Project 1: geometry optimization

The code template below is another Python recipe from the *IPython Cookbook, Second Edition* by Cyrille Rossant, the online text we used to explore the bifurcation diagram of the logistic map. You can find relevant explanations on the code in chapter 9.4 Finding the equilibrium state of a physical system by minimizing its potential energy

This project will require you to find minimal energy configurations of a bar made of a system of masses connected by springs. You will work with increasingly complex scenarios that will test the limits of your computing devices, and that will become increasingly realistic analogies of the process of discovering new properties of materials.

(1) Increase the number of masses in the system, while keeping the length of the bar constant, for several values of number of masses. are the results you obtain physically plausible?

Repeat the process by keeping the initial distance among the masses constant.

(2) Now suppose there is a certain proportion Q of masses that are heavier, namely of mass $\tilde{m} = 2m$. Modify the code below so that these heavier masses are randomly distributed in the system.

(3) Change the stiffness k of the springs attached to a pair of masses (m_i, m_j) according to the function $k = 10000 * (\frac{m_i m_j}{(0.1)^2})^{0.5}$, i.e. the heavier the masses, the stiffer the spring connecting them.

Once more, explore the resulting shape of the bar in equilibrium, and determine whether it is plausible.

N.B. There are multiple strings attached to each mass.

(4) Consider a number of masses up to $n = 10^3$ (if your computational resources allow it), and find which is the optimal proportion Q of heavier masses that, on average, will maximize bending of the bar.

N.B. you will need to test which optimization strategy is the most reliable in finding a realistic equilibrium shape. Address any issue with the convergence of the optimization algorithms.

(5) Research the literature on crystal structure prediction (CSP) by geometry optimization and explain the (many) differences and the similarities between the problem of this project and those of CSP.

9.4. Finding the equilibrium state of a physical system by minimizing its potential energy

Introduction

This project is theoretically based on crystal structures. A crystal structure is the repeating arrangement of atoms (molecules or ions) throughout a crystal (Yang and Day 2021). Crystal Structure Prediction (CSP) corresponds to the ability to predict the crystal structure that a molecule will adopt (Yang and Day 2021).

To do so, in this project, we look at a system of a bar of masses connected by springs and try to find the equilibrium state.

We consider a structure made of masses and springs, attached to a vertical wall as gravity acts on it.

In [2]:
```
import numpy as np #import numpy as np to use arrays
import scipy.optimize as opt # import the scipy.optimize packages for optimization and root finding
import matplotlib.pyplot as plt # import the matplotlib.pyplo package the allows us to plot as plt
%matplotlib inline
import random #import random package that enables us to randomly select objects
import math
from statistics import mean
```

Question 1: This question asks us to increase the number of masses while keeping the length of the bare constant to see if the result are physically possible.

The following lines of code define input variables that define the bar's properties that will be used throughout the code, such as the mass m of the masses, gravity, and the spring stiffness. We therefore, changed the number of masses from 10 to 40 to analyze the plausibility of this configuration. We kept the length of the bar constant. We also changed the initial and relaxed distances between the mass to control the horizontal and vertical lenghts and relaxed length of the bar.

We also change the initial and relaxed distances between the mass to control the horizontal and vertical lenghts and the relaxed length of the bar.

Question 2: This question modifies the code so that Q heavier masses (multiplied by 2) are randomly distributed in the system. To do so, we created an M array to store all the initial n masses of 0.1 in the system. Then, we created a new variable, Q, corresponding to the number of heavier masses distributed in the configuration. We also created a new array of randomly selected Q indices. Then we multiply the M array elements at those indices by 2.

In [3]:
```
g = 9.81  # gravity of Earth

n = 40 # number of masses
# m = 0.1
M = 0.1 * np.ones((n,)) # we change the mass into an array of n mass of 0.1
Q = 15 # variable q that correspond to the  proportion  Q of masses that are heavier
e = .1# initial distance between the masses
l = e  # relaxed length of the springs
k = 10000# spring stiffness

#fix total length
LL=5
ex= LL/(n/2-1)

#changed made to control the horizontal and verticla lengths and relaxed lenght
# ex = e*2 #intinal horizontla lengths
ey = e*5 #intinal verticals lengths

Lx = ex #relaxed horizontla lengths
Ly = ey #relaxed verticals lengths

ind = np.random.choice(n,Q)  # arrays containing Q number of randomly selectec values

M[ind]=2*M[ind] # multiply the mass of those indices by tow in the M array (the mass array)

print(M) #print the M arrays
```

```
[0.1 0.1 0.2 0.1 0.1 0.1 0.1 0.1 0.2 0.1 0.1 0.2 0.1 0.1 0.1 0.2 0.1 0.1 0.2 0.1
 0.2 0.2 0.1 0.1 0.2 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.2 0.1 0.1 0.2 0.1 0.2
 0.1 0.2 0.1 0.1]
```

By printing the array, we can see that Q number of the mass have been changed to 0.2

We also need to change the P0 array that corresponds to the positions of all the masses according to our new changes in the vertical and horizontal length of the bar so that is shown correctly in the figure. We change it by multiplying each horizontal element by ex and changing every other vertical element to ey

