# Runtime-efficient Serial Computation of the Pearson Correlation Coefficient

Jefferson Louise Basilio[1]*

**Abstract**
This exercise studied how much time it would take for a program to compute the Pearson Correlation Coefficient of one vector and columns of a matrix in a serial way. It also tested the difference it would make in utilizing the CPU cache via transposing the matrix. Initially, the method was to fetch the values a row-wise, and then column-wise. Next was to do both orientations plus built-in Python libraries. The results showed that fetching integers column-wise made the program slightly faster. However, adding Python built-in libraries significantly decreased the average runtime of all inputs. These results proves that utilizing CPU cache and built-in libraries can handle relatively larger inputs and reduce runtimes of a serial program.

[1] *Institute of Computer Science, University of the Philippines Los Baños*
***Corresponding author**: jcbasilio1@up.edu.ph

## Contents



## Introduction

The Pearson Correlation Coefficient is a measurement of a linear correlation of two variables which has a range of -1 to +1, inclusive. Having a 0 value means no correlation. Meaning, there is no linear association between the two variables. Having a positive value means there is a positive association - there is a linear proportion existing between the two variables. Positive association means Y increases as X does. Otherwise there is a negative association, which means the two variables inhibit a indirect proportion - Y decreases as X increases[1]. In this exercise, the students are tasked with writing a program which calculates the Pearson Correlation Coefficient of a randomly generated vector and the columns of a randomly generated matrix. Asking for an integer input $n$, the program will generate an $n$ x $n$ matrix and an $n$-sized vector. The given minimum value of $n$ is 100, while the maximum is 20,000.

## Objectives

The objective of this exercise is to determine the time consumed by serially calculating the Pearson Correlation Coefficient of vectors of different sizes. Moreover, this will serve as baseline for further exercise topics which is computing using *threads*, *in parallel* or using *sockets*

## 1. Methodology

The specifications of the machine that will be used are the following:

- Processor - Intel Core i7-7700HQ
- Memory - 8GB DDR4 2400Mhz

Additionally, processes are minimized including browsers, CPU monitors, text editors, and other background services. Only the terminal is left open. The formula for the Pearson Correlation Coefficient is given by the following:

Given an m x n matrix X with m rows and n columns and a m x 1 vector y, a 1 x n vector r holds the Pearson Correlation Coefficient of the columns in X and y, such that

$$r(j) = \{m[X_j y] - [X]_j[y]\} / \{[m[X^2]_j - ([X])^2][m[y^2] - ([y])^2]\}^{1/2} \quad (1)$$

For this exercise, there will be three main observations, namely:

1. Row-major computation with *numpy*

2. Column-major computation with *numpy*
3. Column-major computation using pure built-in functions, for additional information

### 1.1 Row-major with *numpy*

For this part, after getting then randomizing *n* x *n* matrix and the *n*-sized vector, the *n* x *n* matrix is transposed before being plugged into the function *pearson_cor*. This is to force a row-major computation in accordance to *numpy*'s function *dot()*, which calculates all the dot products of the matrix's transposed columns and the *n*-sized vector. The core code for this would be below:

```
def pearson_cor(matrixX, vectorY, mRow,
    nColumn):
  vectorV = []
  matrixLen = len(matrixX)
  Xjy = numpy.dot(matrixX, vectorY) #Get the
      dot product of two vectors
  XSummation = numpy.sum(matrixX, axis=1)
  y = sum(vectorY)
  xSquaredInside = numpy.sum(matrixX ** 2,
      axis=1)
  ySquaredInside = matrixLen *
      sum(map(lambda y: y * y, vectorY))
  ySquared = y ** 2

  for j in range(mRow):
    mxjy = matrixLen * Xjy[j]
    xjy = XSummation[j] * y
    vectorV.append((mxjy - xjy) /
        (((matrixLen * xSquaredInside[j] -
        XSummation[j] ** 2) * (ySquaredInside
        - ySquared)) ** 0.5))
  return vectorV

''' Main program '''
nInput = int(input("Input n: "))
matrixX = initializeNxN(nInput)
    #Randomizes nxn matrix
vectorY = initializeOneVector(nInput)
    #Randomizes n-sized vector
matrixX =
    numpy.transpose(numpy.array(matrixX))
    #Transpose
vectorY = numpy.array(vectorY)
time_before = time.time()
returnVal = pearson_cor(matrixX, vectorY,
    nInput, nInput)
time_after = time.time()
```

