AMOD-5420H-A-HPC

Final Project

Suraj Suresh Sajjan

Introduction:

In this project, we write code for a couple of problems using two of the programming languages namely, C++ and Julia. We shall discuss the complexity of writing code in each language and also compare the performance of each of the language to the other.

We are using atom IDE on CentOS for writing C++ codes. We are using Jupyter notebook with Julia on Windows 10 for writing Julia code.

Before we start with the analysis, let us look at some of the Pros and Cons of C++ and Julia.

C++:

Pros: It is one of the foundation languages which has a huge community and following. There are libraries present for almost all kinds of tasks. C++ allows us to out the large arrays on “heap” which helps in avoiding “stack overflow”. C++ was also one of the earliest programming languages to leverage the concept of Object-Oriented Programming (OOP) and includes inheritance, polymorphism, encapsulation, class, and abstraction. Most of the code written in C work on C++. It is one of the best ways to understand algorithms. It is one of the highly portable languages and is the language most developers choose for multi-device, multi-platform app development. It also allows exception handling and function overloading. Due to its power and efficiency, it has a range of applications from GUI applications to 3D graphics for games to real-time scientific simulations.

Cons: The standard used allows many things that can result in unexpected behavior. There is no boundary check on arrays. It also allows improper type conversion. This could result in an inexperienced programmer to corrupt his memory. It does not do a lot of memory management, forcing the developers to do the majority of it themselves. The C++ object-oriented programming is unnecessarily basic compared to other languages like Java which have better and more flexible systems. C++ programming can be quite complicated as the syntax is complex and the standard library is small. This makes it challenging and hard to learn to code in it. It is also syntactically strict. The syntax has very less flexibility making it difficult to write readable user-friendly code. The higher-level features such as GUIs, threading, and networking depend on the operating system making it unstandardized. This forces the programmer to either include different versions of code or include outside libraries which have already done so.

Julia:

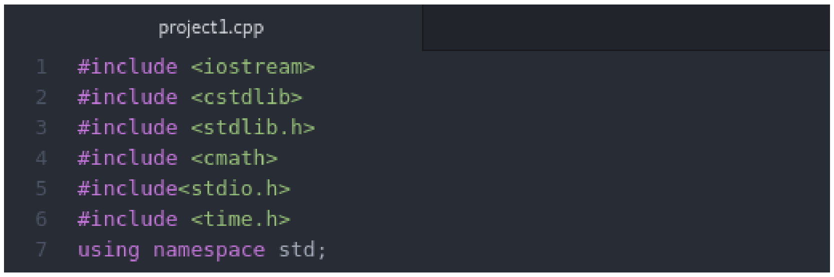
Pros: Julia reaches the speed of compiled languages such as C++ and Fortran. It provides dynamism of high-level languages such as Python as well as mathematical notations found in MATLAB. Julia also provides the statistical ease of languages such as R as well as the ease of use of Python. The combination of these elements makes Julia a very powerful language. Julia a multiple dispatch language. Any function call made may be dispatched to different functions. Another feature of Julia is its speed. Julia is capable of compiling code on the fly. It combines the method of interpreting languages such as Python which transform the code into bytecode with compiled languages such as C which compiles its code into machine code and then running the generated executable. This enables Julia to run at similar speeds of C. Packages are an import part of Julia like R. Julia has a built-in package manager containing over 1900 registered packages. Julia's amazing feature is its parallelism. The programmer can parallelize directly from the command line by calling the desired script with the desired number of cores. It is also capable of sending different threads directly from code.

Cons: Julia came into the picture in the year 2012. Due to its short existence, the support to the language is also relatively limited. As of yet, few libraries exist wit the community being quite small. The design of Julia probably comes from MATLAB, making it unnatural to interface C ++ or Python. Julia arrays are 1-indexed which can be can catch users off guard, the ones who are used to coding in Python or C++.
Julia list comprehensions lack the ability to use conditionals, unlike Python. It can be done with for loops and if/else as normally done. The matrices in Julia are accessed column-major compared to Python's NumPy where the matrices are accessed row-order. This can affect the design decisions on how to iterate over matrices effectively in memory. The dictionaries in Julia are hashed differently than Python dictionaries making them slower in many cases.

