Sequence Risk Simulation and Calculations Code

To run the accumulation and decumulation 30-year simulations there were two python programs created. The first programmed, randomly generated 100,000 permutations using NumPy. NumPy is a library containing a collection of lists. Instead of using the typical lists in python NumPy was called upon as it runs faster allowing for us to generate a larger dataset. Additionally, NumPy has many functions attached with it that can be used to perform actions on the various arrays that are created when invoking NumPy. The second program focused on performing calculations with the set of rates. For instance, finding the final balance of an account when the account holder is depositing or withdrawing amounts. The point of these programs is to discover sequence risk and how the order of rates are important when investing money.

More specifically, in the case of our first program, the function permutations () was used to generate the permutations of a certain length n. In the case of our project n ended up being 30 to represent rates for 30 years. To effectively randomize the simulations attaching NumPy to the beginning of the permutations function like so, np.permutations(), made it where every time the permutation function was invoked a different library was used to generate the permutations making it completely random. To ensure that it was completely random, below was an example our group ran to test. In the test we decided to print one permutation through our program with the rates [1,2,3,4,5,6,7,8,9,10] and then ran it again to see if they were the same. If they were the same python generates the code pseudo randomly otherwise it doesn't invoke the same random library the same time but rather different making it completely random.

Example 1:

```
years = 10

rates = [1,2,3,4,5,6,7,8,9,10]

#runs through our loops for # of years specified

print(mixedRates) #mixedRates is a list of the new permutation
```

Result 1: [[5,9,8,1,3,4,10,7,6,2]]

Run the program a second time

Result 2: [[5,2,3,9,1,7,8,4,6,10]]

Our results proved to us that every time we run our program; we would get a different shuffled version of 100,000 rates. What that exactly means is ran the same 30 rates 100,000 times through our program twice that we would have two different lists of permutations for our results. Through this discovery, our group realized it was better for us to generate the permutations pseudo randomly. If the rates are generated pseudo randomly, then it makes it possible to compare distributions of different rates as the same permutations were used in both runs. To do this, we adjusted the code where we initialized a seed prior to using the random library. By initializing a seed, every time the code ran it went to the same location of the library to repeat the random order that we ran the time before. The fix in the code allowed us to keep our trials of different rates consistent with one another. Once the 100,000 permutations were generated, they were saved into a CSV file that could be observed by our group but also called into the next program we ran. To see the code used in our first program see *Appendix A.*

In the case of our second program, there were five functions created to complete the calculations we found necessary. These five functions were: calculate the final balance with a certain deposit strategy, finding the IRR associated with the final balance values, calculate the final balance with a certain withdrawal strategy, finding the year in which the account failed, and finding the perfect withdrawal amount for a given set of rates each year.

For the functions to work it required the user to manually input values. The first input was the CSV file of the 100,000 permutated rates which was done by opening the CSV file in this new code document. The other important inputs included a list for growth and for increase. The growth list was created to implement a strategy of investing a certain % increase. The growth list is the growth factors necessary for those increase for instance if I wanted my investment each year to increase by 3% then the growth list would be [1.03, 1.03…] which continues for the number of years being invested in our case it was 30. It was extremely important for the length of the growth list be equal to the number of years. Similarly, the increase list worked the same way however the inputs were of a dollar amount to increase an investment. The increase list would look something like [50, 100,…..] until you reached the 30 years of increasing. The reason for including these in a list was to allow the user flexibility to the investment strategy where they could have a growth that happen every few years where the

growth list would then become something like [1.03, 1.03, 1.03, 1.05, 1.05, 1.05, 1.07…]. With the inputs of these list it also required the user to tell the program what type of list of investments it would like to create. In our program, this choice is represented by an n where

If n = 0 the program would perform a calculation on a flat amount inputted each year

If n = 1 the program would perform a calculation on an additional increase from the flat amount each year

If n > 1 the program would perform a calculation on a percentage increase from the flat amount

The last two inputs the user had to provide was used for the decumulation examples. The user had to provide a beginning Balance to start withdrawing from and then a withdrawal amount to take out each year.

