Neal Ma
YSPA Problem Set #1
Dr. Michael Faison
2 July 2018

The code for each of the problems can be found at the bottom of this document. The raw python files will also be attached.

Problem 1. Leibnitz's Pi Approximation

In approaching this problem, I split it up into four distinct tasks: incrementing the denominator by two through every iteration, switching between adding positive and negative integers every iteration, comparing math's version of pi and the approximated version, and calculating the sum with various amounts of terms.

For the first task I wrote a simple expression that took in a counter value (starting at 0) and returned an odd integer (starting at 1 and increasing by 2 every iteration):

```
(counter * 2 + 1)
```

The second task was accomplished by a "switch" that changed from negative to positive and vice versa every iteration through the loop. This was applied to the above expression and added to a sum counter that started at 0:

```
Sum += negative * (counter * 2 + 1)
```

The third part was a percent error calculation using the percent error formula:

$$Percent\ Error\ = \frac{|Accepted\ Value - Calculated\ Value|}{Accepted\ Value} \times 100$$

Note* In this case, math.pi was the accepted value of pi despite it not being entirely accurate

And the final part was achieved by using a loop and just inputting several different powers of 10 and comparing them to the "accepted" value of pi. As the number of iterations increases by a factor of 10, the percent error decrease by around a factor of 10.

Number of Iterations	Accepted Value	Calculated Value	Percent Error
10	0.7853981633974483	0.760460	3.175238%
100	0. 7853981633974483	0.782898	0.318302%
1000	0. 7853981633974483	0.785148	0.031831%

Problem 2. Bubble and Comb Sorts

For the bubble and comb sort problem, I wanted to use the same function for both sorts, so I decided to combine the two into a function called "`sorts`" which took in three parameters: a list, a step distance, and a step increment factor. The list was to sort or to check if sorted, the step distance was to tell the distance between factors that are compared, and the step increment factor

was a constant that determined what the step distance would be the next iteration through the sorting algorithm. This function was called by either a bubble sort function or a comb sort function. If called by the bubble sort function, step distance and step increment factor were both set as 1. This means that only adjacent numbers were compared and it stayed this way until the list was completely sorted. When comb sort was called, step distance was set to the length of the list divided by the step increment factor (and cast to an integer) and step increment factor was set to 1.3 as per convention. At each iteration, the step distance decreased to the current step distance divided by the step increment. Once the step distance reaches one, the step distance is kept at one.

The comb sort was much more efficient than the bubble sort with the comb sort only going through the loop 4 times while the bubble sort typically took about 6-7 times. Time wise, the comb sort took about half of the time that the bubble sort took. The discrepancy between the time and number of iterations through the loop can be attributed to the fewer comparisons that the comb sort takes when the step distance is greater than one. The worst case scenario for the bubble sort would be an array in which it is sorted in decreasing order ($O(n^2)$ comparisons) while the best case would be an array in which the list is already sorted (O(n) comparisons).

Problem 3. Base Conversion

To convert sexagesimal to decimal was a relatively easy conversion. I first checked to see if the number was negative. If it was, I set a negative switch to -1.0 and if it was not, it was set to 1.0. I next initialized a decimal variable and set it to 0. To `decimal` I added the number of degrees, minutes divided by 60.0, and minutes divided by 3600.0.

To convert from decimal to sexagesimal was much more difficult. I took and stored the integer value of the angle passed in as the degree of the sexagesimal. Next, I subtracted the degree from the original angle, multiplied by 60, and followed the same procedure as before to find minutes. I then divided the remaining part by 60, and followed the same procedure with 60^2 to find seconds. The sign of the answer was determined using the same process from the sexagesimal to decimal

Significant figures in all cases were determined by hand calculation of the least significant digit and use of that as a rounding factor.

a) 11.9°
b) -60.519°
c) -8.7543°
d) 60°2’24”
e) 89°59’59.9”
f) -23°26’13.8”

Problem 4. Matrix Determinant

Instead of creating an algorithm that would find the determinant of a 2x2 or 3x3 matrix, I went straight to finding the determinant of an NxN matrix. I went about this recursively. The first step was to iterate through the numbers in the top row of the matrix. For each, I created a temporary matrix of all the digits not in the same row or column. I called the algorithm on this temporary

matrix to find the determinant of it. The base case was a 1x1 matrix in which the determinant is simply the value in the matrix. Because the function calls itself recursively, it can be applied to any size of a square matrix.

Problem 5. Adding Gravity to the Simulations

When I applied an acceleration due to gravity of -100.0 units/second^2, the ball followed a parabolic path downwards instead of its regular linear path. Different initial conditions changed the exact path that the ball took, but did not change the general shape of the path. One oddity that occurred was the rise in height of the ball. Every bounce, the highest point in its trajectory went higher and higher.