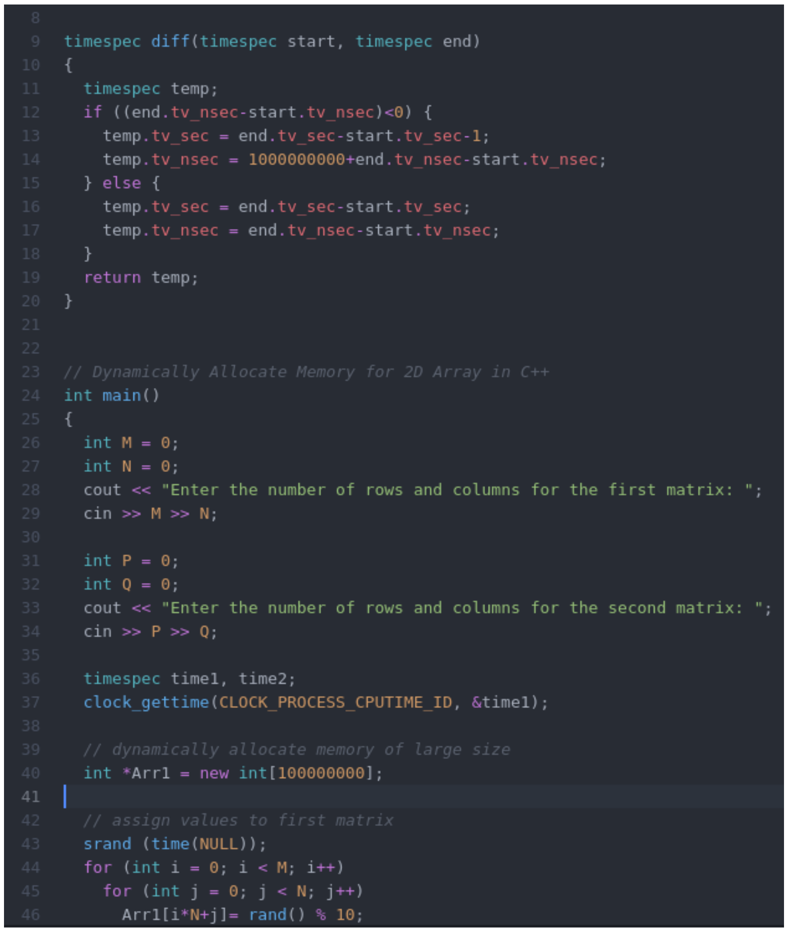
Let us start writing code for some of the simpler tasks such as Matrix elements addition, the addition of diagonal elements and the multiplication of two matrices in both Julia and C++ and see how well they fare against each other.

1. Addition of all the element of the matrix and then the addition of only the diagonal elements followed by multiplication of the first matrix with another of the same dimensions.

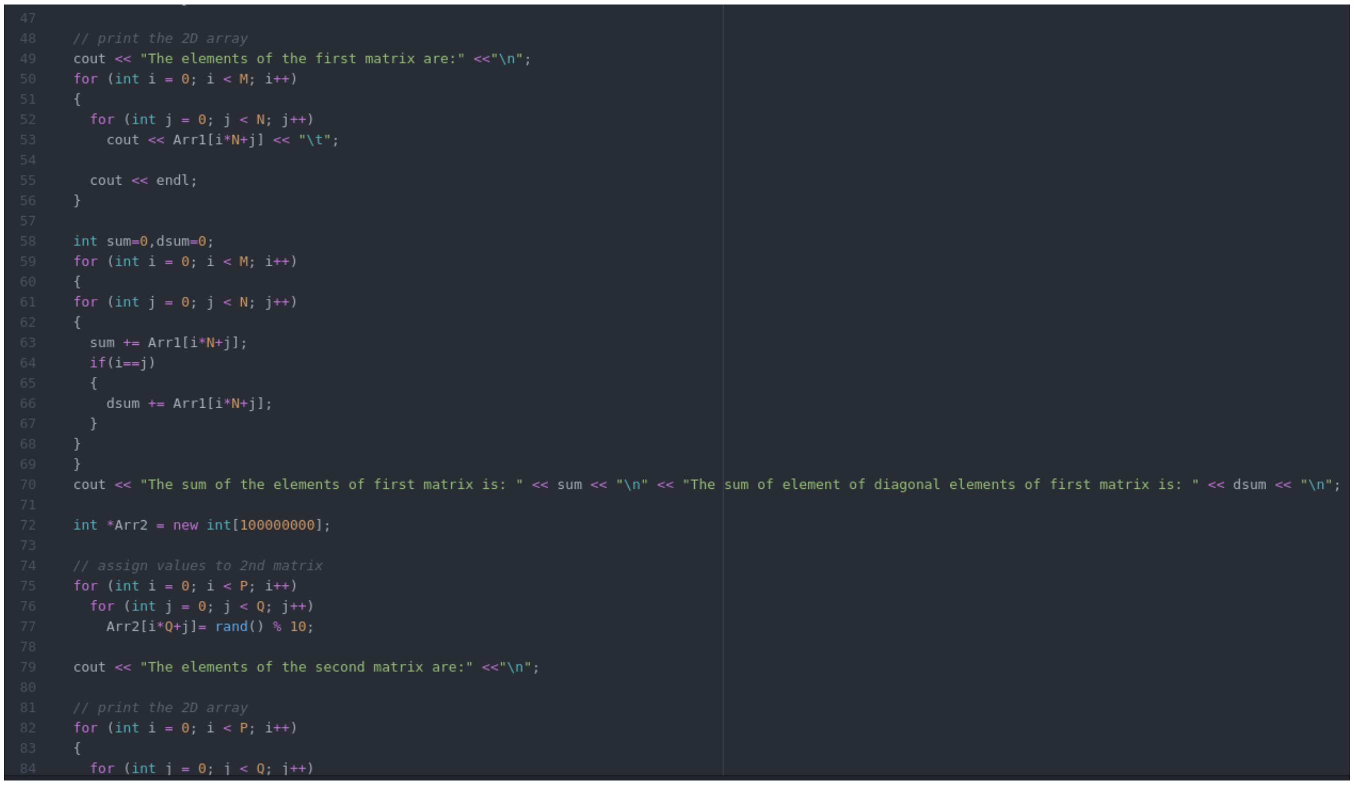
C++ code:



