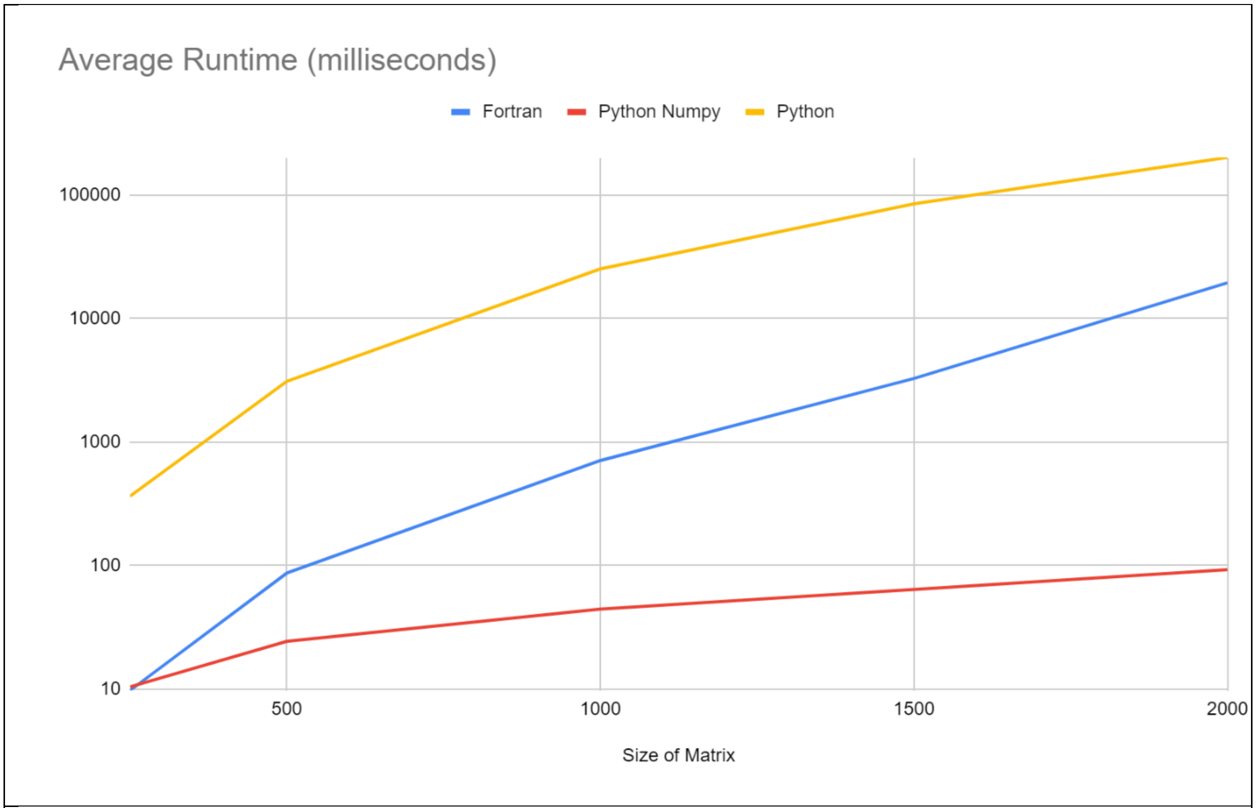
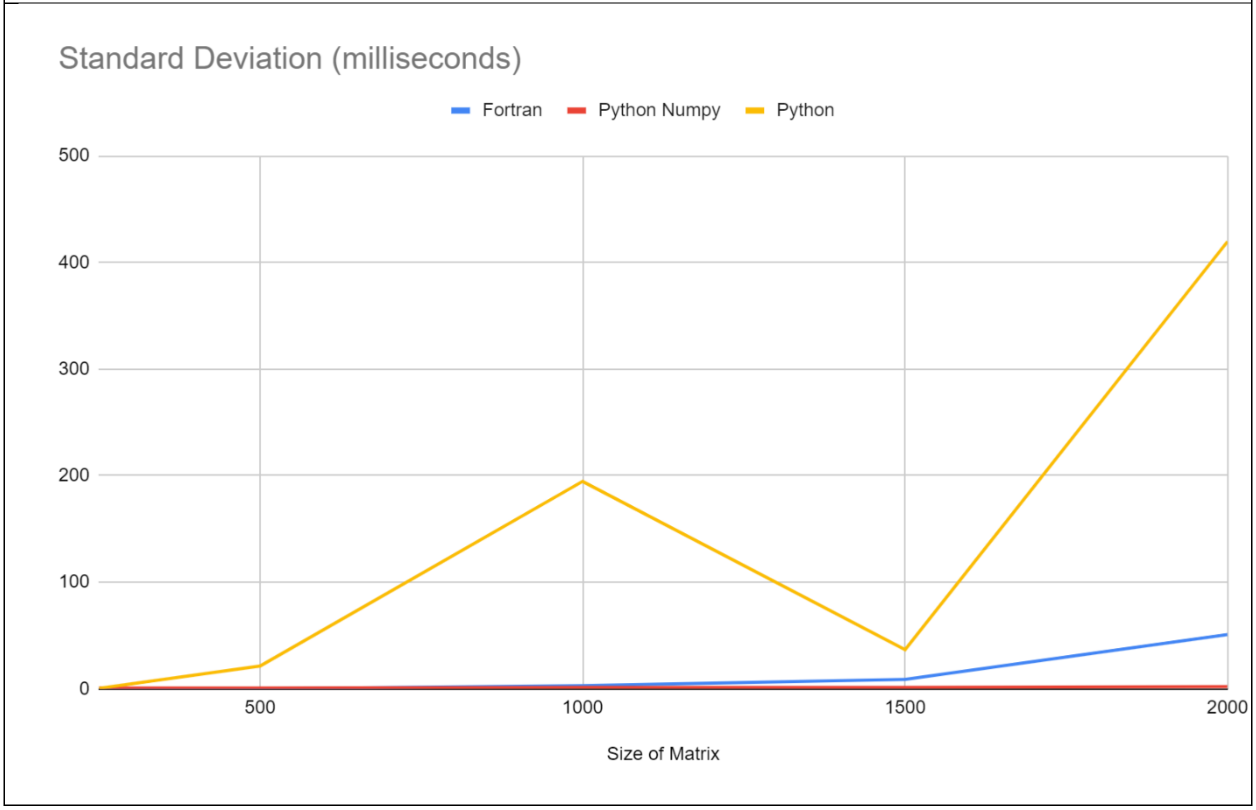
Problem: For this program we were to code gaussian elimination without partial pivoting in Fortran, Python, and Python with Numpy. We were to measure the execution time for 5 matrix sizes 5 times, and then compile all of the data to make a conclusion about compiled versus interpreted languages.

