Jeffrey Lansford

9/28/20

Program 4

We are comparing the runtime of interpreted and compiled languages. We created three programs on Gaussian Elimination in Python with Numpy, Python without Numpy, and Fortran and compared their runtimes. There are two implenataitons of Python with Numpy, one using the matrix of numpy to do the elimination and version 2 where it uses the numpy function linalg.solve() to solve the equation. We can see the difference in using the module.

```
"""
Python With Numpy
Jeffrey Lansford
Program 4
9/28/2020
Python Implmentation of Gauss Elimination with back substition using Numpy
"""
import json
import random
import sys
import time

import numpy as np


def createArray(size):
    """
    to create an matrix of random numbers between 1 and 20 with a given size

    size    --size of matrix, N x N+1
    """
    n = np.zeros((size, size + 1))
    for x in range(size):
        for y in range(size + 1):
            n[x, y] = random.randint(1, 20)
    return n


def createArray_2(size):
    """
    to create an matrix of random numbers between 1 and 20 with a given size

    size    --size of matrix, N x N+1
```

```
    """
    a = np.zeros((size, size))
    b = np.zeros((size, 1))
    for x in range(size):
        for y in range(size):
            a[x, y] = random.randint(1, 20)
    for y in range(size):
        b[y, 0] = random.randint(1, 20)
    return (a, b)


def Gauss(n):
    """
    does Gauss Elimination with back substitution on a given matrix
    based on https://www.codesansar.com/numerical-methods/gauss-elimination-
method-python-program.htm

    n   --matrix to do Gauss Elimination
    """
    size = len(n)

    # creates lower triangular matrix
    for i in range(size):
        for j in range(i + 1, size):
            ratio = n[j, i] / n[i, i]
            n[j] = n[j] - ratio * n[i]

    # Back substitution
    for i in range(size - 1, -1, -1):
        for j in range(i - 1, -1, -1):
            ratio = n[j, i] / n[i, i]
            n[j] = n[j] - ratio * n[i]
        n[i] = n[i] / n[i, i]


def Gauss_2(a, b):
    """
    does Gauss Elimination with back substion on a given matrix using the numpy s
olve linear equations function

    a   -- Coefficient Matrix
    b   -- Ordinate Matrix
    """
    np.linalg.solve(a, b)
```

```
def test_case(size):
    """
    runs test cases on a given size of the matrix. records length of runtime Gaus
s Elimination and returns in milliseconds

    size    --size of matrix
    """
    n = createArray(size)
    start = time.perf_counter()
    Gauss(n)
    end = time.perf_counter()
    return (end - start) * 1000


def test_case_2(size):
    """
    runs test cases on a given size of the matrix. records length of runtime Gaus
s Elimination and returns in milliseconds

    uses Version 2 of Gauss Elimination

    size    --size of matrix
    """
    a, b = createArray_2(size)
    start = time.perf_counter()
    Gauss_2(a, b)
    end = time.perf_counter()
    return (end - start) * 1000


# Sample sizes
sizes = [250, 500, 1000, 1500, 2000]

# Store results into json file for eazy copying of data into excel
results = {"Version 1": {}, "Version 2": {}}

# Test Verison 1
# Run 5 test runs on the different sizes
print("Python with Numpy Version 1")
for i in range(5):
    print(f"test run {i+1}")
    results["Version 1"].update({f"Test Run {i+1}": {}})
    for size in sizes:
        a = test_case(size)
        results["Version 1"][f"Test Run {i+1}"].update({size: a})
```

```python
        print("\t{:4}: {:} milliseconds".format(size, a))

# Test Version 2
print("Python with Numpy Version 2")
for i in range(5):
    print(f"test run {i+1}")
    results["Version 2"].update({f"Test Run {i+1}": {}})
    for size in sizes:
        a = test_case_2(size)
        results["Version 2"][f"Test Run {i+1}"].update({size: a})
        print("\t{:4}: {:} milliseconds".format(size, a))
y = json.dumps(results, indent=4)
f = open("resultsNumpy.json", "w")
f.write(y)
f.close()
```

```python
"""
Python Without Numpy
Jeffrey Lansford
Program 4
9/28/2020
Python Implmentation of Gauss Elimination with back substition using without Nump
y
"""
import json
import random
import sys
import time


def createArray(size):
    """
    to create an matrix of random numbers between 1 and 20 with a given size

    size    --size of matrix, N x N+1
    """
    n = []
    for x in range(size):
        n.append([])
        for _ in range(size + 1):
            n[x].append(float(random.randint(1, 20)))
    return n
```

```
def Gauss(n):
    """
    does Gauss Elimination with back substion on a given matrix
    based on https://www.codesansar.com/numerical-methods/gauss-elimination-
method-python-program.htm

    n   --matrix to do Gauss Elimination
    """
    size = len(n)

    # creates lower triangular matrix
    for i in range(size):
        for j in range(i + 1, size):
            ratio = n[j][i] / n[i][i]
            for k in range(size + 1):
                n[j][k] = n[j][k] - ratio * n[i][k]

    # Back substitution
    for i in range(size - 1, -1, -1):
        for j in range(i - 1, -1, -1):
            ratio = n[j][i] / n[i][i]
            for k in range(size + 1):
                n[j][k] = n[j][k] - ratio * n[i][k]
        temp = n[i][i]
        for k in range(size + 1):
            n[i][k] = n[i][k] * (1.0 / temp)


def test_case(size):
    """
    runs test cases on a given size of the matrix. records length of runtime Gaus
s Elimination and returns in milliseconds

    size    --size of matrix
    """
    n = createArray(size)
    start = time.perf_counter()
    Gauss(n)
    end = time.perf_counter()
    return (end - start) * 1000


# Sample sizes
```

```python
sizes = [250, 500, 1000, 1500, 2000]

# Store results into json file for eazy copying of data into excel
results = {}

# Run 5 test runs on the different sizes
for i in range(5):
    print(f"test run {i+1}")
    results.update({f"Test Run {i+1}": {}})
    for size in sizes:
        a = test_case(size)
        results[f"Test Run {i+1}"].update({size: a})
        print("\t{:4}: {:} milliseconds".format(size, a))
y = json.dumps(results, indent=4)
f = open("resultsNotNumpy.json", "w")
f.write(y)
f.close()
```