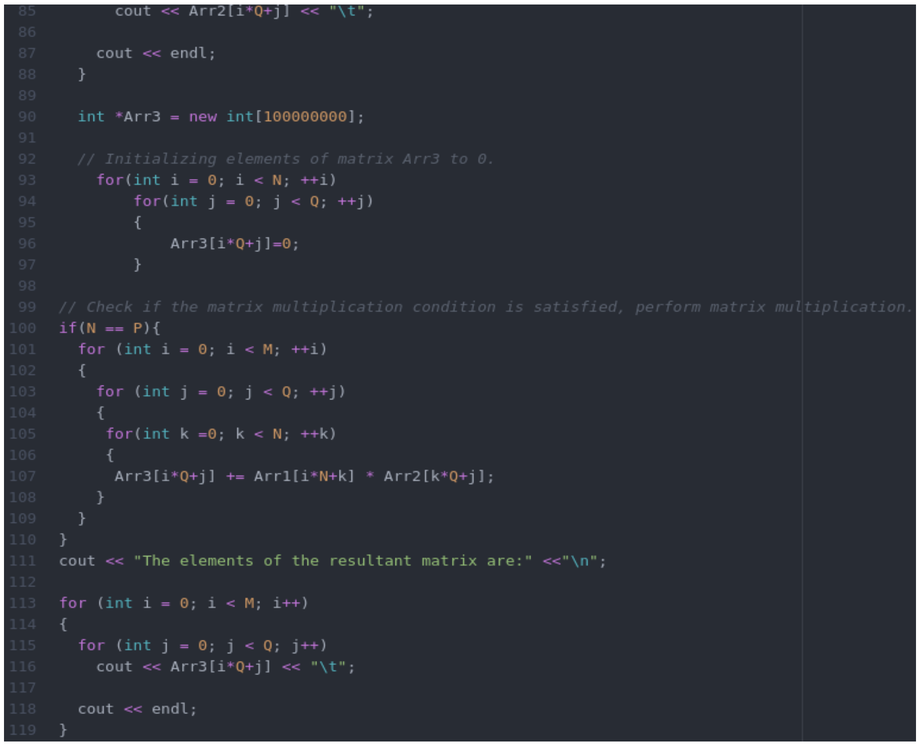
```cpp
#include <iostream>
#include <cstdlib>
#include <stdlib.h>
#include <cmath>
#include<stdio.h>
#include <time.h>
using namespace std;
```



