Problem #1

Three ways to convert the similarities into distances:

1. Subtract the similarity (a value less than 1, which is greater when the items are considered more similar) from 1 to get our psi value, and set the result to the distance.
2. Subtract the similarity from 1, and multiply this number by 100 (to get our psi values) to get large distance values.
3. Subtract the similarity from 1, and compute the square root (to get our psi values) to get more distinct distance values (we take the square root instead of squaring since the psi value will be less than 1).

For my algorithm, I am going to choose to use method (1), since this is the simplest and most straightforward method, and makes the most intuitive sense when we think about what our psi value is supposed to represent in the algorithm (the psychological distance).

Problem #2

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

### BEGIN: CSV DATA EXTRACTION ###

# first, we derive a list of just the sport names
# we end up with this list stored in a global variable 'sports'
with open('Assignment5-similarities.csv', mode = 'r') as csv_file:
    reader = csv.reader(csv_file)
    sports = next(reader)

# we initialise our dictionary that will hold our
# psychological distance values as a global variable
similarityDict = {}

# then we derive the values from the list and place them in the dictionary
with open('Assignment5-similarities.csv', mode = 'r') as csv_file:

    # creatng a dictionary with the sport index as the key (0 thru 20)
    # and the similarities to the other sports as a list as the values (in order)

    reader = csv.DictReader(csv_file)
    elem1 = 0
    for row in reader:
        rowDict = row
        valuesList = rowDict.values()

        # we calculate our psychological distance to be the value from
        # the csv table subtraced from 1 (so that a higher similarity)
        # gives a lower psychological distance)
        valuesList = [(1 - float(x)) for x in valuesList]
        similarityDict[elem1] = valuesList
        elem1 += 1

### END: CSV DATA EXTRACTION ###
```

```
### BEGIN: HELPER FUNCTIONS ###

def genPositions(num):
    # use 21 for num to get random starting coordinates
    # for all 21 sports

    positionMatrix = []
    for i in range(0, num):
        a = np.random.uniform()
        b = np.random.uniform()
        positionMatrix.append((a, b))
    return positionMatrix

def oneStress(p1, p2, in1, in2):
    # calculate the stress of two points

    # we first calculate the physical distance between
    # the coordinates using the distance formula
    dist = ((p1[0] - p2[0]) ** 2) + ((p1[1] - p2[1]) ** 2)
    dist = math.sqrt(dist)

    # then we derive the psychological distance that we
    # extracted from the csv file earlier
    sim = similarityDict[in1][in2]

    return (sim - dist) ** 2

def stress(positions):
    # calculate the stress of all points

    s = 0
    for i in range(0, len(positions)):
        for j in range(i + 1, len(positions)):
            pos1, pos2 = positions[i], positions[j]
            s += oneStress(pos1, pos2, i, j)
    return s
```

Problem #3

```
def gradient(positions):
    gradient = []

    # calculate the gradient of a plane of coordinates
    # p is the index of our coordinate (0 through 20)
    p = 0
    for point in positions:

        # first we calculate gradX, the gradient of the current
        # coordinate's x component
        i = positions[:]
        i[p] = (i[p][0] + 0.001, i[p][1])
        s1 = stress(i)
        i[p]= (i[p][0] - 0.001, i[p][1])
        s2 = stress(i)
        gradX = (s1 - s2) / (2 * 0.001)

        # then we calculate gradY, the gradient of the current
        # coordinate's y component
        i = positions[:]
        i[p]= (i[p][0], i[p][1] + 0.001)
        s1 = stress(i)
        i[p]= (i[p][0], i[p][1] - 0.001)
        s2 = stress(i)
        gradY = (s1 - s2) / (2 * 0.001)

