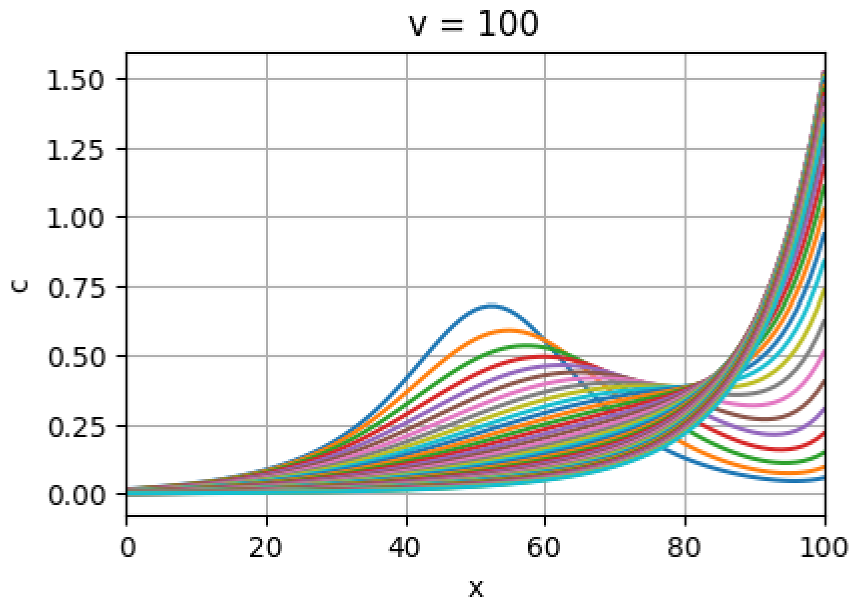
In [ ]:
```
# we want to plot out to show the difference between the two cases

#diffusion constant
D = 1000

#time steps
n = 1000
Tfinal = 1.0
dt = Tfinal/n

# domain
L = 100.0

### change parameters here
v = 10
N = 100
dx = L/N
print("dx = ", dx)
plt.figure(figsize = (4.5, 3))
plt.title("v = 10")
```

```
initial_m, final_m = solve_c_mass(v, dx, dt, N, D, L, n)
plt.show()

### change parameters here
v = 100
N = 100
dx = L/N
print("dx = ", dx)
plt.figure(figsize = (4.5, 3))
plt.title("v = 100")
initial_m, final_m = solve_c_mass(v, dx, dt, N, D, L, n)
plt.show()
```

```
dx =  1.0
```



```
dx =  1.0
```



```
In [ ]: #code for discrete catenary
        import numpy as np
        from numpy.linalg import norm
        import matplotlib.pyplot as plt
        import tensorflow as tf
        import time
In [ ]: # definition of variables and initial guess
        P = tf.Variable([0,-16, -20], dtype=tf.float64)
        ell = tf.Variable([4.0,6.0,5.0], dtype=tf.float64)
        a = tf.Variable(6.0, dtype=tf.float64)
        d_vertical = tf.Variable(3, dtype=tf.float64)

        # x = tf.Variable([50, 50*np.pi/180, 50*np.pi/180/2, -50*np.pi/180], dtype=tf.float64)
        x = tf.Variable([15, 0.45, 0.1,-0.2], dtype=tf.float64)
In [ ]: def Feval(x, ell, a, P, d_vertical):
            N = x.shape[0]-1  #length of T and theta
            T = x[0]
            theta = x[1:]

            F = [tf.Variable(0.0, dtype=tf.float64) for _ in range(N+1)]

            for i in range(N-1):
                F[i] = F[i] + T*(tf.tan(theta[i])-tf.tan(theta[i+1])) + P[i+1]
                F[N-1] = F[N-1] + ell[i]*tf.cos(theta[i])
                F[N]  =  F[N] + ell[i]*tf.sin(-theta[i])

            F[N-1] = F[N-1] + ell[N-1]*tf.cos(theta[N-1]) - 2*a