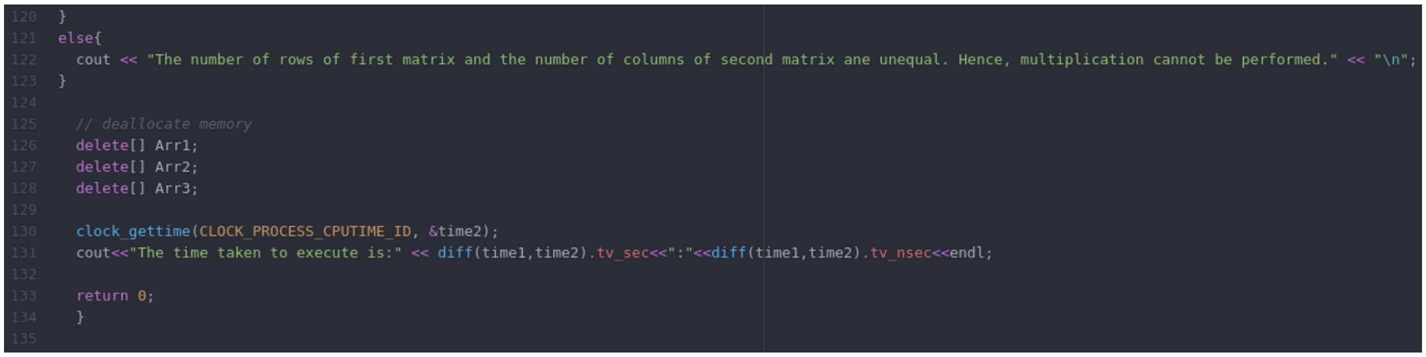
```cpp

timespec diff(timespec start, timespec end)
{
  timespec temp;
  if ((end.tv_nsec-start.tv_nsec)<0) {
    temp.tv_sec = end.tv_sec-start.tv_sec-1;
    temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
  } else {
    temp.tv_sec = end.tv_sec-start.tv_sec;
    temp.tv_nsec = end.tv_nsec-start.tv_nsec;
  }
  return temp;
}


// Dynamically Allocate Memory for 2D Array in C++
int main()
{
  int M = 0;
  int N = 0;
  cout << "Enter the number of rows and columns for the first matrix: ";
  cin >> M >> N;

  int P = 0;
  int Q = 0;
  cout << "Enter the number of rows and columns for the second matrix: ";
  cin >> P >> Q;

  timespec time1, time2;
  clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time1);

  // dynamically allocate memory of large size
  int *Arr1 = new int[100000000];

  // assign values to first matrix
  srand (time(NULL));
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      Arr1[i*N+j]= rand() % 10;
```



```cpp

  // print the 2D array
  cout << "The elements of the first matrix are:" <<"\n";
  for (int i = 0; i < M; i++)
  {
    for (int j = 0; j < N; j++)
      cout << Arr1[i*N+j] << "\t";

    cout << endl;
  }

  int sum=0,dsum=0;
  for (int i = 0; i < M; i++)
  {
  for (int j = 0; j < N; j++)
  {
    sum += Arr1[i*N+j];
    if(i==j)
    {
      dsum += Arr1[i*N+j];
    }
  }
  }
  cout << "The sum of the elements of first matrix is: " << sum << "\n" << "The sum of element of diagonal elements of first matrix is: " << dsum << "\n";

  int *Arr2 = new int[100000000];

  // assign values to 2nd matrix
  for (int i = 0; i < P; i++)
    for (int j = 0; j < Q; j++)
      Arr2[i*Q+j]= rand() % 10;

  cout << "The elements of the second matrix are:" <<"\n";

  // print the 2D array
  for (int i = 0; i < P; i++)
  {
    for (int j = 0; j < Q; j++)
```



```
      cout << Arr2[i*Q+j] << "\t";

    cout << endl;
  }

  int *Arr3 = new int[100000000];

  // Initializing elements of matrix Arr3 to 0.
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < Q; ++j)
        {
            Arr3[i*Q+j]=0;
        }

// Check if the matrix multiplication condition is satisfied, perform matrix multiplication.
if(N == P){
  for (int i = 0; i < M; ++i)
  {
    for (int j = 0; j < Q; ++j)
    {
     for(int k =0; k < N; ++k)
     {
      Arr3[i*Q+j] += Arr1[i*N+k] * Arr2[k*Q+j];
    }
  }
}
cout << "The elements of the resultant matrix are:" <<"\n";

for (int i = 0; i < M; i++)
{
  for (int j = 0; j < Q; j++)
    cout << Arr3[i*Q+j] << "\t";

  cout << endl;
}
```



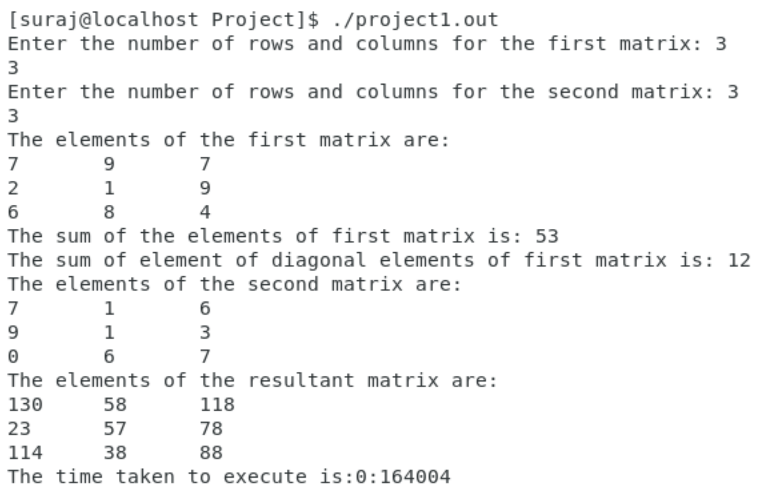
```
}
else{
  cout << "The number of rows of first matrix and the number of columns of second matrix ane unequal. Hence, multiplication cannot be performed." << "\n";
}

  // deallocate memory
  delete[] Arr1;
  delete[] Arr2;
  delete[] Arr3;

  clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time2);
  cout<<"The time taken to execute is:" << diff(time1,time2).tv_sec<<":"<<diff(time1,time2).tv_nsec<<endl;

  return 0;
  }
```

In the above code, we declare Arr1 and Arr2 on heap memory using the new* function. The matrix elements are populated with random elements using the rand function. Then we use for loops to iterate over the elements of the array Arr1. The variable “sum” is used to store the value of the addition of all the matrix elements. We use the condition in if loop where we instruct the compiler to add only the elements whose x, y positions are same, which basically sums up the diagonal elements and the value is stored in “dsum”. Posix high resolution clock for C++ is used to capture the time taken by the code to execute.