From the below graphs and data, we can see that I had to change the graph to a logarithmic scale just so that we could compare Fortran, Python, and Python with Numpy. What we can see is that Numpy comes in with the fastest processing time, followed by Fortran, and lastly by Python. Python is extremely slow just like we expected an interpreted language to be. Meanwhile Fortran is about 50 times faster in comparison. Numpy is around also around 50 times faster than Fortran, this may have to do that around 35% of Numpy is written in C or C++ with heavy optimizations. Altogether these findings support the claim that Compiled languages are faster than interpreted languages.


Average Runtime (milliseconds)
Fortran
Python Numpy
Python
100000
10000
1000
100
10
500
1000
1500
2000
Size of Matrix


Standard Deviation (milliseconds)
Fortran
Python Numpy
Python
500
400
300
200
100
0
500
1000
1500
2000
Size of Matrix

Fortran
(time in nanoseconds)

N=	250	500	1000	1500	2000
Run 1	9598600	87051500	712291300	3289435500	19403629000
Run 2	9770700	87552600	707690700	3279770800	19503962900
Run 3	9777900	86868200	707377200	3283228100	19533361800
Run 4	9908900	87352300	709229900	3267927400	19500700600
Run 5	9790500	87068000	714433300	3270141600	19518225400

N=	250	500	1000	1500	2000
Average	9769320	87178520	710204480	3278100680	19491975940
Standard Deviation	99148	242810	2738618	8054818	45665769

Python With Numpy
(time in nanoseconds)

N=	250	500	1000	1500	2000
Run 1	11635800	24514200	46968300	66147000	95005800
Run 2	10512900	23366300	43517500	64237100	89482400
Run 3	10926900	25411100	43216200	63458400	91804800
Run 4	8999400	25024900	44945100	62462500	94605000
Run 5	10174000	23785900	43682200	64626600	92982400

N=	250	500	1000	1500	2000
Average	10449800	24420480	44465860	64186320	92776080
Standard Deviation	873834	757552	1383260	1228179	2007530

Python Without Numpy
(time in nanoseconds)

N=	250	500	1000	1500	2000
Run 1	365671200	3120271100	25206693100	85043961600	202222826200
Run 2	364919000	3080262000	25146505700	84980426400	201425367800
Run 3	365471300	3095467700	25183657300	84969414000	202166646800

Run 4	364974500	3133637400	25606633100	85000886500	202353743800
Run 5	364327900	3095879600	25160973500	85051062200	202514992800