After all the inputs the user would run the program until it came up with the message "Program is complete". This message ensured us that the program indeed work where it was able to produce five different lists of the calculation results. These lists were then each converted into a CSV file that gave our group access to the finals values of each calculation performed. Our group was able to analyze these CSV files and create histograms for the accumulated values and probability of failure graphs for the decumulation examples. To ensure the program calculations were running properly our group did a check of the calculations in excel for the first five permutations of the 100,000. When these values matched, we believed it would be the case for all 100,000 permutations and that our functions were working properly. An important detail to this code is that the permutations generated did not include the best and the worst-case scenarios. Our group added these values to the dataset after the code ran to end with a total of 100,002 permutations for analysis. To see the code in detail for these calculations please see *Appendix B.*

Appendix A: Pseudo Permutation Maker

'"Retirement Simulations MQP Code"'

```
from itertools import permutations
import functools
import operator
import numpy as np
import random
import csv

#INPUTS
years = 10
permutations = 1
accumList = []
deccumList = []
rates = [31.49, -4.38, 21.83, 11.96, 1.38, 13.69, 32.39, 16, 2.11, 15.06, 26.46, -37, 5.49, 15.79, 4.91, 10.88, 28.68, -22.1, -11.89, -9.1, 21.04, 28.58, 33.36, 22.96, 37.58, 1.32, 10.08, 7.62, 30.47, -3.1, 31.69, 16.61, 5.25, 18.67, 31.73, 6.27, 22.56, 21.55, -4.91, 32.42, 18.44, 6.56, -7.18, 23.84, 37.2, -26.47, -14.66, 18.98, 14.31, 4.01, -8.5, 11.06, 23.98, -10.06, 12.45, 16.48, 22.8, -8.73, 26.89, 0.47, 11.96, 43.36, -10.78, 6.56, 31.56, 52.62, -0.99, 18.37, 24.02, 31.71, 18.79, 5.5, 5.71, -8.07, 36.44, 19.75, 25.9, 20.34, -11.59, -9.78, -0.41, 31.12, -35.03, 33.92, 47.67, -1.44, 53.99, -8.19, -43.34, -24.9, -8.42, 43.61, 37.49, 11.62, 44, -15, -10, 15] #This is a list of rates pulled from the S&P 500 historic values

#a loop creating a random permutation for a certain number of trials for our accumulating list
def main():
    randomRates = []
    index = 0
    while index < years:
        position = np.random.randint(len(rates))
        rate = rates[position]
        randomRates.append(rate)
        rates.pop(position)
        index += 1
```

```
    i = 0
    mixedRates = []
    while i < permutations: #run the loop from i to m number of possibilities
        accPerm = list(np.random.permutation(years)) #generate a random permutation of length n of type
list
        if accPerm in accumList: #check to see if permutation is already in allList
            pass
        else:
            i+=1 #going to the next iteration/permutation
            ratesPerm = change_rates(accPerm, randomRates)
            mixedRates.append(ratesPerm)
            print(mixedRates)

    with open('MQP_Flat_Rates_10_Years_Trial_1.csv', 'w', newline='') as b:
        writer = csv.writer(b)
        writer.writerows(mixedRates)

    print("Program has finished")

def change_rates(accPerm, randomRates):
    """Converts the permutation list of 0-n into a list of rates"""
    permRates = []
    for element in accPerm:
        permRates.append(randomRates[element])
    return permRates

if __name__ == "__main__":
    main()
```