For the optional challenge, I made a function that took in the position of the ball and used its distance from the origin to determine the strength of gravity. This made the ball follow an elliptical path around the center. Once the gravity varied inversely with the distance to the origin, the elliptical orbit became much looser and the ball constantly sped up similar to the previous simulaitons.

```
#Problem Set 1 - Problem 1


from math import *


def leibnitz_approximation(terms):
    approximation = 0   #a sum to store the Leibnitz sum for pi/4
    negative = -1.0     #a switch to change the terms from negative to
postive and vice versa
    for counter in range(0, terms):
        negative *= -1.0    #switches the sign of the term
        approximation += negative / (counter * 2 + 1)
    return approximation


def compare(calculated, known, itterations):
    percent_error = abs(calculated - known) / known * 100   #finds the
percent error of the Leibnitz sum
    print 'The approximation with %d itterations was %f and had a percent
error of %04f%%' % (itterations, calculated, percent_error)


for i in range (1, 4):  #finds the sum at various amounts of iterations
    compare(leibnitz_approximation(10**i), pi/4, 10**i)
```

```
#Problem Set 1 - Problem 2


from datetime import datetime
from random import randint


def sorts(unsorted_list, distance, factor, iterations):
    swapped = False     #sets a switch to determine if the any swaps have
occured
    for counter in range (0, len(unsorted_list) - distance):
        if unsorted_list[counter + distance] < unsorted_list[counter]:  #if
the latter number is larger, swaps the two
            temp = unsorted_list[counter + distance]
            unsorted_list[counter + distance] = unsorted_list[counter]
            unsorted_list[counter] = temp
            swapped = True #tells if any swaps have occured that iteration
    if not swapped and distance == 1: #if no swaps have occured and compare
distance is one, ends the loop
        return [unsorted_list, iterations]
    else:   #lowers the step size if greater than one
        new_jump_size = int(distance / factor)
        if new_jump_size <= 1:
            return sorts(unsorted_list, 1, 1, iterations + 1)
        else:
            return sorts(unsorted_list, new_jump_size, factor, + iterations +
1)


def comb_sort(unsorted_list):    #calls a comb sort with a scale factor and
determined step size
    return sorts(unsorted_list, int(len(unsorted_list) / 1.3), 1.3, 0)


def bubble_sort(unsorted_list):  #calls a generic bubble sort
    return sorts(unsorted_list, 1, 1, 0)


def sort_times(unsorted_lists): #counts the total time and number of
iterations each sort takes on 1000 arrays of 10 random digits
    bubble_iterations = 0
    comb_iterations = 0
```

```
    start_time = datetime.now().microsecond
    for unsorted_list in unsorted_lists:
        bubble_iterations += bubble_sort(unsorted_list)[1]
    mid_time = datetime.now().microsecond
    for unsorted_list in unsorted_lists:
        comb_iterations += comb_sort(unsorted_list)[1]
    end_time = datetime.now().microsecond
    return [mid_time - start_time, end_time - mid_time, bubble_iterations,
comb_iterations]


matrix = [] #creates an array of 1000 arrays of 10 randomized digits to test
the sorting algorithms
for counter in range(0, 1000):
    temp_list = []
    for i in range(0, 10):
        temp_list.append(randint(1, 101))
    matrix.append(temp_list)


times = sort_times(matrix)
print times
print "The average time for the bubble sort was " + str(times[0] / 1000.0) +
" microseconds with an average of " + str(times[2] / 1000.0) + " iterations
per sort."
print "The average time for the comb sort was " + str(times[1] / 1000.0) + "
microseconds with an average of " + str(times[3] / 1000.0) + " iterations per
sort."
print "The comb sort was on average " + str(times[0] * 1.0 / times[1]) + "
times faster than the quick sort."
```

```
#Problem Set 1 - Problem 3


# -*- coding: utf-8 -*-
local_encoding = 'cp850'
deg = u'\xb0'.encode(local_encoding)



def sexagesimal_to_decimal(angle):
    negative = 1.0
    decimal = 0
    if angle[0] == "-": #saves if the given angle is negative or not
        negative = -1.0
        angle = angle[1:]
    places = angle.split('_')   #takes the given angle and changes it into an
array of digits to convert
    for i in range(0, len(places)):