Next, we initialize Arr3 in heap memory in a similar way and initialize the array elements to 0. Then, we use for loops to iterate over the element of Arr1 and Arr2 and store the results generated upon matrix multiplication in Arr3. After the calculations are done, it is important to delete the Arrays declared on heap memory.

Output:

First, let us check the output for a small 3X3 matrix to verify if the results we are getting are correct:

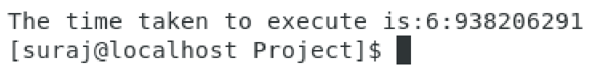


```
[suraj@localhost Project]$ ./project1.out
Enter the number of rows and columns for the first matrix: 3
3
Enter the number of rows and columns for the second matrix: 3
3
The elements of the first matrix are:
7       9       7
2       1       9
6       8       4
The sum of the elements of first matrix is: 53
The sum of element of diagonal elements of first matrix is: 12
The elements of the second matrix are:
7       1       6
9       1       3
0       6       7
The elements of the resultant matrix are:
130     58      118
23      57      78
114     38      88
The time taken to execute is:0:164004
```

We observe that the output is correct, the sum of elements of Matrix 1 is 53, the sum of its diagonals is 12, the resultant matrix upon the multiplication of the two matrices is correct.

We see that it takes 0.16s to execute the code.

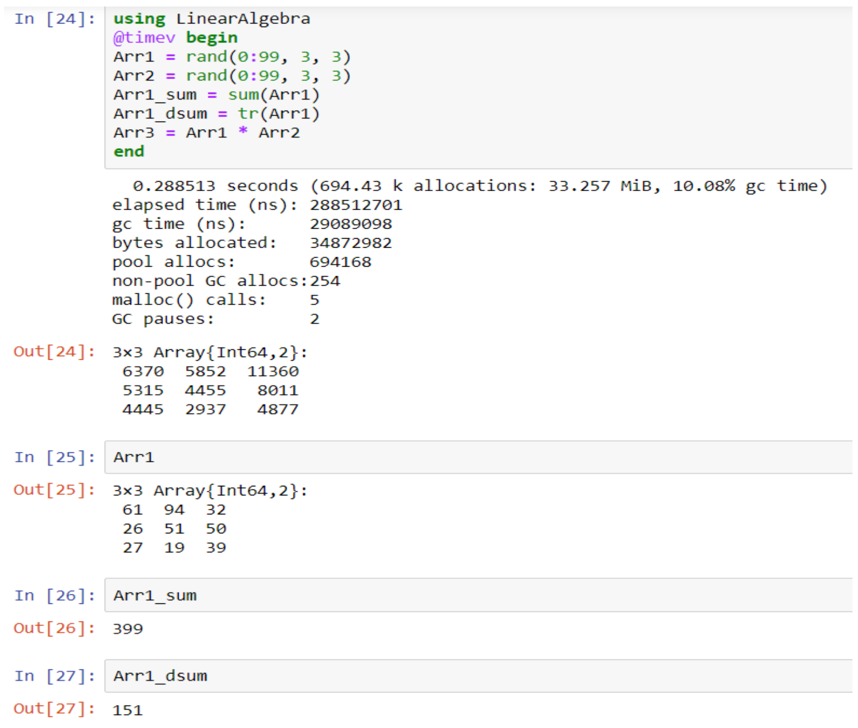
Now let us check the time it takes to compute on matrices of higher dimension, say 1000X1000.



```
The time taken to execute is:6:938206291
[suraj@localhost Project]$
```

We see that it takes 6.94s to do the computations.