            F[N] = F[N] + ell[N-1]*tf.sin(-theta[N-1]) + d_vertical

            # Convert the list of tensors back into a single tensor
            F = tf.stack(F)

            return F
In [ ]: # calculate the jacobian matrix with autodiff
        with tf.GradientTape() as tape:
            # forward pass
            F = Feval(x, ell, a, P, d_vertical)
            print("F is: \n", F)
        # get the gradient of F with respect to x
        J = tape.jacobian(F, x)
        print("J is: \n", J)
```

```
F is:
 tf.Tensor([-10.2591941  -15.45432939   2.47214629   1.65448402], shape=(4,), dtype=float64)
J is:
 tf.Tensor(
[[  0.38272039  18.50013295 -15.1510057    0.        ]
 [  0.30304471   0.          15.1510057  -15.61637038]
 [  0.          -1.73986214  -0.5990005    0.99334665]
 [  0.          -3.60178841  -5.97002499  -4.90033289]], shape=(4, 4), dtype=float64)
```

```
In [ ]: # function for FD
        def J_FD(x, eps, h, ell=ell, a=a, P=P, d_vertical=d_vertical):
            """
Loading [MathJax]/extensions/Safe.js hich to evaluate the Jacobian
```

```
        eps: perturbation to x (step size)
        h: arbitrary vector
        """
        # have everything in float64
        x = x.numpy().astype(np.float64)
        eps = np.float64(eps)
        h = h.astype(np.float64)
        ell = ell.numpy().astype(np.float64)
        a = a.numpy().astype(np.float64)
        P = P.numpy().astype(np.float64)
        d_vertical = d_vertical.numpy().astype(np.float64)

        N = x.shape[0]-1
        J = np.zeros((N+1, N+1), dtype=np.float64)
        for i in range(N+1):
            x_pert = x + eps*h[i]
            F_pert = Feval(x_pert, ell, a, P, d_vertical)
            F = Feval(x, ell, a, P, d_vertical)
            J[:,i] = (F_pert - F)/eps
        return J
```

In [ ]:
```
print("J_FD is: \n", J_FD(x, 1e-6, np.eye(4)))
```