N=	250	500	1000	1500	2000
Average	365072780	3105103560	25260892540	85009150140	202136715480
Standard Deviation	470123	19179513	174073295	32987150	375372211

```
!Name: Marco Salazar, Date: 9/27/2020
!Fortran
!Purpose: Test how fast Gaussian Elimination in Fortran is compared to other languages.
!No inputs instead it calls all the gaussian eliminations in order.

!Main program that calls the function 5 times for each of the 5 values
!Prints out the time in nanoseconds for the operation to be completed in CSV format
PROGRAM gaussian_elimination
    IMPLICIT NONE
    call gauss(250)
    call gauss(250)
    call gauss(250)
    call gauss(250)
    call gauss(250)

    call gauss(500)
    call gauss(500)
    call gauss(500)
    call gauss(500)
    call gauss(500)

    call gauss(1000)
    call gauss(1000)
    call gauss(1000)
    call gauss(1000)
    call gauss(1000)

    call gauss(1500)
    call gauss(1500)
    call gauss(1500)
    call gauss(1500)
    call gauss(1500)
    call gauss(2000)
    call gauss(2000)
    call gauss(2000)
    call gauss(2000)
    call gauss(2000)

contains

!function that gives the time
!Sources: https://gcc.gnu.org/onlinedocs/gfortran/SYSTEM_005fCLOCK.html
integer(kind=8) function times()
    implicit none
    INTEGER(kind=8) :: count, count_rate, count_max
    CALL SYSTEM_CLOCK(count, count_rate, count_max)
    times = count
end function times

!function that performs gaussian elimination without partial pivoting
!Source: https://labmathdu.wordpress.com/gaussian-elimination-without-pivoting/
subroutine gauss(n)
    IMPLICIT NONE

    integer(kind=8) :: starttime,endtime
    INTEGER::n
    INTEGER::i,j,ii,jj

    REAL::s
    REAL,DIMENSION(n,n+1)::a
    REAL,DIMENSION(n)::x

    real::rand

    !Generate all of the random numbers in the array
    do ii=1,n
        do jj=1,n+1
            call random_number(rand)
            !Generate a random number [1, 1000] to avoid divide by 0 errors
            a(ii,jj) = floor(999*rand)+1
        end do
    end do
    starttime = times()

    !Do the Gaussian elimination
    DO j=1,n
        DO i=j+1,n
```

```
            a(i,:)=a(i,:)-a(j,:)*a(i,j)/a(j,j)
        END DO
    END DO

    DO i=n,1,-1
        s=a(i,n+1)
        DO j=i+1,n
            s=s-a(i,j)*x(j)
        END DO
        x(i)=s/a(i,i)
    END DO

    endtime = times()

    !print the execution time in csv format
    Print *, endtime-starttime, ","
end subroutine gauss

END PROGRAM
```

```
#Name:Marco Salazar, Date: 9/27/2020
#Python with Numpy
#Purpose: to compare Python with numpy in gaussian elimination to other programming languages running times.
#No Inputs, instead it calls all the gaussian elimination in order.
#Outputs in CSV format

import numpy as np
import math
import time
import timeit

#https://numpy.org/doc/stable/reference/arrays.nditer.html
#https://stackoverflow.com/questions/52864988/compare-the-result-of-gaussian-elimination-with-the-output-of-numpy-linalg-solve
# creates random array and computes the gaussian elimination of it.
def gaus(length):
    array = np.random.rand(length,length)
    with np.nditer(array, op_flags=['readwrite']) as it:
        for x in it:
            x[...] = math.floor(x*999)+1

    brray = np.random.rand(length,1)
    with np.nditer(array, op_flags=['readwrite']) as it:
        for x in it:
            x[...] = math.floor(x*999)+1
    start_time = timeit.default_timer()*1000000000
    x = np.linalg.solve(array, brray)
    print("%s," % (timeit.default_timer()*1000000000- start_time))

# Do all of the 5 tries for each of the 5 sizes.
gaus(250)
gaus(250)
gaus(250)
gaus(250)
gaus(250)

gaus(500)
gaus(500)
gaus(500)
gaus(500)
gaus(500)

gaus(1000)
gaus(1000)
gaus(1000)
gaus(1000)
gaus(1000)

gaus(1500)
gaus(1500)
gaus(1500)
gaus(1500)
gaus(1500)

gaus(2000)
gaus(2000)
gaus(2000)
gaus(2000)
gaus(2000)
```

```
#Name:Marco Salazar, Date: 9/27/2020
#Python without Numpy
#Purpose: to compare Python without numpy in gaussian elimination to other programming languages running times.
#No Inputs, instead it calls all the gaussian elimination in order.
#Outputs in CSV format

import math
import time
import timeit
import random
```

```
#https://learnche.org/3E4/Assignment_2_-_2010_-_Solution/Bonus_question with edits
def forward_elimination(A, b, n):
    """
    Calculates the forward part of Gaussian elimination.
    """
    for row in range(0, n-1):
        for i in range(row+1, n):
            factor = A[i][row] / A[row][row]
            for j in range(row, n):
                A[i][j] = A[i][j] - factor * A[row][j]

            b[i] = b[i] - factor * b[row]
    return A, b

def back_substitution(a, b, n):
    """"
    Does back substitution, returns the Gauss result.
    """
    x = [0 for j in range(n)]
    x[n-1] = b[n-1] / a[n-1][ n-1]
    for row in range(n-2, -1, -1):
        sums = b[row]
        for j in range(row+1, n):
            sums = sums - a[row][j] * x[j]
        x[row] = sums / a[row][row]
    return x

def gauss(A, b):
    """
    This function performs Gauss elimination without pivoting.
    """
    n = len(A[0])

    A, b = forward_elimination(A, b, n)
    return back_substitution(A, b, n)



def gaus(length):
    array = [[math.floor(random.random()*999)+1 for i in range(length)] for j in range(length)]
    brray = [math.floor(random.random()*999)+1 for j in range(length)]

    start_time = timeit.default_timer()*1000000000
    gauss(array, brray)
    print("%s," % (timeit.default_timer()*1000000000- start_time))

gaus(250)
gaus(250)
gaus(250)
gaus(250)
gaus(250)

gaus(500)
gaus(500)
gaus(500)
gaus(500)
gaus(500)

gaus(1000)
gaus(1000)
gaus(1000)
gaus(1000)
gaus(1000)

gaus(1500)
gaus(1500)
gaus(1500)
gaus(1500)
gaus(1500)

gaus(2000)
gaus(2000)
gaus(2000)
gaus(2000)
gaus(2000)
```