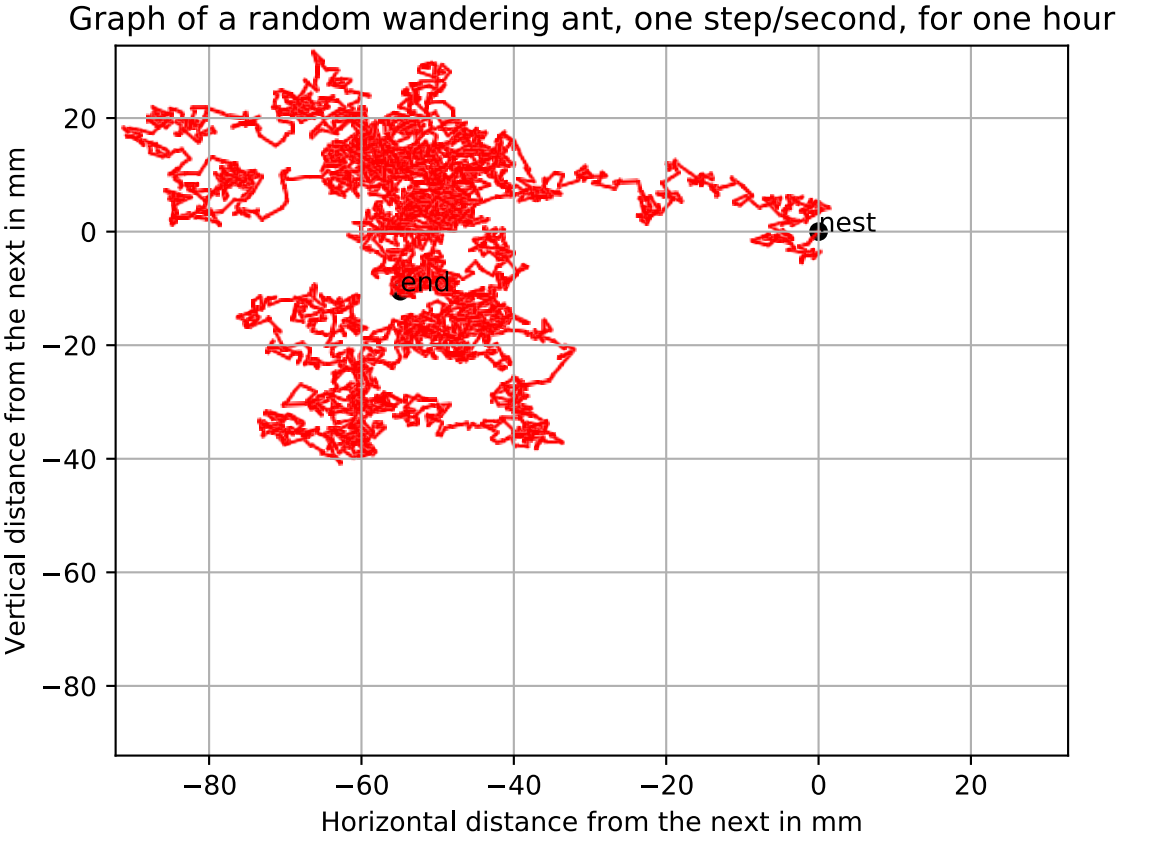
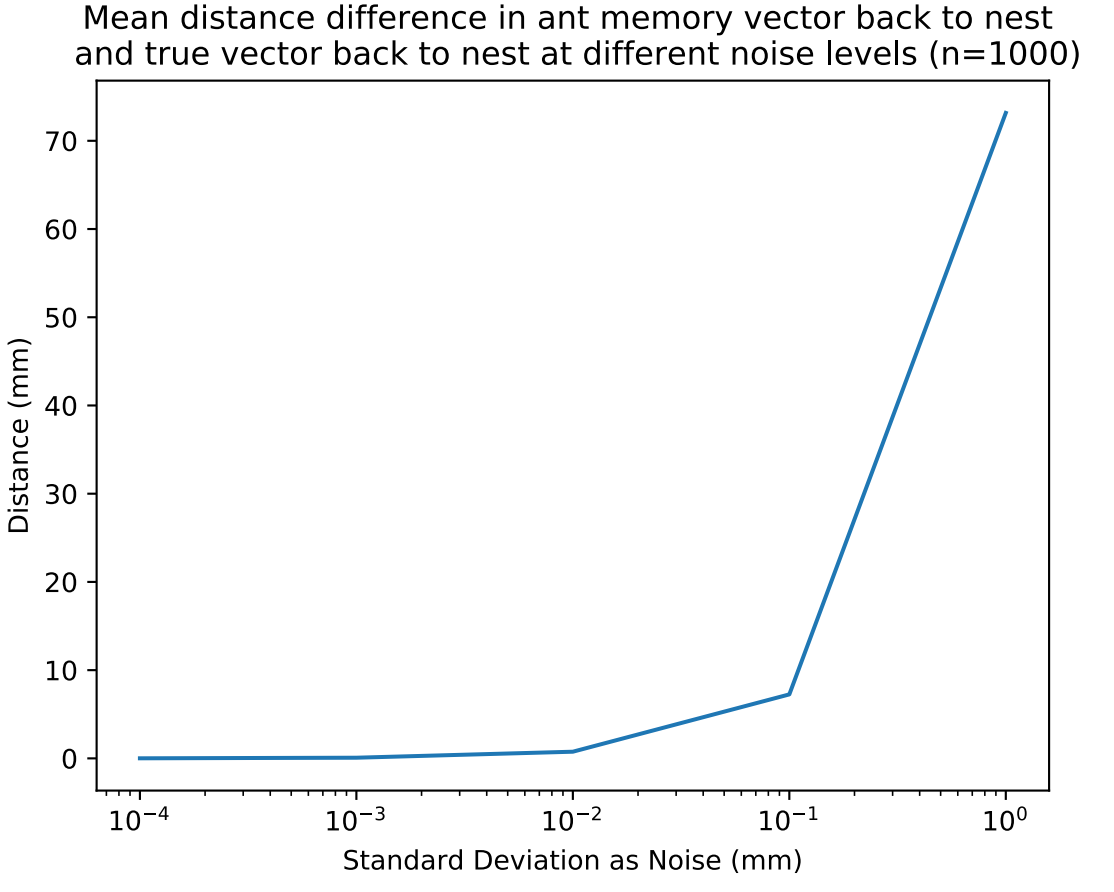
```
"""Written by Joseph Hayes for UC Berkeley CogSci 131 Assignment 3,
Spring 2020"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.cm as cmx
import math

# Problem 1a: Suppose that an ant wandered randomly by taking steps
(x,y), one per
# second, where at each ant step, x and y each come from a normal
distribution with a mean
# of 0 and a standard deviation of 1.0mm (assume this for all
questions below). Plot
# a trace of the ant's path over the course of an hour.
class PathIntegration():

    def distance(x1, y1, x2, y2):
        """Returns the Pythagorean distance between two points"""

        return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

    def getColorList(colors):
        """Color gradient function created with the help of:
        https://stackoverflow.com/questions/12487060/matplotlib-color-
according-to-class-labels/12487355
        by user https://stackoverflow.com/users/2444213/salomonvh """
        cmap = plt.cm.jet
        cmaplist = [cmap(i) for i in range(cmap.N)]
        cmap = cmap.from_list('Custom cmap', cmaplist, cmap.N)

        bounds = np.linspace(0, colors, colors + 1)
        norm = mpl.colors.BoundaryNorm(bounds, cmap.N)
        scalarMap = cmx.ScalarMappable(norm=norm,cmap=cmap)
        list = []
        for i in range(colors):
            list += [scalarMap.to_rgba(i)]

        return list

    def antRandomPath(randomFunction=np.random.normal, mean=0.0,
stdev=1.0, step=1, time=3600, scale=1, color='red', plot=True,
foodLoc=None, nest=[0,0], track=False):
        dataPoints = np.array([nest])
        # vectors = np.array([nest])
        timeAxis = list(range(1, time + 1, step))
        x1, y1 = nest
        shortestDist = 0
```


Graph of a random wandering ant, one step/second, for one hour
Vertical distance from the next in mm
Horizontal distance from the next in mm
nest
end
20
0
−20
−40
−60
−80
−80
−60
−40
−20
0
20