In [4]:
```
P0 = np.zeros((n, 2)) #array P0 correspond to the position of all the masses
P0[:, 0] = np.repeat(ex * np.arange(n // 2), 2) #horizontal position
P0[:, 1] = np.tile((0, -ey), n // 2) # vertical positions
print(P0) # print the m arrays
```

```
[[ 0.          0.        ]
 [ 0.         -0.5       ]
 [ 0.26315789  0.        ]
 [ 0.26315789 -0.5       ]
 [ 0.52631579  0.        ]
 [ 0.52631579 -0.5       ]
 [ 0.78947368  0.        ]
 [ 0.78947368 -0.5       ]
 [ 1.05263158  0.        ]
 [ 1.05263158 -0.5       ]
 [ 1.31578947  0.        ]
 [ 1.31578947 -0.5       ]
 [ 1.57894737  0.        ]
 [ 1.57894737 -0.5       ]
 [ 1.84210526  0.        ]
 [ 1.84210526 -0.5       ]
 [ 2.10526316  0.        ]
 [ 2.10526316 -0.5       ]
 [ 2.36842105  0.        ]
 [ 2.36842105 -0.5       ]
 [ 2.63157895  0.        ]
 [ 2.63157895 -0.5       ]
 [ 2.89473684  0.        ]
 [ 2.89473684 -0.5       ]
 [ 3.15789474  0.        ]
 [ 3.15789474 -0.5       ]
 [ 3.42105263  0.        ]
 [ 3.42105263 -0.5       ]
 [ 3.68421053  0.        ]
 [ 3.68421053 -0.5       ]
 [ 3.94736842  0.        ]
 [ 3.94736842 -0.5       ]
 [ 4.21052632  0.        ]
 [ 4.21052632 -0.5       ]
 [ 4.47368421  0.        ]
 [ 4.47368421 -0.5       ]
 [ 4.73684211  0.        ]
 [ 4.73684211 -0.5       ]
 [ 5.          0.        ]
 [ 5.         -0.5       ]]
```

Question 3: This question asks us to he stiffness k of the springs attached to a pair of masses (m_i, m_j) according to the function $k = 10000 * (\frac{m_i m_j}{(0.1)^2})^{0.5}$.

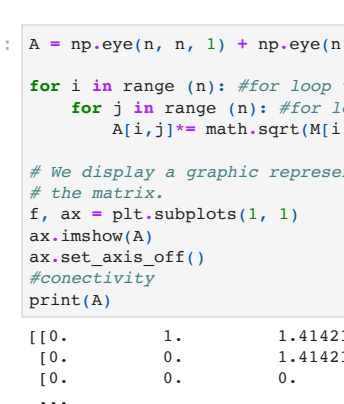
To do so we changed the array A, which represents the conectivity between the masses as k only appears to muliply A. orinally the array A is shaped as with ones on the diagonal 1 and 2 away from the middle diagonal and zeros elsewhere. To change the stiffenes of the spring according to the mass, we can multiply the A array by the new constant according to the function $k = (\frac{m_i m_j}{(0.1)^2})^{0.5}$.

In [5]:
```
A = np.eye(n, n, 1) + np.eye(n, n, 2) # matrix giving connectivity

for i in range (n): #for loop to iterate through m_i
    for j in range (n): #for loop to iterate through m_j
        A[i,j]*= math.sqrt(M[i]*M[j]/((0.1)**2)) #multiply each indicies of A by sqrt(M[i]*M[j]/((0.1)**2))

# We display a graphic representation of
# the matrix.
f, ax = plt.subplots(1, 1)
ax.imshow(A)
ax.set_axis_off()
#conectivity
print(A)
```

```
[[0.         1.         1.41421356 ... 0.         0.         0.        ]
 [0.         0.         1.41421356 ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 ...
 [0.         0.         0.         ... 0.         1.41421356 1.41421356]
 [0.         0.         0.         ... 0.         0.         1.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]]
```



Question 3:

We can see how the new A matrix represents the connectivity of the mass according to how heavy the masses are and how connected there are to other masses. This is then reflected in the stiffness as the heavier the mass, the stiffer the spring.

In [6]:
```
L = Ly* (np.eye(n, n, 1)) + Lx*np.eye(n, n, 2) #change the verticala and horizonal length we set up
for i in range(n // 2 - 1):
    L[2 * i + 1, 2 * i + 2] = np.sqrt(Lx**2+Ly**2) #changed the diagonal length so that it mathes the horizontal aand verical length

print (L)
```

```
[[0.         0.5        0.26315789 ... 0.         0.         0.        ]
 [0.         0.         0.56502396 ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 ...
 [0.         0.         0.         ... 0.         0.56502396 0.26315789]
 [0.         0.         0.         ... 0.         0.         0.5       ]
 [0.         0.         0.         ... 0.         0.         0.        ]]
```

In [7]:
```
I, J = np.nonzero(A)
```

In [8]:
```
def dist(P):
    return np.sqrt((P[:,0]-P[:,0][:,np.newaxis])**2 +
                   (P[:,1]-P[:,1][:,np.newaxis])**2)
```

