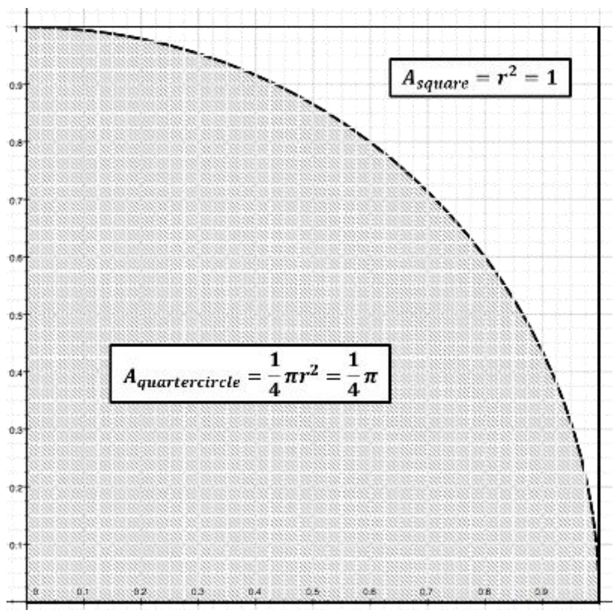
Homework Assignment (Problem Set) 4:

Note, Problem Set 3 directly focuses on Modules 7 and 8: Metaheuristic Algorithms and Monte Carlo Simulation

4 Questions

Rubric:
All questions worth 37.5 points
37.5 Points: Answer and solution are fully correct and detailed professionally.
25-37 Points: Answer and solution are deficient in some manner but mostly correct.
15-24 Points: Answer and solution are missing a key element or two.
1-14 Points: Answer and solution are missing multiple elements are significantly deficient/incomprehensible.
0 Points: No answer provided.

1. Perform Monte Carlo integration using R statistical programming or Python programming to estimate the value of π. To summarize the approach, consider the unit quarter circle illustrated in the figure below:



Generate N pairs of uniform random numbers (x, y), where $x \sim U(0,1)$ and $y \sim U(0,1)$, and each (x, y) pair represents a point in the unit square. To obtain an estimate of π, count the fraction of points that fall inside the unit quarter circle and multiply by 4. Note that the fraction of points that fall inside the quarter circle should tend to the ratio between the area of the unit quarter circle (i.e., ¼ π) as compared to area of the unit square (i.e., 1). We proceed step-by-step:

a) Create a function insidecircle that takes two inputs between 0 and 1 and returns 1 if these points fall within the unit circle.

code	def insidecircle(x, y): if (x**2+y**2) <= 1: return 1 return 0 # else

b) Create a function estimatepi that takes a single input N, generates N pairs of uniform random numbers and uses insidecircle to produce an estimate of π as described above. In addition to the estimate of π, estimatepi should also return the standard error of this estimate, and a 95% confidence interval for the estimate.

code	``` def estimatepi(N): result = {} xa = np.random.uniform(0,1,N) ya = np.random.uniform(0,1,N) pairsa = zip(xa, ya) inside_count = 0 for pair in pairsa: inside_count += insidecircle(*pair) result['pi_est'] = 4*inside_count/N result['pi_err'] = np.sqrt(inside_count/N * (1 - inside_count/N) / N) result['interval'] = conf_interval(result['pi_est'], result['pi_err'], .95) return result ```

c) Use estimatepi to estimate π for N = 1000 to 10000 in increments of 500 and record the estimate, its standard error and the upper and lower bounds of the 95% CI. How large must N be in order to ensure that your estimate of π is within 0.1 of the true value?

code

```
def iter_N(start, end, interval):
    df = pd.DataFrame()
    N = start
    goal = None
    while N <= end:
        result = estimatepi(N)
        print('current N:', N)
        print('\testimate of pi: {} \n\terror: {}\n\t95% confidence interval: {}'.format(*result.values()))
        if np.pi - result['interval'][0] <= .1 and \
            result['interval'][1] - np.pi <= .1:
            if not goal:
                goal = N

        else:
            goal = None

        df = df.append(result, ignore_index=True)
        N = N + interval

    return df, goal
```

solution

```
df_c, goal = iter_N(1000, 10000 , 500)
```

output

	pi_est	pi_err	interval
0	3.112000	0.052569	(3.008965575803036, 3.2150344241969644)
1	3.050667	0.043940	(2.9645440561913596, 3.136789277141974)
2	3.158000	0.036463	(3.0865333899726592, 3.2294666100273406)
3	3.134400	0.032943	(3.06983132116506, 3.1989686788349396)
4	3.142667	0.029968	(3.083928625631475, 3.201404707701858)
5	3.195429	0.027103	(3.142307245180067, 3.2485498976770755)
6	3.154000	0.025828	(3.1033776733683642, 3.2046223266316356)
7	3.160889	0.024278	(3.1133045362511833, 3.208473241526595)
8	3.132800	0.023310	(3.0871125373936, 3.1784874626064)
9	3.173091	0.021842	(3.1302809446320206, 3.2159008735497974)
10	3.134667	0.021262	(3.0929924115766987, 3.1763409217566343)