### 1.2 Column-major with *numpy*

The difference of this to the previous observation was to not transpose the randomized matrix and instead just calculate the dot products directly column-wise. The code for this would be below:

```
def pearson_cor(matrixX, vectorY, mRow,
    nColumn):
  vectorV = []
  matrixLen = len(matrixX)
  Xjy = sum([i*j for i,j in
      zip(matrixX,vectorY)])  #Calculate
      directly column-wise
  XSummation = numpy.sum(matrixX, axis=0)
  y = sum(vectorY)
  xSquaredInside = numpy.sum(matrixX ** 2,
      axis=0)
  ySquaredInside = matrixLen *
      sum(map(lambda y: y * y, vectorY))
  ySquared = y ** 2

  for j in range(mRow):
    mxjy = matrixLen * Xjy[j]
    xjy = XSummation[j] * y
    vectorV.append((mxjy - xjy) /
        (((matrixLen * xSquaredInside[j] -
        XSummation[j] ** 2) * (ySquaredInside
        - ySquared)) ** 0.5))
  return vectorV

''' Main program '''
nInput = int(input("Input n: "))
matrixX = initializeNxN(nInput)
vectorY = initializeOneVector(nInput)
matrixX = numpy.array(matrixX)
vectorY = numpy.array(vectorY)
time_before = time.time()
returnVal = pearson_cor(matrixX, vectorY,
    nInput, nInput)
time_after = time.time()
```

### 1.3 Column-major with pure built-in functions

This observation is just an additional information. Below is the code of what a column-major would be without the use of the library *numpy*

```
def columnWiseMatrix(matrixX):
  return [list(i) for i in zip(*matrixX)]

def summationRowGivenJTimesY(matrixCol,
    vectorY, lenMatrixCol):
  return sum(map(lambda x, y: x * y,
      matrixCol, vectorY)) * lenMatrixCol

def summationVectorSquared(vector):
  return sum(map(lambda x: x * x, vector))

def pearson_cor(matrixX, vectorY, mRow,
    nColumn):
  vectorV = []
  matrixCol = columnWiseMatrix(matrixX) #
      Transpose matrix rows to columns
  y = sum(vectorY)
  matrixLen = len(matrixX)
  ySquaredInside =
```

```
    summationVectorSquared(vectorY) *
    matrixLen
  ySquared = y ** 2

  for j in range(mRow):
    mxjy =
        summationRowGivenJTimesY(matrixCol[j],
        vectorY, matrixLen)
    x = sum(matrixCol[j])
    xjy = x * y
    mXSquared =
        summationVectorSquared(matrixCol[j])
        * matrixLen
    xSquared = x ** 2
    vectorV.append((mxjy - xjy) /
        (((mXSquared - xSquared) *
        (ySquaredInside - ySquared)) ** 0.5))
  return vectorV

''' Main program '''
nInput = int(input("Input n: "))
matrixX = initializeNxN(nInput)
vectorY = initializeOneVector(nInput)
time_before = time.time()
unnecessaryReturn = pearson_cor(matrixX,
    vectorY, nInput, nInput)
time_after = time.time()
```

## 2. Results and Discussion

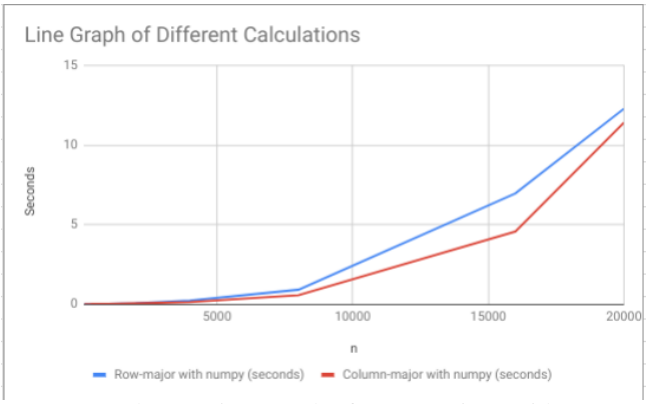


**Figure 1.** The run-time graph of computations with *numpy*

The results are laid down by the corresponding graphs.