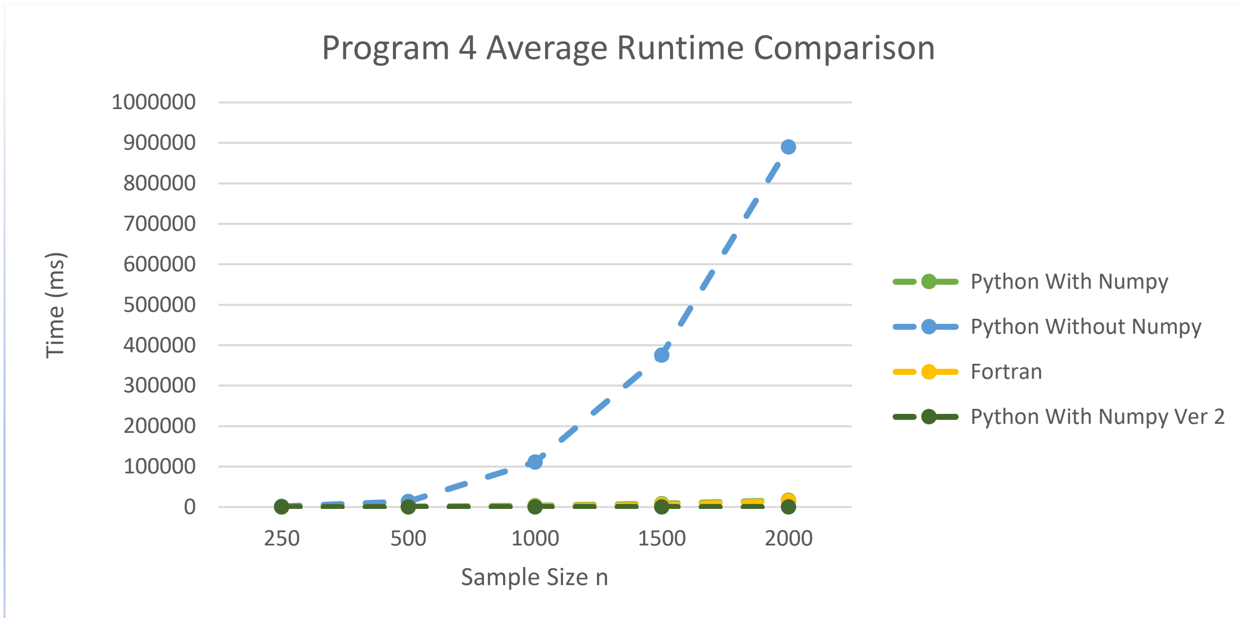
```fortran
    !    Fortran
    !    Jeffrey Lansford
    !    9/28/20
    !    Program 4
    !    Fortran program to test the time it takes to do gaussian elimination on
different samples

       ! creates a NxN+1 matrix with random numbers
       function create_matrix ( N ) result(A)
           implicit NONE
           integer, intent(in) :: N
           integer i,j
           real, dimension(:,:), allocatable :: A

           ALLOCATE(A(N,N+1))
           do i = 1,N
               do j=1,N+1
                   A(i,j) = INT((rand() * (20 - 1 + 1)) + 1 )
               end do
           end do
       end function create_matrix

       ! does Gauss Elimination with back substitution on a given matrix
       ! source: https://labmathdu.wordpress.com/gaussian-elimination-without-
pivoting/
```