```
        if foodLoc is not None:
            x1, y1 = foodLoc
            shortestDist = PathIntegration.distance(foodLoc[0],
foodLoc[1], nest[0], nest[1])

        maxX, maxY = x1, y1
        minX, minY = x1, y1
        # colors = getColorList(time)
        if foodLoc is not None:
            plt.text(foodLoc[0], foodLoc[1], 'food')
            plt.scatter(foodLoc[0], foodLoc[1], color='black')
            color = 'blue'

        for a in timeAxis:
            # color = colors[a - 1]
            dx = randomFunction(mean, stdev)
            dy = randomFunction(mean, stdev)

            if plot == True:
                plt.arrow(x=x1 * scale, y=y1 * scale, dx=dx * scale,
dy=dy * scale, length_includes_head=True, head_width=.1, color=color,
rasterized=True)

            nestDistance = PathIntegration.distance(x1, y1, nest[0],
nest[1])
            x1 += dx
            y1 += dy

            if track is True and foodLoc is not None:
                if nestDistance <= 5.0:
                    return 1

            if track is False and foodLoc is not None:
                shortestDist = min(shortestDist, nestDistance)

            if x1 > maxX:
                maxX = x1
            if y1 > maxY:
                maxY = y1
            if y1 < minY:
                minY = y1
            if x1 < minX:
                minX = x1

        if plot == True:
            minX *= scale
            minY *= scale
            plt.ylim(min(minY, minX) - 1, max(abs(maxY), abs(maxX)) +
1)
            plt.xlim(min(minY, minX) - 1, max(abs(maxY), abs(maxX)) +
```

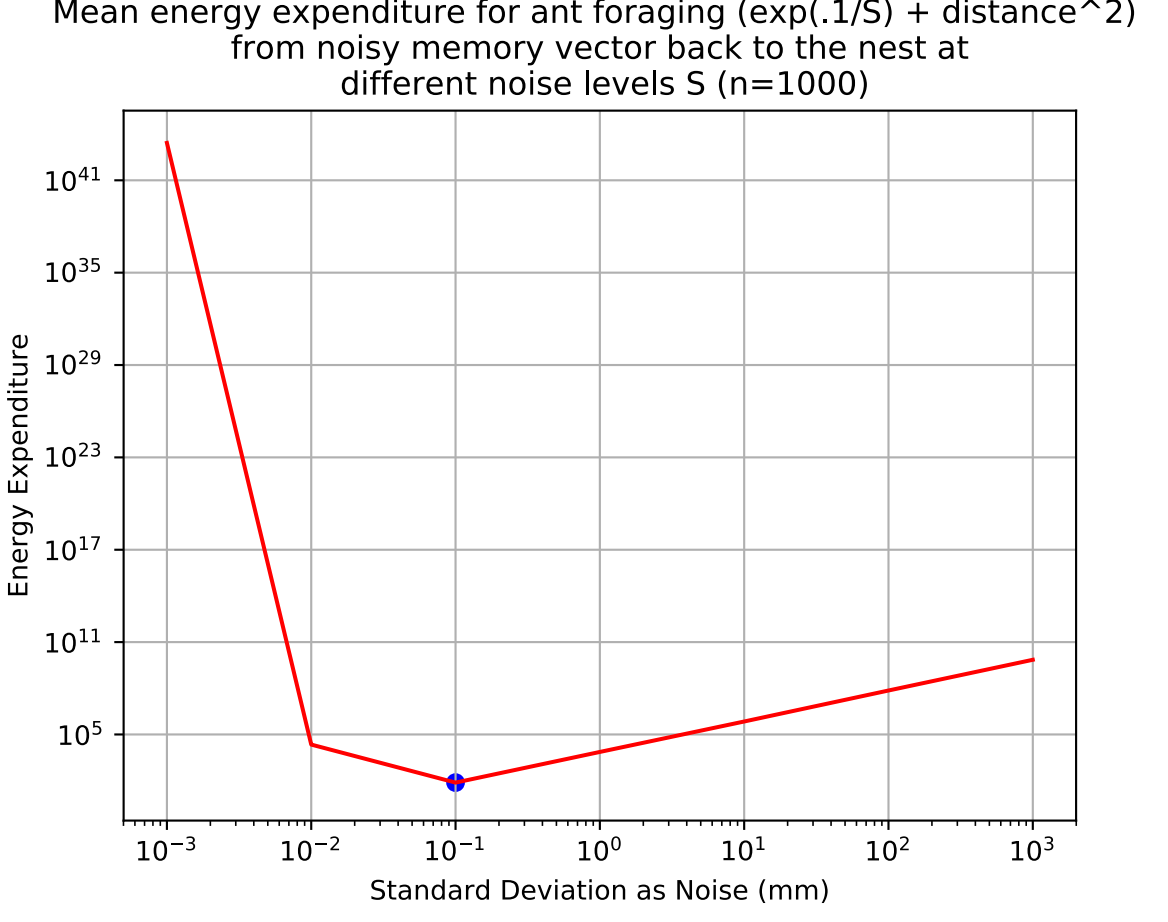
```
1)

            x, y = dataPoints.T
            plt.text(nest[0], nest[1], 'nest')
            plt.text(x1, y1, 'end')
            plt.title('Graph of a random wandering ant, one step/
second, for one hour')
            plt.xlabel('Horizontal distance from the next in mm')
            plt.ylabel('Vertical distance from the next in mm')
            plt.scatter(nest[0], nest[1], color='black')