	11 3.131692 0.020454 (3.091603258131343, 3.1717813572532725) **12 3.138286 0.019655 (3.0997613981827645, 3.176810030388664)** **13 3.156800 0.018839 (3.1198755298112197, 3.1937244701887804)** **14 3.159000 0.018223 (3.1232822668160476, 3.194717733183952)** **15 3.126588 0.017924 (3.0914571576511816, 3.1617193129370533)** **16 3.130222 0.017393 (3.0961322572057766, 3.164312187238668)** **17 3.131789 0.016918 (3.0986303478954533, 3.1649485994729676)** **18 3.128000 0.016515 (3.0956296283221834, 3.160370371677817)** **ensure within .1: 4000**

d) Using the value of N you determined in part c), run estimatepi 500 times and collect 500 different estimates of π. Produce a histogram of the estimates and note the shape of this distribution. Calculate the standard deviation of the estimates – does it match the standard error you obtained in part c)? What percentage of the estimates lies within the 95% CI you obtained in part c)?

code	`print('\n\n----------\nd) collecting 500 at goal N={}'.format(goal))` `df_d = pd.DataFrame()` `for i in range(500):` `    result = estimatepi(goal)` `    df_d = df_d.append(result, ignore_index=True)` `print('std deviation', df_d.pi_est.std())` `perc_within = len(df_d[(df_d.pi_est >= interv[0]) & (df_d.pi_est <= interv[1])]) / len(df_d)` `print('Percent of esitmates falling within interval: {}'.format(perc_within))`
solution	**d) collecting 500 at goal N=4000** **std deviation 0.02597017064878577** **Percent of esitmates falling within interval: 0.92**

2. A salesperson in a large bicycle shop is paid a bonus if he sells more than 4 bicycles a day. The probability of selling more than 4 bicycles a day is only 0.40. If the number of bicycles sold is greater than 4, the distribution of sales as shown below. The shop has four different models of bicycles. The amount of the bonus paid out varies by type. The bonus for model A is $10; 40% of the bicycles sold are of this type. Model B accounts for 35% of the sales and pays a bonus of $15. Model C has a bonus rating of $20 and makes up 20% of the sales. Finally, a model D pays a bonus of $25 for each sale but accounts for only 5% of the sales. Develop a simulation model to calculate the bonus a salesperson can expect in a day.

Table

Number of Bicycles Sold	Probability
5	0.35
6	0.45
7	0.15
8	0.05

Code:

```
import numpy as np
from random import random

days = 500

# list of probabiliies for number of bikes sold
sales_probs = [.15]*4
more_than_4 = [.35, .45, .15, .05]
more_than_4 = [prob * .4 for prob in more_than_4]
sales_probs.extend(more_than_4)

# list of bonus values
bonus_vals = [10, 15, 20, 25]
bonus_probs = [.4, .35, .2, .05]

daily_sales = np.random.choice(
  list(range(1,9)),
  days,
  p=sales_probs
  )


def daily_bonus(n):
    if n <= 4:
        return 0

    bonuses = np.random.choice(
      bonus_vals,
      n,
      p=bonus_probs
      )

    return sum(bonuses)

bonus_history = [daily_bonus(n) for n in daily_sales]

print("mean bonus: ", np.mean(bonus_history))
```

solution **mean bonus: 32.43**

3. Michael is 24 years old and has a 401(k) plan through his employer, a large financial institution. His company matches 50% of his contributions up to 6% of his salary. He currently contributes the maximum amount he can (i.e., 6%). In his 401(k), he has three funds. Investment A is a large-cap index fund, which has had an average annual growth over the past 10 years of 6.63% with a standard deviation of 13.46%. Investment B is a mid-cap index fund with a 10-year average annual growth of 9.89% and a standard deviation of 15.28%. Finally, Investment C is a small-cap Index fund with a 10-year average annual growth rate of 8.55% and a standard deviation of 16.90%. Fifty percent of his contribution is directed to Investment A, 25% to Investment B, and 25% to Investment C. His current salary is $48,000 and based on a compensation survey of financial institutions, he expects an average raise of 2.7% with a standard deviation of 0.4% each year. Develop a simulation model to predict his 401(k) balance at age 60.