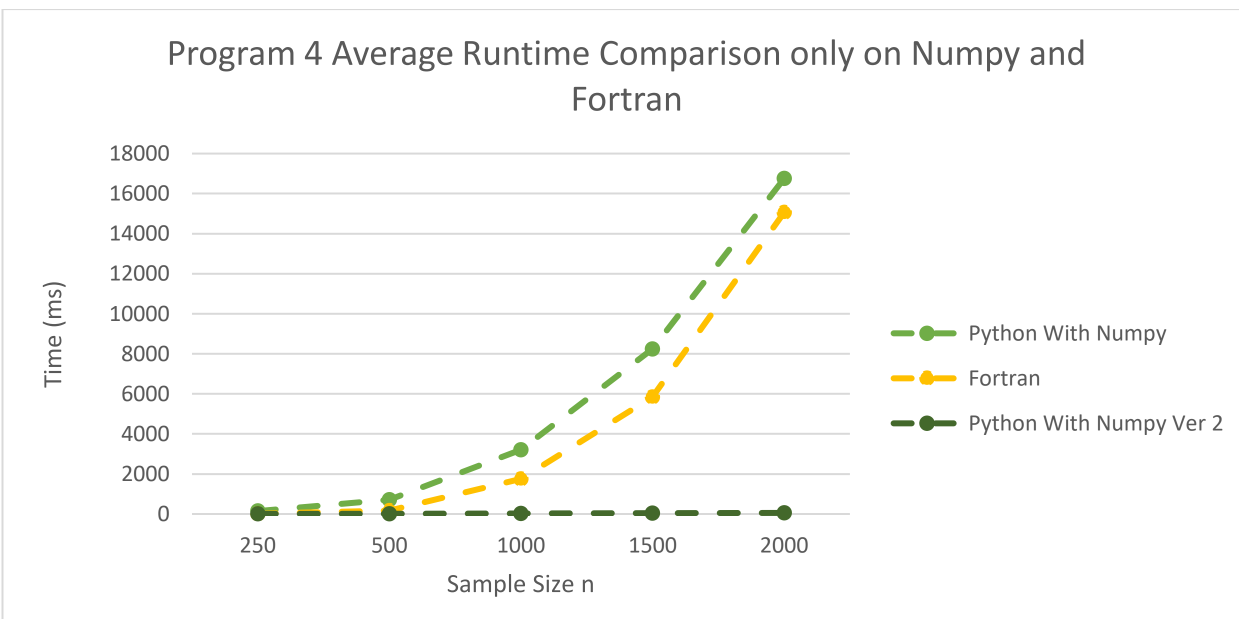
```fortran
       subroutine gaussian_elimination ( a,n )
           implicit none
           real, dimension(:,:), intent(inout) ::a
           INTEGER,intent(in)::n
           INTEGER::i,j
           REAL::s

           !    Creates lower triangulr matrix
           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

           !    Back substitution
           DO i=n,1,-1
               DO j=i-1,1,-1
                     s= a(j,i) / a(i,i)
                     a(j,:) = a(j,:) - (s*a(i,:))
               END DO
               a(i,:) = a(i,:) / a(i,i)
           END DO

       end subroutine gaussian_elimination

       ! runs test cases on a given size and records time of Gauss Elimination an
d retunrs in milliseconds
       real function test_case ( N ) result(T)
       implicit NONE
           interface
               function create_matrix(N) result (A)
                   integer, intent(in) :: N
                   real, dimension(:,:), allocatable :: A
               end function
               subroutine gaussian_elimination(A,N)
                   real, dimension(:,:), intent(inout) ::A
                   integer,intent(in)::N
               end subroutine
          end interface
           integer, intent(in) :: N
           real, dimension(:,:),allocatable :: A

           REAL :: time_begin, time_end

           A=create_matrix(N)
```

```fortran
    CALL CPU_TIME ( time_begin )

    call gaussian_elimination(A,N)

    CALL CPU_TIME ( time_end )

    T= (time_end - time_begin ) *1000


end function test_case

! Main Program
program p4
    implicit NONE
    interface
         real function test_case ( N ) result(T)
             integer, intent(in) :: N
         end function
   end interface

    integer, dimension (5) :: Sizes

    integer :: n
    integer :: j
    integer :: i
    real :: time
    ! Sample sizes
    Sizes(1) = 250
    Sizes(2) = 500
    Sizes(3) = 1000
    Sizes(4) = 1500
    Sizes(5) = 2000

    do j=1,5
        print *,"Test Case ",j
        do n=1,5
            i = Sizes(n)
            time = test_case(i)

            print *, i, " ", time,"milliseconds"
        end do
    end do
```

```
    end program p4
```