In [9]:
```
def show_bar(P):
    fig, ax = plt.subplots(1, 1, figsize=(5, 4))

    # Wall.
    ax.axvline(0, color='k', lw=3)

    # Distance matrix.
    D = dist(P)

    # Get normalized elongation in [-1, 1].
    elong = np.array([D[i, j] - L[i, j]
                      for i, j in zip(I, J)])
    elong_max = np.abs(elong).max()

    # The color depends on the spring tension, which
    # is proportional to the spring elongation.
    colors = np.zeros((len(elong), 4))
    colors[:, -1] = 1  # alpha channel is 1

    # Use two different sequentials colormaps for
    # positive and negative elongations, to show
    # compression and extension in different colors.
    if elong_max > 1e-10:
        # We don't use colors if all elongations are
        # zero.
        elong /= elong_max
        pos, neg = elong > 0, elong < 0
        colors[pos] = plt.cm.copper(elong[pos])
        colors[neg] = plt.cm.bone(-elong[neg])

    # We plot the springs.
    for i, j, c in zip(I, J, colors):
        ax.plot(P[[i, j], 0],
                P[[i, j], 1],
                lw=2,
                color=c,
                )

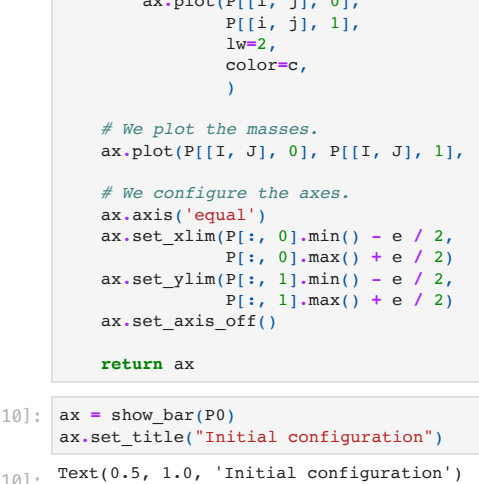
    # We plot the masses.
    ax.plot(P[[I, J], 0], P[[I, J], 1], 'ok',)

    # We configure the axes.
    ax.axis('equal')
    ax.set_xlim(P[:, 0].min() - e / 2,
                P[:, 0].max() + e / 2)
    ax.set_ylim(P[:, 1].min() - e / 2,
                P[:, 1].max() + e / 2)
    ax.set_axis_off()

    return ax
```

In [10]:
```
ax = show_bar(P0)
ax.set_title("Initial configuration")
```

Out[10]:
```
Text(0.5, 1.0, 'Initial configuration')
```



This figure illustrates the system's initial configuration before gravity acts on it. We can see in this figure that the initial distances between the mass were changed and follows a ratio of 2/5 as its length is 0.2 horizontally and 0.5 vertically.

In [11]:
```
def energy(P):
    # The argument P is a vector (flattened matrix).
    # We convert it to a matrix here.
    P = P.reshape((-1, 2))
    # We compute the distance matrix.
    D = dist(P)
    # The potential energy is the sum of the
    # gravitational and elastic potential energies.
    return ((g * M * P[:, 1]).sum() +
            .5 * (k * A * (D - L)**2).sum())
```

In [12]:
```
energy(P0.ravel())
```

Out[12]:
```
-12.753000000000002
```

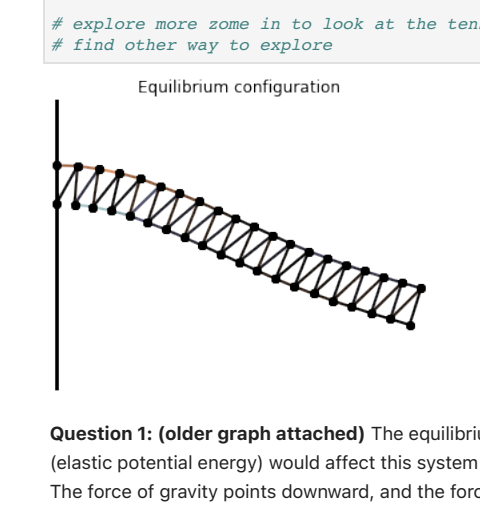
In [13]:
```
bounds = np.c_[P0[:2, :].ravel(),
               P0[:2, :].ravel()].tolist() + \
    [[None, None]] * (2 * (n - 2))
```

In [14]:
```
P1 = opt.minimize(energy, P0.ravel(),
                  method='Powell',
                  bounds=bounds).x.reshape((-1, 2))
```

In [15]:
```
ax = show_bar(P1)
ax.set_title("Equilibrium configuration")

img3=plt.savefig('question 3 ffig.png')

# explore more zome in to look at the tension
# find other way to explore
```



Question 1: (older graph attached) The equilibrium configuration shows us how the potential energy due to gravity (gravitational force) and spring forces (elastic potential energy) would affect this system of mass. The graph for this question shows that the bar tilts upward in the middle and then goes back down. The force of gravity points downward, and the force from the wall holding the system holds the left side of the bar. Therefore, a plausible equilibrium configuration would curve downward toward the right as the force of gravity wouldn't be canceled by force from the wall. Therefore this configuration does not appear to be plausible. However, the system is also held by springs. Therefore a concentration of the tension could cause the system to curve back up to release some tne, but since the graph of question 1 has such a dramatic curve in the middle, it is not plausible.

Question 2: (older graph attached) As said earlier, the force of gravity points downward, and the force from the wall holding the system holds the left side of the bar. For me, the configuration of question 2 is not bending down, so it is not plausible.