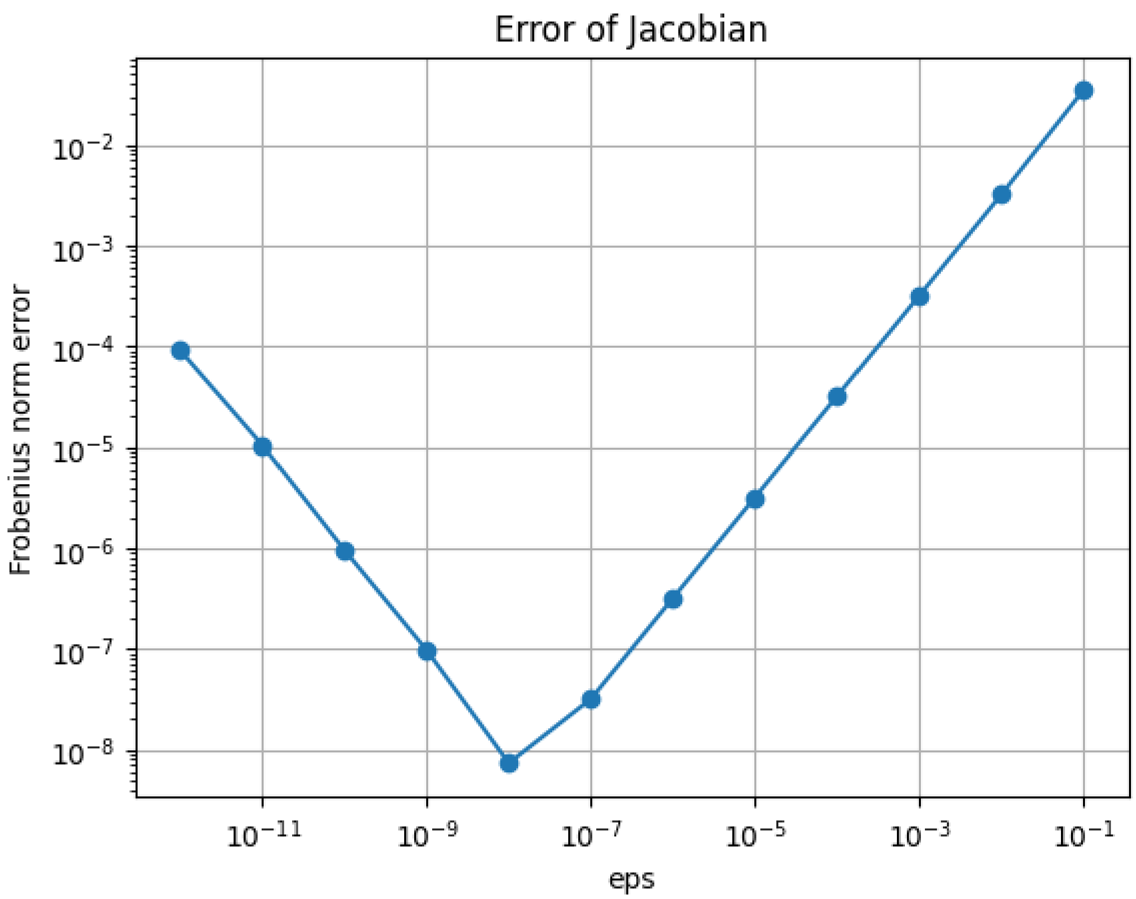
```
J_FD is:
 [[ 0.38272039  18.50014188 -15.15100722   0.        ]
 [  0.30304471   0.          15.15100722 -15.61636721]
 [  0.          -1.73986394  -0.59900349   0.9933442 ]
 [  0.          -3.60178754  -5.97002469  -4.90033339]]
```

In [ ]:
```
# eps from 1e-1 to 1e-12
eps_array = np.logspace(-1, -12, 12)
print("eps_array is: \n", eps_array)

err_list = []

for i in range(len(eps_array)):
    eps = eps_array[i]
    J_FD_ = J_FD(x, eps, np.eye(4))
    err = norm(J - J_FD_, ord ="fro") / norm(J, ord="fro")
    err_list.append(err)
```

```
eps_array is:
 [1.e-01 1.e-02 1.e-03 1.e-04 1.e-05 1.e-06 1.e-07 1.e-08 1.e-09 1.e-10
 1.e-11 1.e-12]
```

In [ ]:
```
print("Error_list is: \n", err_list)
# plot error
plt.loglog(eps_array, err_list, '-o')
plt.xlabel('eps')
plt.ylabel('Frobenius norm error')
plt.title('Error of Jacobian')
plt.grid(True)
plt.show()
```

```
Error_list is:
 [0.03479743016187109, 0.0032093235141138696, 0.00031860039924384566, 3.183708100522245e-05, 3.183477659005
9475e-06, 3.183333248694745e-07, 3.178202281101829e-08, 7.3521392735226955e-09, 9.676545842943959e-08, 9.5087
25308532651e-07, 1.0469595460920237e-05, 9.078729988239666e-05]
```

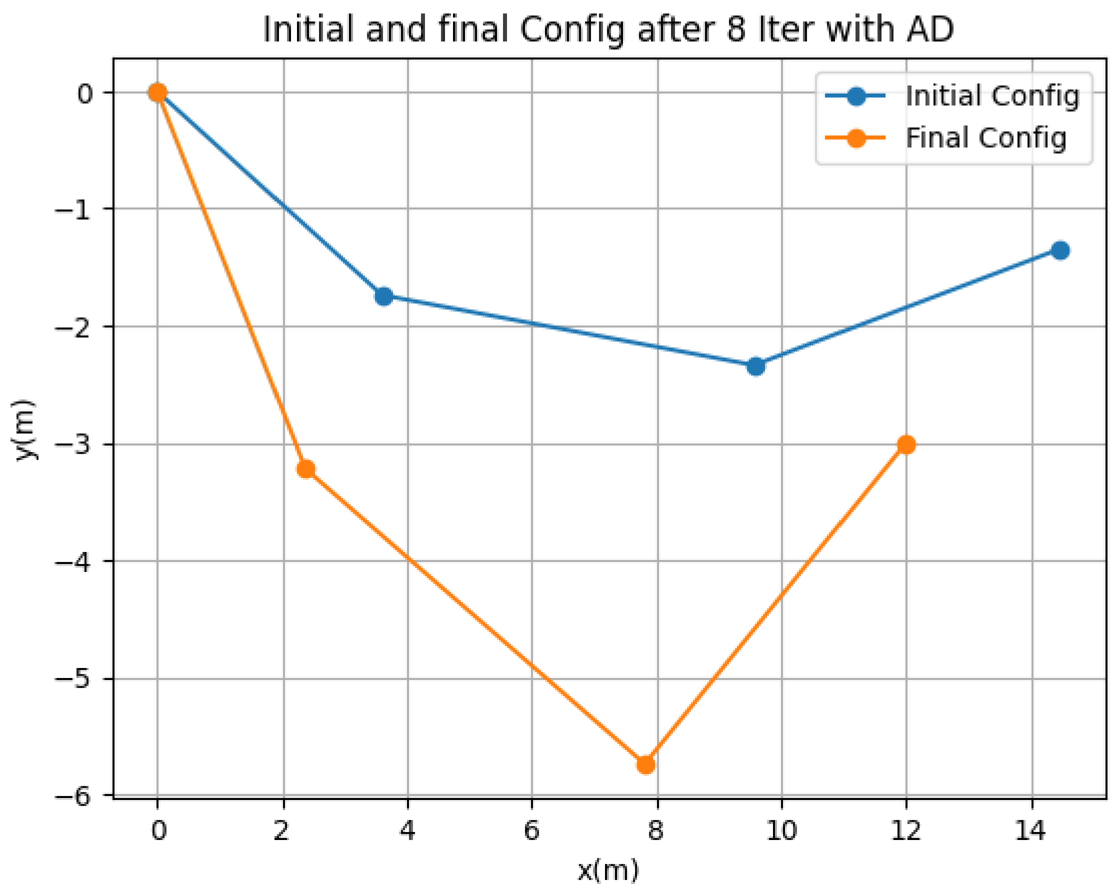


In [ ]:
```
def PlotConfig_up(theta_array, ell_array, ax, label):
    """
    theta_array: array of angles (which corresponds to N segments); N-1 nodes
    """
    # plot the configuration
    x = np.zeros(theta_array.shape[0]+1)
    y = np.zeros(theta_array.shape[0]+1)

    x[0] = 0
    y[0] = 0

    for i in range(theta_array.shape[0]):
        x[i+1] = x[i] + ell_array[i]*np.cos(theta_array[i])
        y[i+1] = y[i] - ell_array[i]*np.sin(theta_array[i])

    # ax.plot(x,y, x, y, 'o-', label=label)
    ax.plot(x,y, 'o-', label=label)
    return
```
In [ ]:
```
# AD Jacobian
# Redefine variables and initial guess
# definition of variables and initial guess
P = tf.Variable([0,-16, -20], dtype=tf.float64)
ell = tf.Variable([4.0,6.0,5.0], dtype=tf.float64)
a = tf.Variable(6.0, dtype=tf.float64)
d_vertical = tf.Variable(3, dtype=tf.float64)

# We concatenate T, Theta 1, Theta 2...
# x = tf.Variable([50, 50*np.pi/180, 50*np.pi/180/2, -50*np.pi/180], dtype=tf.float64)
```

```
x = tf.Variable([15, 0.45, 0.1,-0.2], dtype=tf.float64)


# plot the first
fig = plt.gcf()
# we are plotting two different configurations (initial and final and label them)
ax = fig.gca()
plt_1 = PlotConfig_up(x[1:], ell, ax, label='Initial Config')

# param for Newton's method
err_tol = 1e-12
err = 1
max_iter = 100

# initialize
iter = 0
while (err > err_tol) & (iter < max_iter):
    iter += 1
    print("Iter %d: " % iter)

    # calculate the jacobian matrix with autodiff
    start = time.time()

    with tf.GradientTape() as tape:
        # forward pass
        # Resid = -Feval(x, ell, a, P, d_vertical)
        # use assign_sub to update Resid in place
        Resid = Feval(x, ell, a, P, d_vertical)
    # get the gradient of F with respect to x
    J = tape.jacobian(Resid, x)
    # print("J is: \n", J)

    end = time.time()
    print("Time for autodiff: ", end-start)

    if iter == 1:
        # Resid0 = Resid
        Resid0 = Resid

    err = tf.norm(Resid)/tf.norm(Resid0)

    # Update x using the Newton-Raphson method
    # use x.assign_sub to update x in place
    x.assign_add(tf.squeeze(tf.linalg.solve(J, tf.expand_dims(-Resid, 1))))
    # print("x is: \n", x)

    if iter == max_iter:
        print("Maximum number of iterations reached")
        print("x is: \n", x)
        print("J is: \n", J)
        print("Residual is: \n", Resid)
        print("Error is: \n", err)