		Python With Numpy Runtime (milliseconds)	Python With Numpy Ver 2 (milliseconds)	Python Without Numpy Runtime (milliseconds)	Fortran Runtime (milliseconds)
Test Run 1	250	150.340615	7.580614001	1673.531147	18.2860012
	500	722.451615	15.230606	13955.92429	169.589996
	1000	3248.193132	25.390997	112638.4737	1724.06006
	1500	8267.263848	35.511681	376714.8769	5548.72705
	2000	16802.76659	53.77538	890237.1387	15226.3389
Test Run 2	250	148.020904	6.422822	1655.63291	17.7345276
	500	703.492311	14.908305	13640.60448	172.340393
	1000	3199.850065	25.308434	110412.627	1826.24438
	1500	8247.976325	35.186319	374970.0842	5569.72314
	2000	16676.37818	53.391513	890363.4964	15145.4521
Test Run 3	250	146.565954	6.447104002	1655.068345	17.326355
	500	707.830145	15.514842	13631.13695	166.664124
	1000	3194.521737	25.329335	110439.3045	1739.32263
	1500	8230.046723	36.029055	375032.4	5933.81104
	2000	16798.94426	54.493686	889779.3242	15062.2363
Test Run 4	250	143.991378	6.620549	1654.020789	17.1813965
	500	710.513833	15.131258	13620.85476	164.558411
	1000	3201.064735	25.658236	110520.3389	1755.37866
	1500	8201.293432	45.031931	374921.1646	6179.86279
	2000	16805.92837	54.318574	890325.1904	14898.8262
Test Run 5	250	147.493125	6.514468005	1656.353108	17.8604126
	500	717.631964	15.36279701	13622.71053	167.388916
	1000	3227.470055	25.602421	110415.7876	1721.90088
	1500	8221.453807	35.44697	374930.7945	5990.08203
	2000	16669.36816	54.872416	890048.4804	14962.9824
Average	250	147.2823952	6.717111402	1658.92126	17.67773858
	500	712.3839736	15.2295616	13694.2462	168.108368
	1000	3214.219945	25.4578846	110885.3064	1753.381322
	1500	8233.606827	37.4411912	375313.864	5844.44121
	2000	16750.67711	54.1703138	890150.726	15059.16718
Standard Deviation	250	2.306869784	0.488757514	8.211531807	0.440568975
	500	7.61897715	0.23022113	146.4917167	2.971148925
	1000	22.89822728	0.161528547	981.0183965	42.89934742
	1500	25.21806529	4.254345888	784.4104199	275.9631922
	2000	71.11120187	0.587792813	240.6084415	132.6288361





Looking at the graphs, we can see that Python without Numpy takes a very long time compared to the others. This gives us a clue on how fast purely interpreted languages take compared to compiled languages. The second graph goes into finer details between Python with the module Numpy and Fortran. Here we can see that Python with Numpy can be faster and slower than Fortran based on how it is used. Python With Numpy solution uses Numpy arrays to represent its matrix and probably uses the module when running the arithmetic on the matrix elements. We can see that we are still have a performance hit on execution speed from Python. When using the Python With Numpy Ver 2, we see that we have a dramatic difference in speed. Ver 2 uses the numpy function linalg.solve() to do the Gaussian Elimination within the numpy module. This solution is tons faster than even the Fortran solution, which we would expect to be faster than all three solutions as Fortran is a compiled language. The argument that interpreted languages are slower than compiled languages may not hold up in all cases. Here, Python takes advantage of using modules to help speed it up to make it equivalent to a compiled language, or even faster than it. Though the data is very noisy, especially when we reach higher runtimes when more than likely the OS would start to impede on runtime.