            plt.scatter(x1, y1, color='black')
            plt.grid()
            plt.close()
            # plt.savefig('Assignment3_1a.pdf')
            # plt.show()

        if track == True:
            return 0
        if track == False and foodLoc is not None:
            # print(shortestDist)
            return shortestDist
        else:
            return (x1, y1)

PathIntegration.antRandomPath(plot=True)
#end problem 1a

# Problem 1b: Let's think about why ants need to perform path
integration. Suppose that instead
# of path integration, when an ant found food, it just continued to
wander with
# random steps until it got back to the nest. Using a simulation,
estimate the probability
# that an ant who finds food after 1 hour will make its way back to
within 5mm of the nest
# over the course of the next hour (note that if it comes within 5mm
of a nest, it stops).
# How many simulations do you need to run? Do the results show that
this is a good strategy?
# Why or why not?

# successes = 0
# numTrials = 10000
#
# for i in range(numTrials):
#     foodLocation = PathIntegration.antRandomPath(plot=False)
#     successes += PathIntegration.antRandomPath(plot=False,
track=True, foodLoc=[foodLocation[0], foodLocation[1]])
#
# print('P(finds nest, 1 hour)=', successes / numTrials * 100, '%', '#
```

```
samples:', numTrials)
"""Answer for 1b:
        P(finds nest, 1 hour)= 14.96 % # samples: 10000. After running
10,000 samples,
        the ant only gets within 5.0mm of the nest 14.96% of the time.
This is
        very poor strategy, it is extremely inefficient, the ant would
rarely make
        its way back to the nest with food. """
#end Problem 1b

# Problem 1c: If the ant searches for an hour, finds food, and then
searches for the nest
# by continuing to walk at random, on average, what is the closest
distance it will come to
# the nest over the course of the next hour? (Do not assume it stops
if it comes within 5mm)