        gradient.append((gradX, gradY))
        p += 1

    # at the end, gradient hold a list of our gradients in order
    # i.e. (gradX1, gradY1, gradX2, gradY2, ...)
    return gradient
```

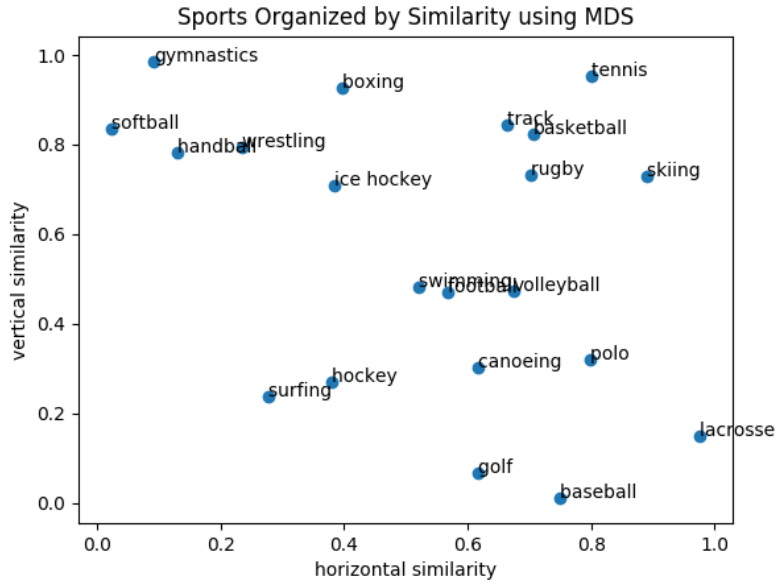
Problem #4 (Code)

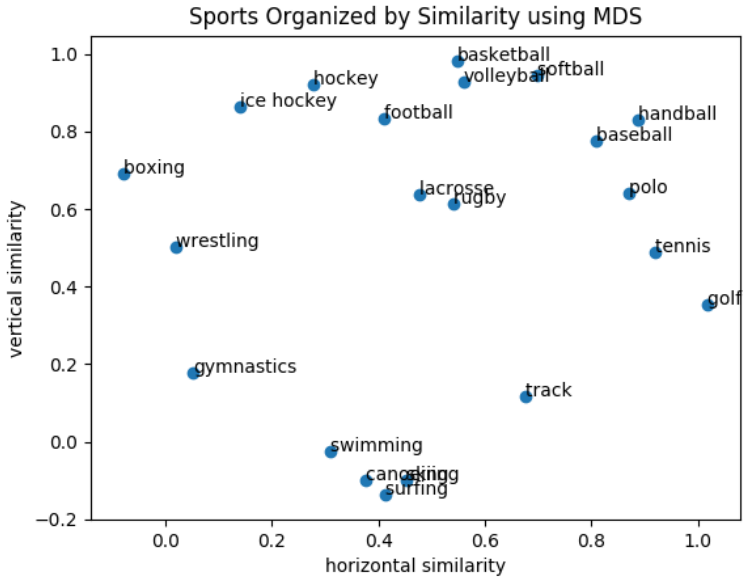
```
def adjustCoor(positions, gradient):

    # adjust our coordinates based on the values we got from the gradient
    for i in range(0, len(positions)):

        # checking whether to move x up or down
        if gradient[i][0] < 0:
            positions[i] = (positions[i][0] + (0.01 * gradient[i][0]), positions[i][1])
        elif gradient[i][0] > 0:
            positions[i] = (positions[i][0] - (0.01 * gradient[i][0]), positions[i][1])

        # checking whether to move y up or down
        if gradient[i][1] < 0:
            positions[i] = (positions[i][0], positions[i][1] + (0.01 * gradient[i][1]))
        elif gradient[i][1] > 0:
            positions[i] = (positions[i][0], positions[i][1] - (0.01 * gradient[i][1]))
    return positions

def displayPlot(positions):

    # calculates the stress of our coordinates
    #  and creates a scatterplot of our coordinates
    print(stress(positions))
    x = []
    y = []
    for i in range(0, len(positions)):
        x.append(positions[i][0])
        y.append(positions[i][1])
    plt.scatter(x, y)
    for i, sport in enumerate(sports):
        plt.annotate(sport, (x[i], y[i]))
    plt.show()

### END: HELPER FUNCTIONS
```

```
### BEGIN: MDS EXECUTION ###

def executeMDS(i):

    # our main execution function that pulls everything together
    # start by generating random coordinates for our sports
    p = genPositions(21)

    # display the orig. stress and scatterplot before MDS
    displayPlot(p)

    # for i iterations, we will calculate the gradient and
    # adjust our coordinates accordingly
    for k in range(0, i):
        g = gradient(p)
        p = adjustCoor(p, g)