Question 3: TThe equilibrium configuration shows us how the potential energy due to gravity (gravitational force) and spring forces (elastic potential energy) would affect this system of mass. We can see that the bar is tilting upward, which could correspond to a non-physically possible situation. The force of gravity points downward, and the force from the wall holding the system holds the left side of the bar. Therefore, a plausible equilibrium configuration would curve downward toward the right as the force of gravity wouldn't be canceled by force from the wall. Thus this configuration seems slightly not plausible. However, in the equilibrium configuration, the springs near the wall would be maximally extended, and the ones at the bottom compressed. This shows in the figure, but since the system is curving up, we still need to figure out if this tilt is natural.

To do that, we can focus on the tension force. Too much tension possibly causes the spring system to tilt up to release some of that tension. Therefore, we can zoom into the plot to evaluate the tension. We, thus, created a new show_bar2 function to zoom into the graph at the position where the spring is tilting up.

In [106...
```
def show_bar2(P):
    fig, ax = plt.subplots(1, 1, figsize=(5, 4))

    # Wall.
    ax.axvline(0, color='k', lw=3)

    # Distance matrix.
    D = dist(P)

    # Get normalized elongation in [-1, 1].
    elong = np.array([D[i, j] - L[i, j]
                      for i, j in zip(I, J)])
    elong_max = np.abs(elong).max()

    # The color depends on the spring tension, which
    # is proportional to the spring elongation.
    colors = np.zeros((len(elong), 4))
    colors[:, -1] = 1  # alpha channel is 1

    # Use two different sequentials colormaps for
    # positive and negative elongations, to show
    # compression and extension in different colors.
    if elong_max > 1e-10:
        # We don't use colors if all elongations are
        # zero.
        elong /= elong_max
        pos, neg = elong > 0, elong < 0
        colors[pos] = plt.cm.copper(elong[pos])
        colors[neg] = plt.cm.bone(-elong[neg])

    # We plot the springs.
    for i, j, c in zip(I, J, colors):
        ax.plot(P[[i, j], 0],
                P[[i, j], 1],
                lw=2,
                color=c,
                )

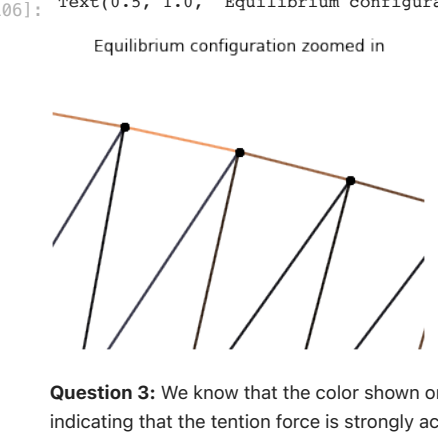
    # We plot the masses.
    ax.plot(P[[I, J], 0], P[[I, J], 1], 'ok',)

    # We configure the axes.
    ax.axis('equal')
    ax.set_xlim(P[:n/4, 0].min() - e / 2,
                P[:n/4, 0].max() + e / 2)
    ax.set_ylim(P[:n/2, 1].min() - e / 2,
                P[:, 1].max() + e / 5)
    ax.set_axis_off()

    return ax


ax = show_bar2(P1)
ax.set_title("Equilibrium configuration zoomed in")
```

Out[106]:
```
Text(0.5, 1.0, 'Equilibrium configuration zoomed in')
```



Question 3: We know that the color shown on the spring depends on the spring tension. we can see that when the sping is bending the ie is very yellow indicating that the tention force is strongly acting on those masses. However, when the masses are stiling up we can see thet the color is fading to black. This mean tha the masses are tilting to relase the tension and stay in the equilibrium position. Therefore, I conclude that configuation is most likely plausible.

Question 4: This question asks us which optimal proportion Q of heavier masses that, on average, will maximize bending of the bar.

We first start by increasing the number of mass to find he maximum number of masses that we can add. We find that this number of mass is 100.

In [16]:
```
g = 9.81  # gravity of Earth

n = 100 # number of masses
# m = 0.1
M = 0.1 * np.ones((n,)) # we change the mass into an array of n mass of 0.1
Q = 40 # variable q that correspond to the  proportion  Q of masses that are heavier
e = .1# initial distance between the masses
l = e  # relaxed length of the springs
k = 10000# spring stiffness

#fix total length
LL=5
ex= LL/(n/2-1)

#changed made to control the horizontal and verticla lengths and relaxed lenght
ex = e*2 #intinal horizontla lengths
ey = e*5 #intinal verticals lengths


Lx = ex #relaxed horizontla lengths
Ly = ey #relaxed verticals lengths
# ind = [2,5,18]

ind = np.random.choice(n,Q)  # arrays containing Q number of randomly selectec values

M[ind]=2*M[ind] # multiply the mass of those indices by tow in the M array (the mass array)

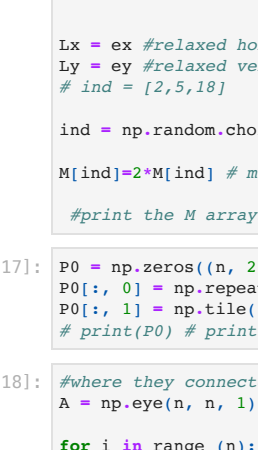
 #print the M arrays
```

In [17]:
```
P0 = np.zeros((n, 2)) #array P0 correspond to the position of all the masses
P0[:, 0] = np.repeat(ex * np.arange(n // 2), 2) #horizontal position
P0[:, 1] = np.tile((0, -ey), n // 2) # vertical positions
# print(P0) # print the m arrays
```