    if err < err_tol:
        print("Converged to tolerance")
```

```
            print("x is: \n", x)
            print("J is: \n", J)
            print("Residual is: \n", Resid)
            print("Error is: \n", err)
            break

    # plot the final configuration
    plt_2 = PlotConfig_up(x[1:], ell, ax, label='Final Config')
    plt.title('Initial and final Config after %d Iter with AD' % iter)
    plt.grid(True)
    plt.xlabel('x(m)')
    plt.ylabel('y(m)')
    plt.legend()
    plt.show()
```

```
Iter 1:
Time for autodiff:  0.6794023513793945
Iter 2:
Time for autodiff:  0.6706385612487793
Iter 3:
Time for autodiff:  0.6703572273254395
Iter 4:
Time for autodiff:  0.6719644069671631
Iter 5:
Time for autodiff:  0.6678853034973145
Iter 6:
Time for autodiff:  0.6677122116088867
Iter 7:
Time for autodiff:  0.670135498046875
Iter 8:
Time for autodiff:  0.6702513694763184
Converged to tolerance
x is:
 <tf.Variable 'Variable:0' shape=(4,) dtype=float64, numpy=array([17.88840896,  0.93580275,  0.43344988, -0.58004954])>
J is:
 tf.Tensor(
[[  0.89443393  50.84051363 -21.71986945   0.        ]
 [  1.11804242   0.          21.71986945 -25.56859128]
 [  0.          -3.22030203  -2.52002484   2.74032687]
 [  0.          -2.37268937  -5.44513313  -4.1821775 ]], shape=(4, 4), dtype=float64)
Residual is:
 tf.Tensor([-1.77635684e-15  0.00000000e+00  0.00000000e+00  4.44089210e-16], shape=(4,), dtype=float64)
Error is:
 tf.Tensor(9.746452899901953e-17, shape=(), dtype=float64)
```

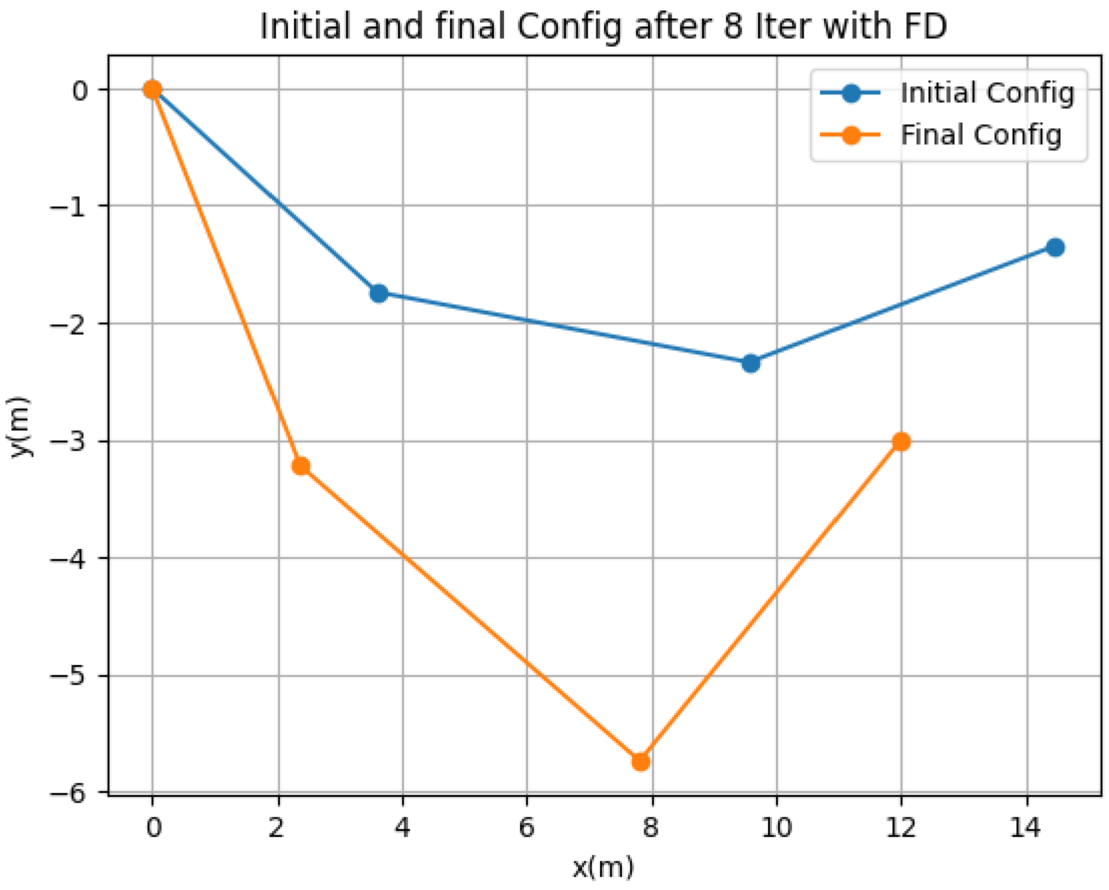


In [ ]:
```
# Redefine variables and initial guess
# definition of variables and initial guess
P = tf.Variable([0,-16, -20], dtype=tf.float64)
ell = tf.Variable([4.0,6.0,5.0], dtype=tf.float64)
a = tf.Variable(6.0, dtype=tf.float64)
d_vertical = tf.Variable(3, dtype=tf.float64)