# Find this average distance with a simulation.

# numTrials = 100
# shortestDist = []
#
# for i in range(numTrials):
#     foodLocation = PathIntegration.antRandomPath(plot=False) #first
hour
#     shortestDist += [PathIntegration.antRandomPath(plot=False,
track=False, foodLoc=[foodLocation[0], foodLocation[1]])] #second hour
# print(min(shortestDist))
# print(sum(shortestDist) / numTrials)
"""Answer for 1c:
        I am assuming that we are taking the shortest distance from
the ant at any step to the nest over the course
        of a search from food to the nest; this is one sample. I also
assume we do this over many trials, sum all of the shortest
        distances, and divide by the number of samples taken. I
calculated average distances using 100 trials
        5 times and got the following values in mm: [46.975, 44.964,
49.160, 47.394, 48.221].
        I interpret these average distances to be inefficiency (energy
or distance loss) from randomly
        searching for the nest. """


"""Problem 2"""

def antPathIntegration(noiseDev, time=3600, moveDev=1.0, nest=(0,0),
mean=0):
    """Returns the Euclidean distance between end two points of two
random paths, one path
        incorporates noise with every step along its path. """
```


Mean distance difference in ant memory vector back to nest
and true vector back to nest at different noise levels (n=1000)
Distance (mm)
0
10
20
30
40
50
60
70
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
Standard Deviation as Noise (mm)

```
    memoryPath = [nest]
    actualPath = [nest]
    actualX, actualY = nest
    noiseX, noiseY = nest

    for i in range(time):
        #ant moves
        deltaX = np.random.normal(mean, moveDev)
        deltaY = np.random.normal(mean, moveDev)
        actualX += deltaX #update point
        actualY += deltaY #update point

        actualPath.append((actualX, actualY)) #the actual move, no
noise

        #question: does next noise step go from actual or from last
noise step?