    # display our ending stress and scatterplot after MDS
    displayPlot(p)
    return p
```

Problem #4 (Graphs)

Here are the scatterplots of the coordinates of the sports before vs. after MDS was applied:

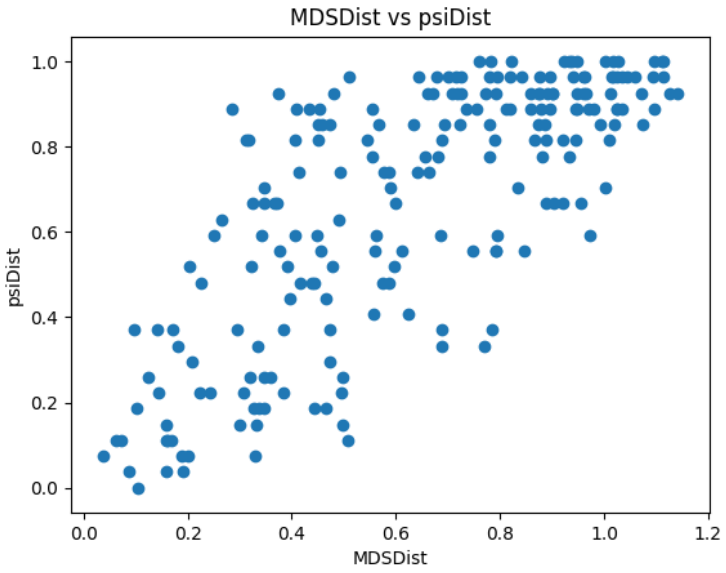




One can see that the MDS algorithm did a good job of organizing the sports in a way that most people would organize them, with water sports clustered together, ball sports closer to each other than non-ball sports, contact sports (football, hockey, and rugby) close together, and more unique sports further away from other sports. So yes, this graph does match my intuitions about how the items of this domain should be arranged on a coordinate plane.

Problem #5

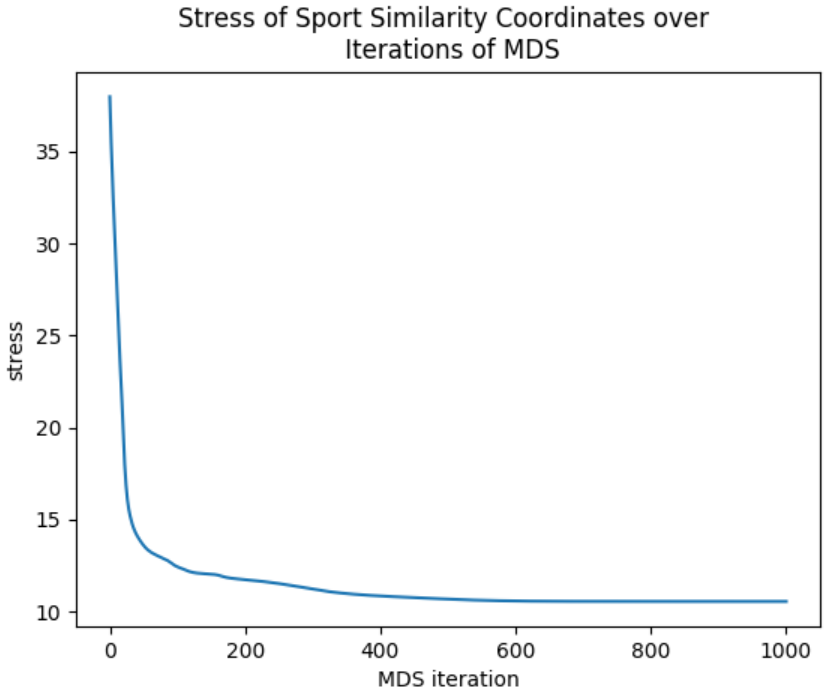
```python
def plotMDSvsPsi(i):

    # this function will plot the psi distances against the distances
    # we calculate with MDS

    p = executeMDS(i)
    MDSDist = []
    for j in range(0, len(p)):
        for k in range(j + 1, len(p)):
            d = dist(p[j], p[k])
            MDSDist.append(d)

    psiDist = []
    count = 1
    for key in psiDict:
        curr = psiDict[key]
        psiDist += curr[count:]
        count += 1