In [18]:
```
#where they connected
A = np.eye(n, n, 1) + np.eye(n, n, 2) # matrix giving connectivity

for i in range (n):
    for j in range (n):
        A[i,j]*= math.sqrt(M[i]*M[j]/((0.1)**2))


#A = A * math.sqrt(M_pair/(0.1)**2)
#if want to change k change A
# We display a graphic representation of
# the matrix.
f, ax = plt.subplots(1, 1)
ax.imshow(A)
ax.set_axis_off()
#conectivity
# print(A)
```



In [19]:
```
L = Ly* (np.eye(n, n, 1)) + Lx*np.eye(n, n, 2) #change the verticals and horisonal length
for i in range(n // 2 - 1):
    L[2 * i + 1, 2 * i + 2] = np.sqrt(Lx**2+Ly**2) #changed the diagonal length

# print (L)
```

In [20]:
```
I, J = np.nonzero(A)
```

In [21]:
```
def dist(P):
    return np.sqrt((P[:,0]-P[:,0][:,np.newaxis])**2 +
                   (P[:,1]-P[:,1][:,np.newaxis])**2)
```

In [22]:
```
def show_bar(P):
    fig, ax = plt.subplots(1, 1, figsize=(5, 4))

    # Wall.
    ax.axvline(0, color='k', lw=3)

    # Distance matrix.
    D = dist(P)

    # Get normalized elongation in [-1, 1].
    elong = np.array([D[i, j] - L[i, j]
                      for i, j in zip(I, J)])
    elong_max = np.abs(elong).max()

    # The color depends on the spring tension, which
    # is proportional to the spring elongation.
    colors = np.zeros((len(elong), 4))
    colors[:, -1] = 1  # alpha channel is 1

    # Use two different sequentials colormaps for
    # positive and negative elongations, to show
    # compression and extension in different colors.
    if elong_max > 1e-10:
        # We don't use colors if all elongations are
        # zero.
        elong /= elong_max
        pos, neg = elong > 0, elong < 0
        colors[pos] = plt.cm.copper(elong[pos])
        colors[neg] = plt.cm.bone(-elong[neg])

    # We plot the springs.
    for i, j, c in zip(I, J, colors):
        ax.plot(P[[i, j], 0],
                P[[i, j], 1],
                lw=2,
                color=c,
                )

    # We plot the masses.
    ax.plot(P[[I, J], 0], P[[I, J], 1], 'ok',)

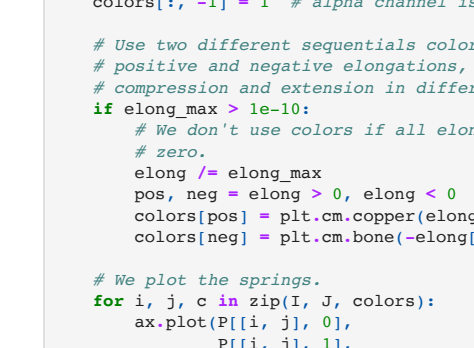
    # We configure the axes.
    ax.axis('equal')
    ax.set_xlim(P[:, 0].min() - e / 2,
                P[:, 0].max() + e / 2)
    ax.set_ylim(P[:, 1].min() - e / 2,
                P[:, 1].max() + e / 2)
    ax.set_axis_off()

    return ax
```

In [23]:
```
ax = show_bar(P0)

ax.set_title("Initial configuration")
```

Out[23]:
```
Text(0.5, 1.0, 'Initial configuration')
```



In [24]:
```
def energy(P):
    # The argument P is a vector (flattened matrix).
    # We convert it to a matrix here.
    P = P.reshape((-1, 2))
    # We compute the distance matrix.
    D = dist(P)
    # The potential energy is the sum of the
    # gravitational and elastic potential energies.
    return ((g * M * P[:, 1]).sum() +
            .5 * (k * A * (D - L)**2).sum())
```

In [25]:
```
energy(P0.ravel())
```

Out[25]:
```
-30.411
```

In [26]:
```
bounds = np.c_[P0[:2, :].ravel(),
               P0[:2, :].ravel()].tolist() + \
    [[None, None]] * (2 * (n - 2))
```

In [27]:
```
P1 = opt.minimize(energy, P0.ravel(),
                  method='Powell',
                  bounds=bounds).x.reshape((-1, 2))
```

We also need to find the a function that dscribes the bending of the system of masses. we choose to compute the center of mass in x position.

In [28]:
```
CenterOfMass= 0.0 # center of mass
Masses= 0.0 # sum of the massand their position

for k in range (n):
    Masses += M[k]*P0[k,:] #add all the mass multiplied my their position


CenterOfMass0= Masses/np.sum(M)
print(CenterOfMass0)


CenterOfMass= 0.0 # center of mass
Masses= 0.0 # sum of the massand their position

for k in range (n):
    Masses += M[k]*P1[k,:] #add all the mass multiplied my their position

CenterOfMass1= Masses/np.sum(M)
print(CenterOfMass1)

bending = CenterOfMass0[1]- CenterOfMass1[1]

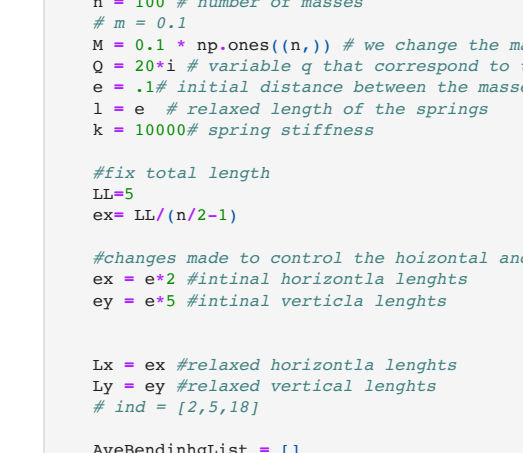
print(bending)
```

```
[ 4.80615385 -0.23846154]
[ 4.69226875 -1.25815949]
1.019697954324107
```