# We concatenate T, Theta 1, Theta 2...
# x = tf.Variable([50, 50*np.pi/180, 50*np.pi/180/2, -50*np.pi/180], dtype=tf.float64)
x = tf.Variable([15, 0.45, 0.1,-0.2], dtype=tf.float64)

# implementation of Newton's method (With FD Jacobian
# plot the first
fig = plt.gcf()
# we are plotting two different configurations (initial and final and label them)
ax = fig.gca()
plt_1 = PlotConfig_up(x[1:], ell, ax, label='Initial Config')

# param for Newton's method
err_tol = np.float64(1e-12)
err = np.float64(1)
max_iter = 100
eps = np.float64(1e-8)
h = np.eye(4)

# initialize
iter = 0
fig = plt.gcf()
```

```
ax = fig.gca()

while (err > err_tol) & (iter < max_iter):
    # # plot the configuration
    # PlotConfig_up(x[1:], ell, ax)
    iter += 1
    print("Iter %d: " % iter)

    # forward pass (for evaluation)
    Resid = np.float64(Feval(x, ell, a, P, d_vertical))

    if iter == 1:
        Resid0 = Resid

    # calculate the error
    err = np.float64(norm(Resid))/np.float64(norm(Resid0))
    # err = norm(Resid)/norm(Resid0)
    print("Error is: ", err)

    # calculate the jacobian matrix with FD
    start = time.time()
    J = J_FD(x, eps, h)     # x = T,t1,t2,t3
    end = time.time()
    print("Time for FD: ", end-start)

    # solve the linear system
    delta_x = np.linalg.solve(J, -Resid)

    # update x
    T = x[0] + delta_x[0]
    theta = x[1:] + delta_x[1:]
    x = tf.Variable(np.concatenate(([T], theta)), dtype=tf.float64)

    # print("x is: \n", x)
    # print("F is: \n", Resid)
    # print("J is: \n", J)

    if iter == max_iter:
        print("Maximum number of iterations reached")
        print("x is: \n", x)
        print("Residual is: \n", Resid)
        print("Error is: \n", err)
    if err < err_tol:
        print("Converged to tolerance")
        print("x is: \n", x)
        print("Residual is: \n", Resid)
        print("Error is: \n", err)
        break

# plot the final configuration
plt_2 = PlotConfig_up(x[1:], ell, ax, label='Final Config')
plt.title('Initial and final Config after %d Iter with FD' % iter)
plt.grid(True)
plt.xlabel('x(m)')
```

```
    plt.ylabel('y(m)')
    plt.legend()
    plt.show()
```

```
Iter 1:
Error is:  1.0
Time for FD:  0.030774593353271484
Iter 2:
Error is:  1.5862347310110296
Time for FD:  0.030662059783935547
Iter 3:
Error is:  0.7125281922235093
Time for FD:  0.030745744705200195
Iter 4:
Error is:  0.136175369182438
Time for FD:  0.03072047233581543
Iter 5:
Error is:  0.005996090634284588
Time for FD:  0.030786991119384766
Iter 6:
Error is:  1.1549083321668428e-05
Time for FD:  0.030765533447265625
Iter 7:
Error is:  4.537273185518587e-11
Time for FD:  0.0307769775390625
Iter 8:
Error is:  0.0
Time for FD:  0.03083491325378418
Converged to tolerance
x is:
 <tf.Variable 'Variable:0' shape=(4,) dtype=float64, numpy=array([17.88840896,  0.93580275,  0.43344988, -0.58004954])>
Residual is:
 [0. 0. 0. 0.]
Error is:
 0.0
```


Initial and final Config after 8 Iter with FD
Initial Config
Final Config
y(m)
x(m)
0
−1
−2
−3
−4
−5
−6
0
2
4
6
8
10
12
14