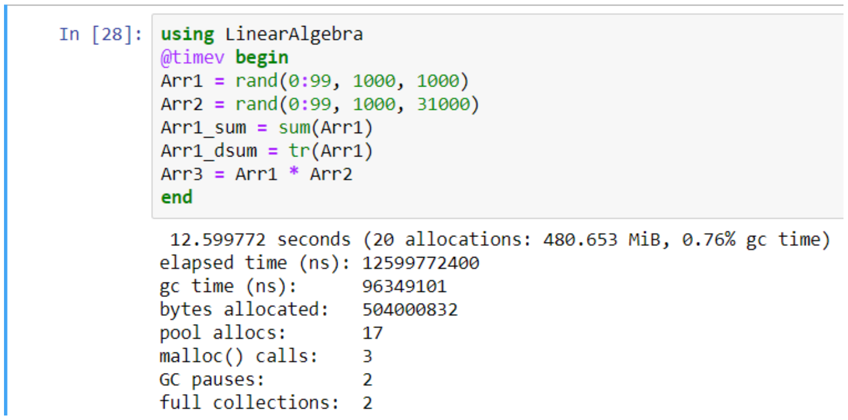
Now, let us perform the above calculations in Julia:



```
In [24]: using LinearAlgebra
         @timev begin
         Arr1 = rand(0:99, 3, 3)
         Arr2 = rand(0:99, 3, 3)
         Arr1_sum = sum(Arr1)
         Arr1_dsum = tr(Arr1)
         Arr3 = Arr1 * Arr2
         end

           0.288513 seconds (694.43 k allocations: 33.257 MiB, 10.08% gc time)
         elapsed time (ns): 288512701
         gc time (ns):      29089098
         bytes allocated:   34872982
         pool allocs:       694168
         non-pool GC allocs:254
         malloc() calls:    5
         GC pauses:         2

Out[24]: 3×3 Array{Int64,2}:
          6370  5852  11360
          5315  4455   8011
          4445  2937   4877

In [25]: Arr1

Out[25]: 3×3 Array{Int64,2}:
          61  94  32
          26  51  50
          27  19  39

In [26]: Arr1_sum

Out[26]: 399

In [27]: Arr1_dsum

Out[27]: 151
```

From the above output, we found the value of the sum of all the elements of the array to be 399, the sum of the diagonal element to be 151 and the resultant matrix after the multiplication of the two matrices is stored in Arr3.

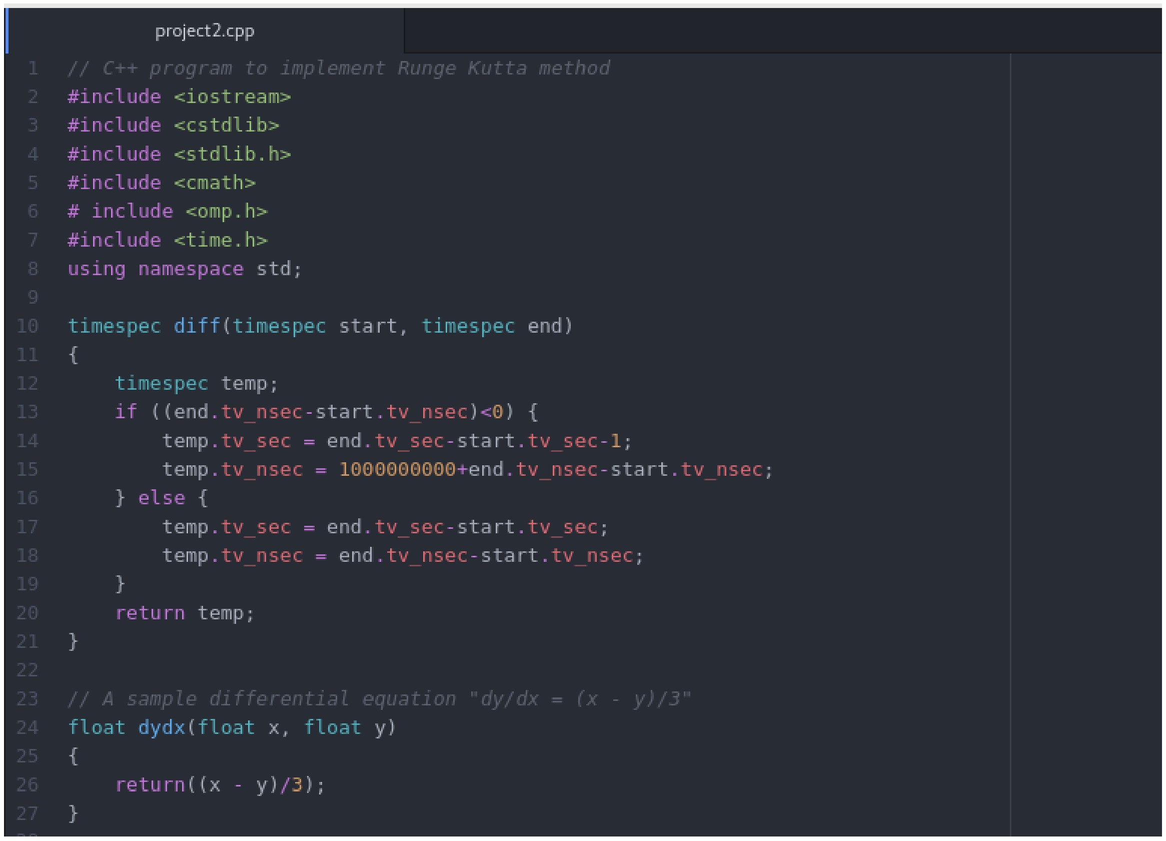
We observe that Julia takes 0.29s to do the above calculations on 3X3 matrices compared to 0.16s in C++. Now, let us check the results on a 1000X1000 matrix.