We see that x position of the center of mass is 2.21 horizontally which half of the our vertcal length of 5. The y position of the center of mass is -1.12 for the P1 array. This gives and array of 0.86. this makes sens as we can see the system is bended the middle

In [29]:
```
ax = show_bar(P1)
ax.set_title("Equilibrium configuration")
# many val of percentage
```

Out[29]:
```
Text(0.5, 1.0, 'Equilibrium configuration')
```



As we add more masses we can see that the system of mass doesn't look plausible. As it goes straight a the end.

Now we will look at the bending of more configurations with different proportions of Q heavier masses going from 0 to 100% every 20%. We took the average of 4 different bendings for each proportion to get a more precise result, as it was the maximum amount this computer allowed. This allowed us to look at the bending vs. the proportion of heavier masses.

In [30]:
```
bendingList=[]
for i in range(6):
    g = 9.81  # gravity of Earth

    n = 100 # number of masses
    # m = 0.1
    M = 0.1 * np.ones((n,)) # we change the mass into an array of n mass of 0.1
    Q= 20*i # variable q that correspond to the  proportion  Q of masses that are heavier
    e = .1# initial distance between the masses
    l = e  # relaxed length of the springs
    k = 10000# spring stiffness

    #fix total length
    LL=5
    ex= LL/(n/2-1)

    #changed made to control the horizontal and verticla lengths and relaxed lenght
    ex = e*2 #intinal horizontla lengths
    ey = e*5 #intinal verticals lengths


    Lx = ex #relaxed horizontla lengths
    Ly = ey #relaxed verticals lengths
    # ind = [2,5,18]

    AveBendingList = []

    for j in range(4):

        ind = np.random.choice(n,Q)  # arrays containing Q number of randomly selectec values

        M[ind]=2*M[ind] # multiply the mass of those indices by tow in the M array (the mass array)

        #print the M arrays
        P0 = np.zeros((n, 2)) #array P0 correspond to the position of all the masses
        P0[:, 0] = np.repeat(ex * np.arange(n // 2), 2) #horizontal position
        P0[:, 1] = np.tile((0, -ey), n // 2) # vertical positions
        # print(P0) # print the m arrays

        #where they connected
        A = np.eye(n, n, 1) + np.eye(n, n, 2) # matrix giving connectivity

        for i in range (n):
            for j in range (n):
                A[i,j]*= math.sqrt(M[i]*M[j]/((0.1)**2))


        L = Ly* (np.eye(n, n, 1)) + Lx*np.eye(n, n, 2) #change the verticals and horisonal length
        for i in range(n // 2 - 1):
            L[2 * i + 1, 2 * i + 2] = np.sqrt(Lx**2+Ly**2) #changed the diagonal length


        I, J = np.nonzero(A)

        def dist(P):
            return np.sqrt((P[:,0]-P[:,0][:,np.newaxis])**2 +
                           (P[:,1]-P[:,1][:,np.newaxis])**2)

        def show_bar(P):
            fig, ax = plt.subplots(1, 1, figsize=(5, 4))

            # Wall.
            ax.axvline(0, color='k', lw=3)

            # Distance matrix.
            D = dist(P)

            # Get normalized elongation in [-1, 1].
            elong = np.array([D[i, j] - L[i, j]
                              for i, j in zip(I, J)])
            elong_max = np.abs(elong).max()

            # The color depends on the spring tension, which
            # is proportional to the spring elongation.
            colors = np.zeros((len(elong), 4))
            colors[:, -1] = 1  # alpha channel is 1

            # Use two different sequentials colormaps for
            # positive and negative elongations, to show
            # compression and extension in different colors.
            if elong_max > 1e-10:
                # We don't use colors if all elongations are
                # zero.
                elong /= elong_max
                pos, neg = elong > 0, elong < 0
                colors[pos] = plt.cm.copper(elong[pos])
                colors[neg] = plt.cm.bone(-elong[neg])

            # We plot the springs.
            for i, j, c in zip(I, J, colors):
                ax.plot(P[[i, j], 0],
                        P[[i, j], 1],
                        lw=2,
                        color=c,
                        )

            # We plot the masses.
            ax.plot(P[[I, J], 0], P[[I, J], 1], 'ok',)

            # We configure the axes.
            ax.axis('equal')
            ax.set_xlim(P[:, 0].min() - e / 2,
                        P[:, 0].max() + e / 2)
            ax.set_ylim(P[:, 1].min() - e / 2,
                        P[:, 1].max() + e / 2)
            ax.set_axis_off()

            return ax

            #ax = show_bar(P0)

            #ax.set_title("Initial configuration")

        def energy(P):
            # The argument P is a vector (flattened matrix).
            # We convert it to a matrix here.
            P = P.reshape((-1, 2))
            # We compute the distance matrix.
            D = dist(P)
            # The potential energy is the sum of the
            # gravitational and elastic potential energies.
            return ((g * M * P[:, 1]).sum() +
                    .5 * (k * A * (D - L)**2).sum())

        energy(P0.ravel())

        bounds = np.c_[P0[:2, :].ravel(),
                       P0[:2, :].ravel()].tolist() + \
            [[None, None]] * (2 * (n - 2))

        P1 = opt.minimize(energy, P0.ravel(),
                          method='Powell',
                          bounds=bounds).x.reshape((-1, 2))

        # center of mass to dermine the bending

        #Bending
        CenterOfMass0= 0.0 # center of mass
        Masses= 0.0 # sum of the massand their position

        for k in range (n):
            Masses += M[k]*P0[k,:] #add all the mass multiplied my their position

        CenterOfMass0= Masses/np.sum(M)


        CenterOfMass1= 0.0 # center of mass
        Masses= 0.0 # sum of the massand their position

        for k in range (n):
            Masses += M[k]*P1[k,:] #add all the mass multiplied my their position

        CenterOfMass1= Masses/np.sum(M)

        bending = CenterOfMass0[1]- CenterOfMass1[1]

        AveBendingList.append(bending)

    bendingAve = mean(AveBendingList)

    bendingList.append(bendingAve)

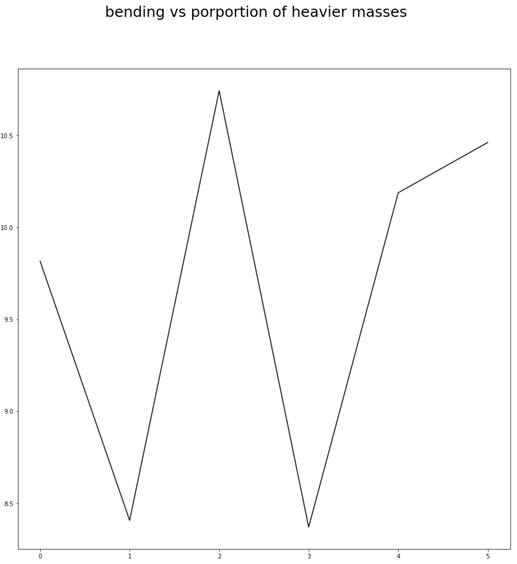
print(bendingList)
```

```
[9.815553792042666, 8.405650148985803, 10.742628795762855, 8.369085308688284, 10.187808056361408, 10.461355940075841]
```