Appendix B: Simulation Calculations

'''Retirement Simulations MQP Code'''

```
from itertools import permutations
import functools
import operator
import numpy as np
import csv

#ACCUMULATION INPUTS

deposit = 1000
n = 0 #determines which accumulation strategy to run

#n = 0 is a flat amount inputed each year
#n = 1 is an increase in amount by an additional flat amount
#n > 1 is an increase in amount by percentage

accumList = []
IRR = []

#DECUMULATION INPUTS
beginningBalance = 50000
withdrawal = 5000
deccumList = []
failureYear = []

#OTHER INPUTS
#growth = [1.00, 1.05, 1.05, 1.05, 1.05, 1.05, 1.08, 1.08, 1.08, 1.08, 1.08, 1.11, 1.11, 1.11, 1.11, 1.11,
1.14, 1.14, 1.14, 1.14,
        #1.14, 1.17, 1.17, 1.17, 1.17, 1.17, 1.20, 1.20, 1.20, 1.20] #The factors in which an accumulation
fund can increase by a percentage
```

```
growth = [1.00, 1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00, 1.00,1.00,1.00,1.00,1.00,1.00,1.00,
        1.00, 1.00,1.00,1.00, 1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00,1.00, 1.00,1.00,1.00,
        1.00,1.00,1.00,1.00,1.00,1.00,1.00, 1.00,1.00,1.00,1.00,1.00,1.00,1.00,
        1.00, 1.00,1.00]


increase = [0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700,
          750, 800, 850, 900, 950,1000, 1050, 1100, 1150, 1200, 1250, 1300, 1350, 1400, 1450] #The factors in which an accumulation fund can increase by a certain amount


#a loop creating a random permutation for a certain number of trials for our accumulating list
def main():
   "Open the file of 100,000 rates and then performs the calculation functions resulting in a saved CSV"
   yearly_rates = []
   with open('MQP_50_Years_Geo_Mean_Set_10.csv','r') as data: #opens the file
      for line in csv.reader(data):
         yearly_rates.append(line)
   PWA = []
   for ratesPerm in yearly_rates:
      ratesPerm = convert_to_float(ratesPerm)
      list_of_deposits = deposits(n,deposit,growth)
      totalWithDeposit = calc_total_with_deposit(ratesPerm, list_of_deposits)
      calculatedIRR = compute_IRR(ratesPerm, list_of_deposits)
      totalWithWithdrawal = calc_total_with_withdrawal(ratesPerm, beginningBalance)
      accumList.append(totalWithDeposit)
      print(type(accumList))
      deccumList.append(totalWithWithdrawal)
      print(type(deccumList))
      listOfFailures = find_year_of_failure(ratesPerm, beginningBalance)
      failureYear.append(listOfFailures)
      print(type(failureYear))
      IRR.append(calculatedIRR)
```

```
        print(type(IRR))
        perfectRate = compute_perfect_withdrawal_amount(ratesPerm, beginningBalance)
        PWA.append(perfectRate)
        print(type(PWA))

    '''with open('Accumulation_Values_50_Years_Geo_Mean_Set_10.csv', 'w', newline='') as c:
        writer = csv.writer(c, delimiter=';')
        writer.writerow(accumList)
    with open('Decumulated_Values_50_Years_Geo_Mean_Set_10.csv', 'w', newline='') as d:
        writer = csv.writer(d, delimiter=';')
        writer.writerow(deccumList)
    with open('Year_of_Failure_Values_50_Years_Geo_Mean_Set_10.csv', 'w', newline='') as e:
        writer = csv.writer(e)
        writer.writerows(failureYear)
        e.close()
    with open('IRR_Values_50_Years_Geo_Mean_Set_10.csv', 'w', newline='') as f:
        writer = csv.writer(f, delimiter= ';')
        writer.writerow(IRR)
    with open('Withdrawal_Values_50_Years_Geo_Mean_Set_10.csv', 'w', newline='') as g:
        writer = csv.writer(g, delimiter=';')
        writer.writerow(PWA)
        g.close()

    print("Program has finished")'''

def deposits(n,depositAmount,growth):
    """Determines the depositing strategy and creates it in a list of 30 years"""
    deposits = []
    if n == 0:
        i = 0
        while i < len(growth):
```

```
            deposits.append(depositAmount) #adds the flat deposit to a list
            i+= 1
    elif n == 1:
        total = depositAmount
        for value in increase: #steps in the list of flat amount increases
            total = total + value