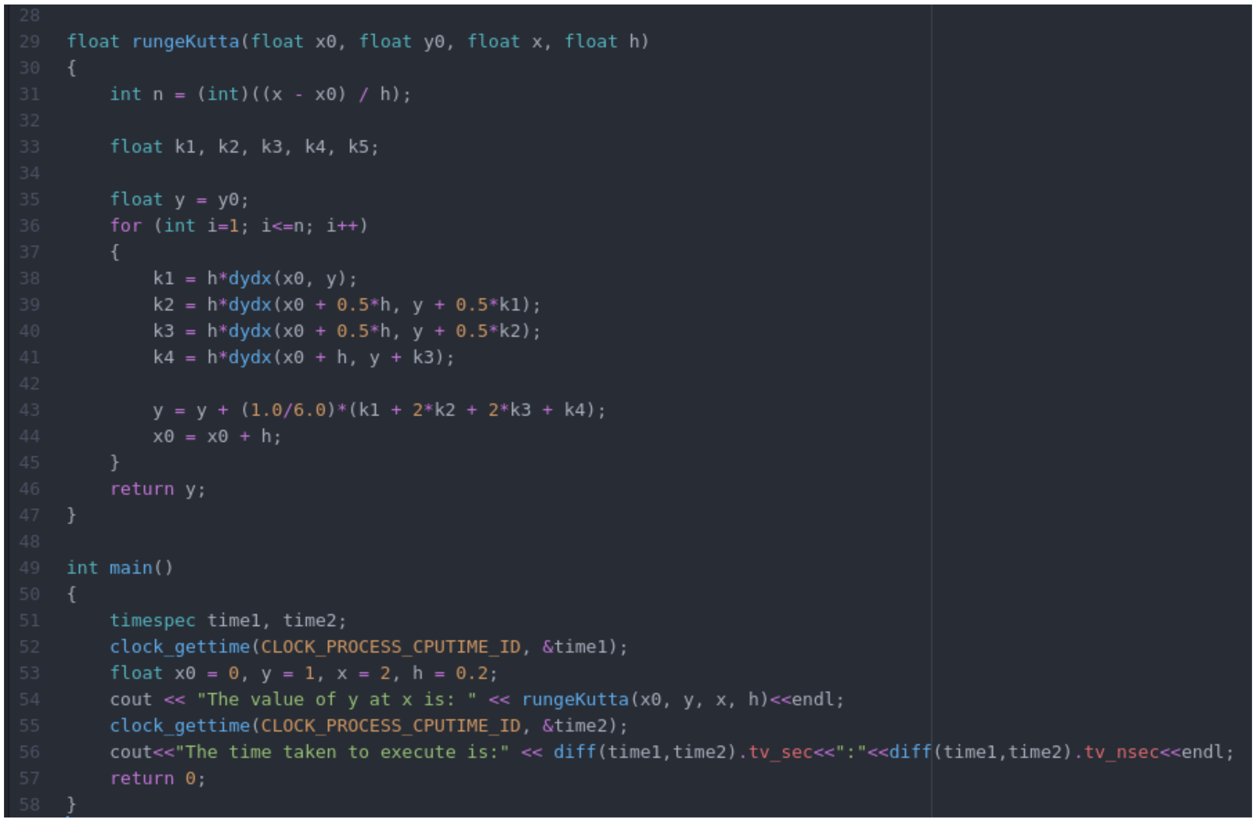
```
In [28]: using LinearAlgebra
         @timev begin
         Arr1 = rand(0:99, 1000, 1000)
         Arr2 = rand(0:99, 1000, 31000)
         Arr1_sum = sum(Arr1)
         Arr1_dsum = tr(Arr1)
         Arr3 = Arr1 * Arr2
         end

          12.599772 seconds (20 allocations: 480.653 MiB, 0.76% gc time)
         elapsed time (ns): 12599772400
         gc time (ns):      96349101
         bytes allocated:   504000832
         pool allocs:       17
         malloc() calls:    3
         GC pauses:         2
         full collections:  2
```

We note that Julia takes 12.60s to run the above program compared to 6.9s on C++.