### 2.1 Observation 1

The maximum $n$ that can be handled by the program on the machine is 30,000. Beyond that, the OS hangs and will not respond.

### 2.2 Observation 2

The row-major observation performed slightly better than the column-major on inputs under 1000. However, after hitting the 1000 mark, the column-major constantly performed better throughout.

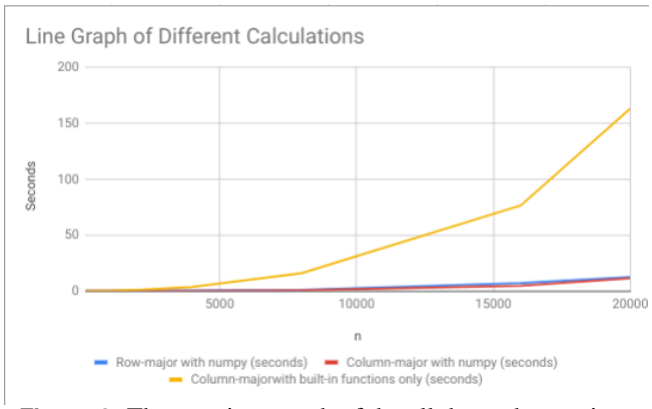


**Figure 2.** The run-time graph of the all three observations

### 2.3 Observation 3

The same program was modified to show how it will perform without the *numpy* library. Shown in *Figure 2*, as the size grows, it bears worse results. Starting from the first input ($n = 100$), it performed worse relatively from the first two constantly.

## 3. Conclusion

Judging from the findings, using dedicated libraries specializing in specific fields(*numpy* for data sets, for example) results in overall better improvement. However, the most important thing that can be inferred from this exercise is that careful and thorough analysis of the problem merits favorable results. For example, the improvement of utilizing the CPU cache via doing more *cache-hits* than *cache-misses* by computing column-wise may not be seen from smaller inputs, but it is evident that the improvement stayed constant in the long run.

## 4. List of Collaborators

1. Jefferson Louise Basilio
   *Nothing follows*

## 5. References

1. Pearson Product-Moment Correlation(2018). Retrieved from https://statistics.laerd.com/statistical-guides/pearson-correlation-coefficient-statistical-guide.php

## 6. Appendices

Code of Row-major:

```
import numpy

#Returns a vector v
def pearson_cor(matrixX, vectorY, mRow,
    nColumn):
  vectorV = []
  matrixLen = len(matrixX)
```

```
  Xjy = numpy.dot(matrixX, vectorY)
      #Get the all the dot product of the
      vectors of matriX against the vectorY
  XSummation = numpy.sum(matrixX, axis=1)
      #Compute the summation of matriX in
      row, given by axis=1
  y = sum(vectorY)
      #Get the sum of vectorY
  xSquaredInside = numpy.sum(matrixX ** 2,
      axis=1) #Get the squared summation of
      rows in matrix
  ySquaredInside = matrixLen *
      sum(map(lambda y: y * y, vectorY))
      #Get the squared summation of elements
      of vectorY multiplied by the size of
      matrix
  ySquared = y ** 2
      #Get the square of summation of vectorY

  for j in range(mRow):
    mxjy = matrixLen * Xjy[j]
        #Select the Xjy from the list of
        computed dot products. multiplied by
        the size of matrix
    xjy = XSummation[j] * y
        #Select the x from the summations of
        rows of Matrix, multiplied by the
        summation of vectorY
    vectorV.append((mxjy - xjy) /
        (((matrixLen * xSquaredInside[j] -
        XSummation[j] ** 2) * (ySquaredInside
        - ySquared)) ** 0.5)) #Perform
        Pearson operation
  return vectorV

''' Main program '''
nInput = int(input("Input n: "))
matrixX = initializeNxN(nInput)
    #Initializes the nxn matrix
vectorY = initializeOneVector(nInput)
    #Initializes the n-sized vector
matrixX =
    numpy.transpose(numpy.array(matrixX))
    #Converts to numpy array then transposes
vectorY = numpy.array(vectorY)
    #Converts to numpy array
time_before = time.time()
returnVal = pearson_cor(matrixX, vectorY,
    nInput, nInput)
time_after = time.time()
print("Time elapsed: ", time_after -
    time_before)
```