            deposits.append(total) #adds the increasing deposits to a list
    else:
        total = depositAmount
        for rate in growth: #steps in the list of growth percentages
            total = total * rate
            deposits.append(total) #adds the percentage increasing deposits to a list

    return deposits

def convert_to_float(ratePerms):
    """Converts the rates into a workable list changing the elements to a float"""
    convertedRates = []
    for element in ratePerms: #steps through each element in the list of rates
        convertedRates.append(float(element))
    return convertedRates

def calc_total_with_deposit(permRates, list_of_deposits):
    """Calculates the resulting total based on the depositing strategy and list of perm rates"""
    totalAccumulation = []
    sum_of_totals = 0
    index = 0
    while index < len(permRates):
        totalAccumulation = permRates[index:] #moves through the rates list and starts at new beginning
position
        total = list_of_deposits[index]
        for rate in totalAccumulation: #adjust the rate to its new accumulated value
```

```
        total = total * ((rate/100)+ 1)
    sum_of_totals  = sum_of_totals + total
    index += 1

  return sum_of_totals



def calc_total_with_withdrawal(permRates, beginningBalance):
  """Calculates the balance in the retirement fund while withdrawing money each time from the list of perm rates"""
  remainingbalance = 0
  total = beginningBalance #sets the beginning balance
  for rate in permRates:
    withdrawal = 0.10 * total #only include this when you want to take out a percentage each year
    total = (((rate/100) + 1) * total) - withdrawal #calculates what is left in balance after withdrawal
  remainingbalance = remainingbalance + total #determines what is left in the account at the end of 30 years


  return remainingbalance


def find_year_of_failure(permRates, beginningBalance):
  """Determines the year in which the account value drops below 0 (i.e fails)"""
  remainingBalance = 0
  total = beginningBalance #sets the beginning balance
  failures = [] #creates a list to keep track of accounts that fail
  for rate in permRates:
    withdrawal = 0.10 * total #only include this when you want to take out a percentage each year
    total = (((rate / 100) + 1) * total) - withdrawal
    if total < 0: #determines if the account failed
      failures.append(permRates.index(rate)) #records the year in which the account failed
    else:
      pass
```

```
    return failures

def compute_IRR(permRates, list_of_deposits):
    """Computes the IRR for the calculated accumulation fund"""
    totalAccumulation = []
    sum_of_totals = 0
    index = 0
    while index < len(permRates):
        totalAccumulation = permRates[index:]
        total = list_of_deposits[index]
        for rate in totalAccumulation: #compute the accumulation values
            total = total * ((rate/100)+ 1)
        sum_of_totals = sum_of_totals + total
        futurevalue = sum_of_totals * -1 #creates the future value to be negative
        index += 1
    list_of_deposits.append(futurevalue) #adds the final value
    IRR = np.irr(list_of_deposits) #computes the IRR for that accumulation trial
    return IRR

def compute_perfect_withdrawal_amount(permRates, beginningBalance):
    """Determines the perfect withdrawal amount for a decumulation strategy"""
    PWR = []
    listOfRates = []
    results = []
    i = 0
    while i < len(permRates):
        listOfRates = permRates[i:] #gives the starting position of rates
        total = beginningBalance
        for rate in listOfRates:
            total = total * ((rate/100)+ 1)
            PWR.append(total) #adds the perfect withdrawal rate
```

```
        results.append(PWR[-1]) #adds the first element of the PWR list
        i += 1
    finalTotal = (sum(results[1:])/beginningBalance) + 1
    perfectWithdrawalAmount = results[0] / finalTotal #calculates the perfect withdrawal amount

    return perfectWithdrawalAmount

if __name__ == "__main__":
    main()
```