    #print(MDSDist)
    #print(psiDist)

    plt.xlabel('MDSDist')
    plt.ylabel('psiDist')
    plt.title('MDSDist vs psiDist')
    plt.scatter(MDSDist, psiDist)
    plt.show()

plotMDSvsPsi(1000)
```



From the plot above, we can see there's is a trend towards an y=x configuration. This is what we want, since our goal is for the distances produced by our MDS algorithm to match the psychological distances in the data, which ultimately lowers the stress. Because my graph generally follows this y=x configuration, I could classify it as a good graph. A bad graph would be one that doesn't follow the y=x pattern. This could include a graph with points scattered with no clear configuration, or a graph that follows a completely different configuration (for example: y=-x, y=1/x, y=2x, etc.).

Problem #6

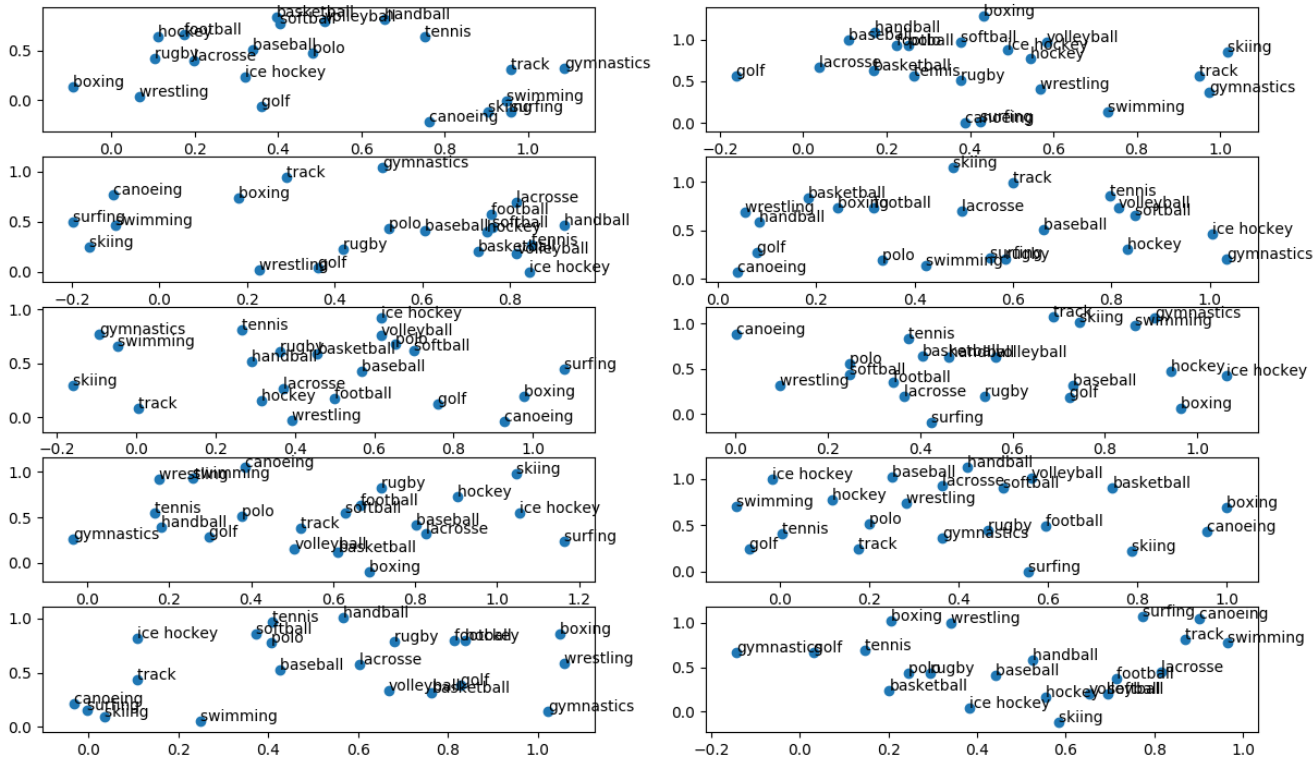
```
def executeMDSStress(i):

    # initialize an array to hold the stress values over time
    stresses = []

    # for i iterations, we will calculate the gradient and
    # adjust our coordinates accordingly, then calculate stress
    # and add it to our stresses array
    p = executeMDS(i)
    s = stress(p)
    stresses.append(s)