Code of column-major:

```
import numpy

def pearson_cor(matrixX, vectorY, mRow,
    nColumn):
  vectorV = []
  matrixLen = len(matrixX)
  Xjy = sum([i*j for i,j in
      zip(matrixX,vectorY)]) #Get the all
      the dot product of the vectors of
      matriX against the vectorY, performed
      column-wise
  XSummation = numpy.sum(matrixX, axis=0)
      #Compute the summation of matriX in
      columns, given by axis=0
  y = sum(vectorY)
      #Get the sum of vectorY
  xSquaredInside = numpy.sum(matrixX ** 2,
      axis=0) #Get the squared summation of
      columns in matrix
  ySquaredInside = matrixLen *
      sum(map(lambda y: y * y, vectorY))
      #Get the squared summation of elements
      of vectorY multiplied by the size of
      matrix
  ySquared = y ** 2
      #Get the square of summation of vectorY

  for j in range(mRow):
    mxjy = matrixLen * Xjy[j]
        #Select the Xjy from the list of
        computed dot products. multiplied by
        the size of matrix
    xjy = XSummation[j] * y
        #Select the x from the summations of
        rows of Matrix, multiplied by the
        summation of vectorY
    vectorV.append((mxjy - xjy) /
        (((matrixLen * xSquaredInside[j] -
        XSummation[j] ** 2) * (ySquaredInside
        - ySquared)) ** 0.5)) #Perform
        Pearson operation
  return vectorV

''' Main program '''
nInput = int(input("Input n: "))
matrixX = initializeNxN(nInput)
    #Initializes the nxn matrix
vectorY = initializeOneVector(nInput)
    #Initializes the n-sized vector
time_before = time.time()
returnVal = pearson_cor(matrixX, vectorY,
    nInput, nInput)
time_after = time.time()
print("Time elapsed: ", time_after -
    time_before)
```

Code of column-major with pure built-in functions:

```
import numpy

def columnWiseMatrix(matrixX):
  return [list(i) for i in zip(*matrixX)]

def summationRowGivenJTimesY(matrixCol,
    vectorY, lenMatrixCol):
```

```
  return sum(map(lambda x, y: x * y,
      matrixCol, vectorY)) * lenMatrixCol

#Returns a summation of modified element of
    a vector
def summationVectorSquared(vector):
  return sum(map(lambda x: x * x, vector))

#Returns a vector v
def pearson_cor(matrixX, vectorY, mRow,
    nColumn):
  vectorV = []
  matrixCol = columnWiseMatrix(matrixX)  #
      Transpose matrix rows to columns
  y = sum(vectorY)
      #Get sum of vectorY
  matrixLen = len(matrixX)
  ySquaredInside =
      summationVectorSquared(vectorY) *
      matrixLen #Performs squared summation
      of vectorY
  ySquared = y ** 2
      #Squares the summation of vectorY

  for j in range(mRow):
    mxjy =
        summationRowGivenJTimesY(matrixCol[j],
        vectorY, matrixLen) #Calculates the
        dot product per row against vectorY
    x = sum(matrixCol[j])

        #Calculates the sum per row of matrix
    xjy = x * y
    mXSquared =
        summationVectorSquared(matrixCol[j])
        * matrixLen #Performs square
        summations of row, multiplied by the
        matrix size
    xSquared = x ** 2

        #Squares the summation of row
    vectorV.append((mxjy - xjy) /
        (((mXSquared - xSquared) *
        (ySquaredInside - ySquared)) ** 0.5))
        #Perform Pearson operation
  return vectorV

''' Main program '''
nInput = int(input("Input n: "))
matrixX = initializeNxN(nInput)
vectorY = initializeOneVector(nInput)
time_before = time.time()
returnVal = pearson_cor(matrixX, vectorY,
    nInput, nInput)
time_after = time.time()
time_elapsed = time_after - time_before
print("Time elapsed: ", time_elapsed)
```