In [33]:
```
fig1, (ax1) = plt.subplots(figsize=(15, 15),sharex=True) # create figure 1
```

```
fig1.suptitle('bending vs porportion of heavier masses', fontsize=25) #assigne a title to figure 1
ax1.plot(bendingList,'k') #plot the sampling created for H
```

Out[33]:
```
[<matplotlib.lines.Line2D at 0x7fb1c853e4c0>]
```



We have the highest bending happen for proportions at 40% 80%, and 100% with the highest bending at 40%.

Now let's see if this system configuration of 40 is physically plausible.

In [36]:
```
g = 9.81  # gravity of Earth

n = 100 # number of masses
# m = 0.1
M = 0.1 * np.ones((n,)) # we change the mass into an array of n mass of 0.1
Q = 40 # variable q that correspond to the  proportion  Q of masses that are heavier
e = .1# initial distance between the masses
l = e  # relaxed length of the springs
k = 10000# spring stiffness

#fix total length
LL=5
ex= LL/(n/2-1)

#changes made to control the hoizontal and verticla lengths and relaxed lenght
ex = e*2 #intinal horizontla lenghts
ey = e*5 #intinal verticla lenghts


Lx = ex #relaxed horizontla lenghts
Ly = ey #relaxed vertical lenghts
# ind = [2,5,18]

ind = np.random.choice(n,Q)  # arrays containing Q number of rendomly selectec values

M[ind]=2*M[ind] # multiply the mass of those indices by tow in the M array (the mass array)

#print the M arrays
P0 = np.zeros((n, 2)) #array P0 correspond to the position of all the masses
P0[:, 0] = np.repeat(ex * np.arange(n // 2), 2) #horizontal position
P0[:, 1] = np.tile((0, -ey), n // 2) # vertical positions
# print(P0) # print the m arrays

#where they connected
A = np.eye(n, n, 1) + np.eye(n, n, 2) # matrix giving connectivity

for i in range (n):
    for j in range (n):
        A[i,j]*= math.sqrt(M[i]*M[j]/((0.1)**2))



L = Ly* (np.eye(n, n, 1)) + Lx*np.eye(n, n, 2) #change the verticala and horizonal length
for i in range(n // 2 - 1):
    L[2 * i + 1, 2 * i + 2] = np.sqrt(Lx**2+Ly**2) #changed the diagonal length


I, J = np.nonzero(A)

def dist(P):
    return np.sqrt((P[:,0]-P[:,0][:,np.newaxis])**2 +
                   (P[:,1]-P[:,1][:,np.newaxis])**2)

def show_bar(P):
    fig, ax = plt.subplots(1, 1, figsize=(5, 4))

    # Wall.
    ax.axvline(0, color='k', lw=3)

    # Distance matrix.
    D = dist(P)

    # Get normalized elongation in [-1, 1].
    elong = np.array([D[i, j] - L[i, j]
                      for i, j in zip(I, J)])
    elong_max = np.abs(elong).max()

    # The color depends on the spring tension, which
    # is proportional to the spring elongation.
    colors = np.zeros((len(elong), 4))
    colors[:, -1] = 1  # alpha channel is 1

    # Use two different sequentials colormaps for
    # positive and negative elongations, to show
    # compression and extension in different colors.
    if elong_max > 1e-10:
        # We don't use colors if all elongations are
        # zero.
        elong /= elong_max
        pos, neg = elong > 0, elong < 0
        colors[pos] = plt.cm.copper(elong[pos])
        colors[neg] = plt.cm.bone(-elong[neg])

    # We plot the springs.
    for i, j, c in zip(I, J, colors):
        ax.plot(P[[i, j], 0],
                P[[i, j], 1],
                lw=2,
                color=c,
                )

    # We plot the masses.
    ax.plot(P[[I, J], 0], P[[I, J], 1], 'ok',)

    # We configure the axes.
    ax.axis('equal')
    ax.set_xlim(P[:, 0].min() - e / 2,
                P[:, 0].max() + e / 2)
    ax.set_ylim(P[:, 1].min() - e / 2,
                P[:, 1].max() + e / 2)
    ax.set_axis_off()

    return ax

    #ax = show_bar(P0)

    #ax.set_title("Initial configuration")

def energy(P):
    # The argument P is a vector (flattened matrix).
    # We convert it to a matrix here.
    P = P.reshape((-1, 2))
    # We compute the distance matrix.
    D = dist(P)
    # The potential energy is the sum of the
    # gravitational and elastic potential energies.
    return ((g * M * P[:, 1]).sum() +