        noiseX += (deltaX + np.random.normal(mean, noiseDev)) #noisy
point + new noisy point
        noiseY += (deltaY + np.random.normal(mean, noiseDev)) #noisy
point + new noisy point
        memoryPath.append((noiseX, noiseY)) #the move plus memory
noise

    actX, actY = actualPath[-1]
    memX, memY = memoryPath[-1]

    return PathIntegration.distance(actX, actY, memX, memY)

# distance = antPathIntegration(1, time=3600)
# print(distance)

def pathIntegrationSimulation(noiseDev=[], samples=1000, time=3600):
    """Returns the mean distance of end points for all standard
deviations in
        noiseDev and the noiseDev list. """
    #return noiseDev for x-axis numbers and mean distances over all
samples for each noiseDev
    meanDist = [] #list for storing the mean distances for every
noiseDev

    for S in noiseDev:
        meanDist += [sum([antPathIntegration(S, time) for i in
range(samples)]) / samples]
    return noiseDev, meanDist

# noises, distances = pathIntegrationSimulation([1.0, 0.1, 0.01,
0.001, 0.0001], time=10)

# plt.semilogx(noises, distances)
```

```
# plt.ylabel('Distance (mm)')
# plt.xlabel('Standard Deviation as Noise (mm)')
# plt.title('Mean distance difference in ant memory vector back to
nest \n and true vector back to nest at different noise levels
(n=1000)')
# # plt.savefig('path_integration_deviation.pdf')
# plt.show()
"""Answer for problem 2:
    The distance from the next goes up logarithmically as noise
increases from .0001 to 1.
    The accuracy of the ant's memory is crucial to its survival."""


"""Problem 3a """
def pathIntegrationEnergy(noiseDev=[], samples=1000, time=3600):
    """Returns the mean energy expenditure of a foraging trip at exp(.
1/noiseDev)
        plus (distance to the nest)^2 over samples number of trials.
"""
    meanEnergy = [] #list for storing the mean distances for every
noiseDev

    for S in noiseDev:
        # print("Path finding energy:", S, np.exp(.1/S))
        meanEnergy += [sum([(np.exp(.1/S) + (antPathIntegration(S,
time=time))**2) for i in range(samples)]) / samples]
    return noiseDev, meanEnergy

# noises, energy = pathIntegrationEnergy([.001, .01, .1, 1, 10, 100,
1000], time=3600, samples=1000)
#
# plt.loglog(noises, energy, color='red')
# plt.scatter(.1, min(energy), color='blue')
# plt.grid()
# plt.title('Mean energy expenditure for ant foraging (exp(.1/S) +
distance^2) \n from noisy memory vector back to the nest at \n
different noise levels S (n=1000)')
# plt.xlabel('Standard Deviation as Noise (mm)')
# plt.ylabel('Energy Expenditure')
# plt.savefig('path_integration_energy.pdf')
# plt.close()

"""Problem 3b Answer:
    To use a smaller S and thus less noise as the ant moves, it
requires more energy
    expenditure per step. The minimum shows that there is a compromise
at about
    0.1mm noise. At this level of noise, the ant gets fairly close to
the nest
    after doing path integration resulting in a small d**2 energy
```


Mean energy expenditure for ant foraging (exp(.1/S) + distance^2)
from noisy memory vector back to the nest at
different noise levels S (n=1000)
Energy Expenditure
10^{41}
10^{35}
10^{29}
10^{23}
10^{17}
10^{11}
10^{5}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
Standard Deviation as Noise (mm)

```
expenditure and
    the exp(.1/S) energy is very low as well. Evolutionarily speaking,
ants likely
    carried this level of inherent accuracy genetically as core
knowledge. It has
    the highest return with minimum energy cost. Core knowledge of
this type is
    very important to worker ants because they tend to have short
lifespans (depending
    on the species) giving them little time to learn more efficient
methods of foraging. """

#end
```