Assumptions: retires on 60th birthday. 12 contributions per year (paid monthly)

Code:

```
import numpy as np

salary = 48000
raise_mean = .027
raise_std = .004
age_current = 24
age_retire = 60
emp_contribution = .12
match = .5

return_wghts = { 'a': .5,
            'b': .25,
            'c': .25
            }
return_means = { 'a': .0663,
            'b': .0989,
            'c': .0855
            }
return_stds  = { 'a': .1346,
            'b': .1528,
            'c': .1690
            }
fund_values  = { 'a': 0,
            'b': 0,
            'c': 0
            }

salaries = []

def calc_contribution_return(fund_values, contribution, fund):
    contribution_part = return_wghts[fund] * contribution/12
    fund_return = np.random.normal(loc=return_means[fund], scale=return_stds[fund])
    print('\t\tfund {} return: {}'.format(fund, fund_return))
    annuitized = contribution_part*(((1 + (fund_return/12))**(12))-1)/(fund_return/12)
    return fund_values[fund]*(1+fund_return) + annuitized



for i in range(age_current, age_retire):
    year_contribution = salary * (emp_contribution * (1+match))
```

	`    for fund in fund_values.keys():` `        fund_values[fund] = calc_contribution_return(fund_values, year_contribution, fund)` `    raise_pct = np.random.normal(loc=raise_mean, scale=raise_std)` `    salary = salary * (1+raise_pct)` `    salaries.append(salary)` `    print('age: {}\n\tsalary: {}\n\tcontribution: {}\n\tnew act value: {}'.format(i, salary, year_contribution, sum(fund_values.values())))`
Solution:	**Sample output from final year:** **age: 59** **fund a return: 0.16283030372580448** **fund b return: -0.017867779815352 55** **fund c return: -0.04826594545427208** **salary: 123820.87047576507** **contribution: 21745.832985375917** **new act value: 2576743.812384866**

4. Develop a simulated annealing procedure in either R or Python to solve the following knapsack problem: (Note, this problem can be solved to optimality using integer programming; however, the focus of this question is on developing the simulated annealing method).

Maximize $12x_1 + 16x_2 + 22x_3 + 8x_4$

S.T. $4x_1 + 5x_2 + 7x_3 + 3x_4 \leq 14$

x_i ~ binary for all i

code	<pre>import numpy as np # start the values off at an extreme x1, x2, x3, x4 = 0, 0, 0, 0 z1, z2, z3, z4 = 0, 0, 0, 0 # number of epochs N=100 curr_sol = None def objective(x1, x2, x3, x4): return 12*x1 + 16*x2 + 22*x3 + 8*x4 def constraint(x1, x2, x3, x4): return 4*x1 + 5*x2 + 7*x3 + 3*x4 def evaluate(curr_sol, ys, zs): y1, y2, y3, y4 = ys z1, z2, z3, z4 = zs candidate_sol = objective(y1, y2, y3, y4) if constraint(y1, y2, y3, y4) <= 14: if curr_sol is None: # if no valid solution has yet been found return candidate_sol, y1, y2, y3, y4 diff = candidate_sol - curr_sol print("difference between candidate and current best", diff) if diff > 0: print("accept new") return candidate_sol, y1, y2, y3, y4 else: print("not more optimal. skip") return curr_sol, z1, z2, z3, z4 print('constraint violated. skip') return curr_sol, z1, z2, z3, z4 for i in range(N): # randomly reassign 1 value exec("%s = %d" % (np.random.choice(['x1', 'x2', 'x3', 'x4']), np.random.choice([0,1]))) curr_sol, z1, z2, z3, z4 = evaluate(curr_sol, (x1, x2, x3, x4), (z1, z2, z3, z4))</pre>

	print("final:\n\tx1: {}\n\t x2: {}\n\t x3: {}\n\t x4: {}\n\tobjective: {}".format(z1, z2, z3, z4, curr_sol))
Sample output & solution	**difference between candidate and current best -18 not more optimal. skip difference between candidate and current best 4 accept new constraint violated. skip difference between candidate and current best 0 not more optimal. skip difference between candidate and current best -8 not more optimal. skip difference between candidate and current best -8 not more optimal. skip difference between candidate and current best -8 not more optimal. skip difference between candidate and current best -30 not more optimal. skip final: x1: 1 x2: 0 x3: 1 x4: 1 objective: 42**