        decimal += float(places[i]) / (60.0 ** i)
    decimal *= negative #makes the decimal negative if it was initially
    print decimal


def decimal_to_sexagesimal(angle):
    time = []
    negative = 1.0
    sexagesimal = ""
    if angle < 0:
        negative = -1.0
        angle *= -1.0
    time.append(int(angle)) #finds the degree of the angle
    angle -= int(angle)
    time.append(int(angle*60.0)) #find the arcminutes of the angle
    angle -= time[1] /  60.0
    time.append(angle*3600.0) #find the arcseconds of the angle
    time[0] *= negative
    time[0] = int(time[0])
```

```
    print str(time[0]) + deg + str(time[1]) + "\'" + str(time[2]) + "\""
#returns the angle in the correct format


sexagesimal_to_decimal("11_54")

sexagesimal_to_decimal("-60_31_10")

sexagesimal_to_decimal("-8_45_15.94")


decimal_to_sexagesimal(60.04)

decimal_to_sexagesimal(89.99999)

decimal_to_sexagesimal(-23.43715)
```

```
#Problem Set 1 - Problem 3


# -*- coding: utf-8 -*-
local_encoding = 'cp850'
deg = u'\xb0'.encode(local_encoding)



def sexagesimal_to_decimal(angle):
    negative = 1.0
    decimal = 0
    if angle[0] == "-": #saves if the given angle is negative or not
        negative = -1.0
        angle = angle[1:]
    places = angle.split('_')   #takes the given angle and changes it into an
array of digits to convert
    for i in range(0, len(places)):
        decimal += float(places[i]) / (60.0 ** i)
    decimal *= negative #makes the decimal negative if it was initially
    print decimal


def decimal_to_sexagesimal(angle):
    time = []
    negative = 1.0
    sexagesimal = ""
    if angle < 0:
        negative = -1.0
        angle *= -1.0
    time.append(int(angle)) #finds the degree of the angle
    angle -= int(angle)
    time.append(int(angle*60.0)) #find the arcminutes of the angle
    angle -= time[1] /  60.0
    time.append(angle*3600.0) #find the arcseconds of the angle
    time[0] *= negative
    time[0] = int(time[0])
```

```
    print str(time[0]) + deg + str(time[1]) + "\'" + str(time[2]) + "\""
#returns the angle in the correct format


sexagesimal_to_decimal("11_54")

sexagesimal_to_decimal("-60_31_10")

sexagesimal_to_decimal("-8_45_15.94")


decimal_to_sexagesimal(60.04)

decimal_to_sexagesimal(89.99999)

decimal_to_sexagesimal(-23.43715)
```

```
#Problem Set 1 - Problem 5


from visual import *


#initializes the ball and all of the walls
ball = sphere(pos=(0.1, 0.1, 2), radius=0.5, color=color.cyan)
wallR = box(pos=(6, 0, 0), size=(0.2, 12, 12), color=color.green)
wallL = box(pos=(-6, 0, 0), size=(0.2, 12, 12), color=color.green)
wallT = box(pos=(0, 6, 0), size=(12, 0.2, 12), color=color.blue)
wallB = box(pos=(0, -6, 0), size=(12, 0.2, 12), color=color.blue)
wallBack = box(pos=(0, 0, -6), size=(12, 12, 0.2), color=color.red)


gravity_factor = -10 #a factor for gravity (negative as objects accelerate
TOWARDS the source of gravity)


def find_gravity(orb): #returns the gravity at a point away from the origin
    grav_x = gravity_factor / (ball.pos.x ** 2)
    grav_y = gravity_factor / (ball.pos.y ** 2)
    grav_z = gravity_factor / (ball.pos.z ** 2)
    return vector(grav_x, grav_y, grav_z)


accel_due_to_gravity = find_gravity(ball) #finds the initial acceleration due
to gravity
#accel_due_to_gravity = -500.0 #constant gravity if you want it directed
downwards
ball.velocity = vector(2, 0, 0)
deltat = 0.005
t = 0
vscale = 0.1
varr = arrow(pos=ball.pos, axis=ball.velocity * vscale, color=color.yellow)
ball.trail = curve(color=ball.color)
scene.autoscale = False
while True:
    rate(100)
    ball.trail.append(pos=ball.pos)
    ball.pos = ball.pos + ball.velocity*deltat
```

```
    varr.axis = ball.velocity * vscale
    varr.pos = ball.pos
    t += deltat
    #ball.velocity.y += accel_due_to_gravity * deltat #used if gravity is
constant in magnitude and direction


    #changes velocity in each direction depending on the acceleration due to
position
    ball.velocity.x += accel_due_to_gravity.x * deltat
    ball.velocity.y += accel_due_to_gravity.y * deltat
    ball.velocity.z += accel_due_to_gravity.z * deltat


    #switches the trajectory of the ball if it passes a wall
    if ball.pos.x > wallR.pos.x or ball.pos.x < wallL.pos.x:
        ball.velocity.x *= -1.0
    if ball.pos.y > wallT.pos.y or ball.pos.y < wallB.pos.y:
        ball.velocity.y *= -1.0
    if ball.pos.z < wallBack.pos.z or ball.pos.z > 6:
        ball.velocity.z *= -1.0
```