    # we now have the data to plot our stress values over time
    x = np.arange(0, i + 1)
    y = stresses
    plt.xlabel('MDS iteration')
    plt.ylabel('stress')
    plt.title('Stress of Sport Similarity Coordinates over \n Iterations of MDS')
    plt.plot(x, y)
    plt.show()
```



Using this plot, we can see that the stress decreases drastically in the first 100 iterations, and then continues to decrease until around the 600th iterations, where the decrease begins to become basically negligible. Using this graph, we can see that around 600 iterations is a safe number of iterations to use in order to minimize the stress as much as possible.

Problem #7

```
def makeSubplots(t, i):

    # this function will plot t subplots, each one representing a fresh trial
    # of our MDS algorithm (t must be divisble by 2)

    fig, sub = plt.subplots(t // 2, 2)

    # j will be our row index
    for j in range(0, t // 2):
        # k will be our column index
        for k in range(0, 2):

            # we generate a new vector of positions and run MDS
            p = executeMDS(i)

            # we create the plot for this trial
            x = []
            y = []
            for i in range(0, len(p)):
                x.append(p[i][0])
                y.append(p[i][1])
            sub[j][k].scatter(x, y)
            for i, sport in enumerate(sports):
                sub[j][k].annotate(sport, (x[i], y[i]))

    # once finished with all trials, we show all trials together
    plt.show()
```



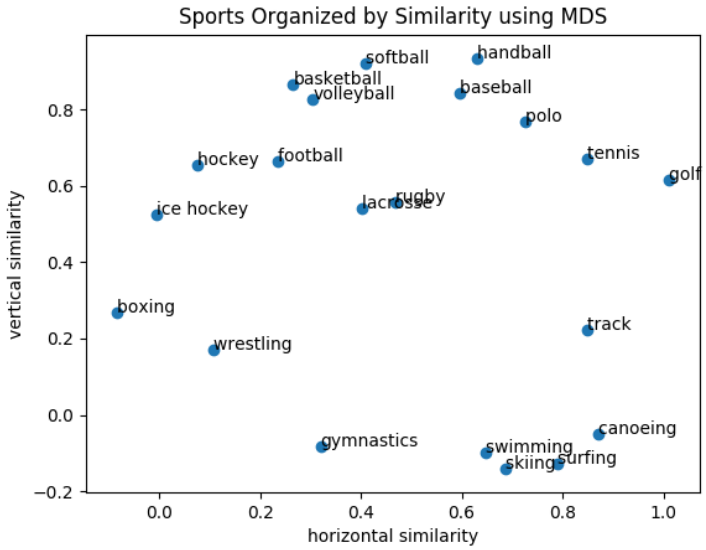
From the 10 graphs above, we can see that all the graphs are wildly different upon first glance. However, if we look closer, we see that some general patterns and arrangements stay consistently close (for example, water sports being clumped together, wrestling and boxing being close to each other, etc.). This happens because our points are placed randomly on the plane at the beginning of each run of MDS. When the algorithm starts causing the points to move around and decrease stress, those initial positions of the points are going to affect where they ultimately end up (since they don't need to travel any particular distance, they just need to move closer to their similar counterparts).

Problem #8

To find the best graph out of 10 trials, we have to run our algorithm 10 times and determine which one happened to produce the lowest stress. To do this, I've written this code that performs the 10 trials, checking at the end of each one if it has produced a stress that is lower than the previous best. I hold the lowest stress in a variable, and compare against it each trial. I also keep track of the best trial number, and the resulting vector of points of the best trial. This way, I can plot the best trial once I have completed all 10 trials.

```
def findBestTrial(t, i):

    # this function will run our MDS algorithm t times and use the
    # ending stress of each trial to determine which trial produced the
    # best graph

    # first we iniialize our variable that will help us keep track
    # of the best graph
    p = genPositions(21)
    best = stress(p)
    bestIndex = 0
    bestPositions = p

    # then we run our algorithm t times
    for k in range(1, t + 1):
        p = executeMDS(i)
        s = stress(p)

        # for each trial, we have to see if the trial's result is
        # better than our previous best result
        if (s < best):
            best = s
            bestIndex = k
            bestPositions = p

    print("Our best trial was trial", bestIndex)
    displayPlot(bestPositions)
```

For example, I ran this code, and it produced this graph and this output:




```
Our best trial was trial 2
Stress: 8.773470498816629
```