Now, let us work on another problem where we determine the value of y at an x using the 4th order Runge-Kutta method.



```
// C++ program to implement Runge Kutta method
#include <iostream>
#include <cstdlib>
#include <stdlib.h>
#include <cmath>
# include <omp.h>
#include <time.h>
using namespace std;

timespec diff(timespec start, timespec end)
{
    timespec temp;
    if ((end.tv_nsec-start.tv_nsec)<0) {
        temp.tv_sec = end.tv_sec-start.tv_sec-1;
        temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
    } else {
        temp.tv_sec = end.tv_sec-start.tv_sec;
        temp.tv_nsec = end.tv_nsec-start.tv_nsec;
    }
    return temp;
}

// A sample differential equation "dy/dx = (x - y)/3"
float dydx(float x, float y)
{
    return((x - y)/3);
}
```



```cpp

float rungeKutta(float x0, float y0, float x, float h)
{
    int n = (int)((x - x0) / h);

    float k1, k2, k3, k4, k5;

    float y = y0;
    for (int i=1; i<=n; i++)
    {
        k1 = h*dydx(x0, y);
        k2 = h*dydx(x0 + 0.5*h, y + 0.5*k1);
        k3 = h*dydx(x0 + 0.5*h, y + 0.5*k2);
        k4 = h*dydx(x0 + h, y + k3);

        y = y + (1.0/6.0)*(k1 + 2*k2 + 2*k3 + k4);
        x0 = x0 + h;
    }
    return y;
}

int main()
{
    timespec time1, time2;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time1);
    float x0 = 0, y = 1, x = 2, h = 0.2;
    cout << "The value of y at x is: " << rungeKutta(x0, y, x, h)<<endl;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time2);
    cout<<"The time taken to execute is:" << diff(time1,time2).tv_sec<<":"<<diff(time1,time2).tv_nsec<<endl;
    return 0;
}
```

Runge Kutta method is a method used to find the approximate value of y for a given x. With the use of 4th order Runge Kutta method, only 1st order differential equations can be solved.

Output:

```
[suraj@localhost Project]$ g++ project2.cpp -o project2.out
[suraj@localhost Project]$ ./project2.out
The value of y at x is: 1.05367
The time taken to execute is:0:72148
[suraj@localhost Project]$
```

We observe that the value of y at x =2 is 1.05367. The program takes 0.7214s to execute.

Now let us look at the code to perform the same calculation using Julia.

The code for Julia can be found below:

```julia
#Code for 4th order Runge-Kutta
# A sample differential equation "dy / dx = (x - y)/3"
@timev begin
function dydx(x, y)
    return ((x - y)/3)
end

function rungeKutta(x0, y0, x, h)
    n = floor((x - x0)/h)
    y = y0
    for i in 1:n
        k1 = h * dydx(x0, y)
        k2 = h * dydx(x0 + 0.5h, y + 0.5k1)
        k3 = h * dydx(x0 + 0.5h, y + 0.5k2)
        k4 = h * dydx(x0 + h, y + k3)

        # Update next value of y
        y = y + (k1 + 2k2 + 2k3 + k4)/6

        # Update next value of x
        x0 = x0 + h
            end
    return y
    end

# Driver method
x0 = 0.0
y = 1.0
x = 2.0
h = 0.2
println("The value of y at x is: ",rungeKutta(x0, y, x, h))
    end
```

Output:

```
The value of y at x is: 1.0536687143839l1
  0.012705 seconds (51.15 k allocations: 2.689 MiB)
elapsed time (ns): 12705399
bytes allocated:   2819416
pool allocs:       51148
non-pool GC allocs:5
```

We observe that the value of y at x =2 is 1.05368 which is the same value we got from coding in C++. Julia takes 0.0127 seconds to execute compared to 0.7214s in C++.

Observation and conclusion:

The first thing to be noted is how simple and easy to understand the code of Julia is. The first problem was to calculate the sum of elements of the first matrix, the sum of elements of its diagonals as well as multiply it with another matrix. In Julia, we needed only 8 lines of code for the whole problem whereas it was 134 lines of code in C++ for the same computation. We found that C++ was significantly faster than Julia. Both for a small matrix as well as a matrix of dimension 1000X1000, C++ was faster. Almost twice as fast for the larger matrices. However, in the second problem where we calculated the value of y for a given x using 4th order Range-Kutta method, we found that Julia was significantly faster compared to C++. We observe that for computational intense problems like the first problem with multiple matrices with a large number of rows and columns, C++ seems to work faster. For relatively less computationally intense problems like the second problem, Julia lives up to its name to bring the compiling speeds of C and the ease of writing code of Python together.

References:

https://datatofish.com/add-julia-to-jupyter/

http://www.pkofod.com/2017/04/24/timing-in-julia/

https://en.wikibooks.org/wiki/Introducing_Julia/Arrays_and_tuples

https://www.geeksforgeeks.org/runge-kutta-4th-order-method-solve-differential-equation/