            .5 * (k * A * (D - L)**2).sum())

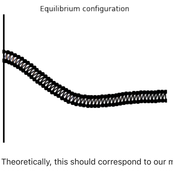
energy(P0.ravel())

bounds = np.c_[P0[:2, :].ravel(),
               P0[:2, :].ravel()].tolist() + \
    [[None, None]] * (2 * (n - 2))

P1 = opt.minimize(energy, P0.ravel(),
                  method='Powell',
                  bounds=bounds).x.reshape((-1, 2))

ax = show_bar(P1)
ax.set_title("Equilibrium configuration")
# many val of percentage
```

Out[36]:
```
Text(0.5, 1.0, 'Equilibrium configuration')
```



Theoretically, this should correspond to our most plausible configuration. We can see that its bending is more pronounced than for the last figure. However, it bends up in the middle. Also, we can see by the colors in the graph that the system is experiencing significant tension right before it tilts. As we said earlier, too much tension could cause the spring system to tilt up to release some of that tension. However, the masse doesn't go back down as expected with the force of gravity. This configuration could be plausible as the spring could lift the rest of the masses as we have heavier masses, even though this tilt is very drastic.

Question 5 :

According to (Yang and Day 2021) ,all current methods are based on searching for the local minima on the high dimensional energy surface. They represent the energy as a function of variables that describe a crystal structure. This is similar to what we did as we computed and minimized the potential energy used to find the equilibrium configuration.

This other study (Schlegel 2011) mentions quasi-Newton methods; similarly, to our project, they focus on finding the equilibrium geometry of an individual molecule focusing on minimizing the potential energy but using the potential energy surface, which we didn't do. However, their method is different as it includes factors such as the choice of coordinates, methods for calculating the step direction, Hessian updating methods, line search, and step size control strategies.

Another method mentioned in (Yang and Day 2021) is quasi-random (QR) structure generation; unlike our approach, which used minimized energy, this method provides a sampling of the energy landscape. It creates the lattice energy surface, essential for fully exploring the structural diversity available to a molecule in forming stable crystal structures. Those methods focused on potential energy surfaces, which describe the energy of a system. This is different as we only used the energy to plot an equilibrium configuration.

Lastly, this study (Kusaba, Liu, and Yoshida 2022) presents a unique crystal structure prediction (CSP) method that relies on a machine learning algorithm called metric learning. It used a binary classifier trained on many already identified crystal structures to determine the isomorphism of crystal structures formed by two given chemical compositions with an accuracy of approximately 96.4%. This method is entirely different from the project as it uses machine learning, which shows how diverse the technics the crystal structure prediction methods are.

In general, CSP is Risk is a powerful tool to avoid risk and identify potentially stable polymorphs ahead of time, saving money and time and improving manufacturing, and new materials discovery.

References:

Kusaba, Minoru, Chang Liu, and Ryo Yoshida. 2022. "Crystal Structure Prediction with Machine Learning-Based Element Substitution." Computational Materials Science 211 (August): 111496. https://doi.org/10.1016/j.commatsci.2022.111496.

Schlegel, H. Bernhard. 2011. "Geometry Optimization." WIREs Computational Molecular Science 1 (5): 790–809. https://doi.org/10.1002/wcms.34.

Yang, Shiyue, and Graeme M. Day. 2021. "Exploration and Optimization in Crystal Structure Prediction: Combining Basin Hopping with Quasi-Random Sampling." Journal of Chemical Theory and Computation 17 (3): 1988–99. https://doi.org/10.1021/acs.jctc.0c01101.

In [ ]:
```

```