CS205 C/ C++ Program Design

Assignment 3

Name: Ooi Yee Jing

SID: 11910238

Part 1- Source Code

Source Code in Visual Studio 2019

Assignment3.cpp

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <ctime>
#include <time.h>
#include <chrono>
#include "DotProduct.h"
using namespace std;

int main()
{
    auto startPrep = std::chrono::steady_clock::now();
    //long n = 200000000;
    //float* v1 = new float[n];
    //float* v2 = new float[n];

    //Generate RANDOM float number for two vectors
    //srand(time(NULL));
    //for (long i = 0; i < n; i++) {
    //    v1[i] = static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX));
    //    v2[i] = static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX));
    //}

    long n;
    std::cout << "Enter the number of elements in the vectors: ";
    cin >> n;
    float* v1 = new float[n];
    float* v2 = new float[n];
    int noOfInputErrorV1 = 0;
    int noOfInputErrorV2 = 0;

    std::cout << "Enter values for Vector 1: ";
    for (long i = 0; i < n; i++) {
        cin >> v1[i];

        //The input will be changed to "1" automatically if it is an invalid
input.
        if (cin.fail())
        {
            noOfInputErrorV1++;
            v1[i] = 1;
            cin.clear(); //This corrects the stream.
            cin.ignore(); //This skips the left over stream data.
        }
        else
            continue;
    }
    if (noOfInputErrorV1 >= 1) {
        cout << "Vector 1 Input modified. All non-digit values are changed to
'1'.\n\n";
    }

    //Ignore the old overflow input to prevent wrong input into vector.
    std::cin.ignore(32767, '\n');
    std::cout << "Enter values for Vector 2: ";
    for (long i = 0; i < n; i++){
        cin >> v2[i];

        //The input will be changed to "1" automatically if it is an invalid
input.
        if (cin.fail())
        {
            noOfInputErrorV2++;
            v1[i] = 1;
            cin.clear(); //This corrects the stream.
            cin.ignore(); //This skips the left over stream data.
        }
        else
            continue;
    }
    if (noOfInputErrorV2 >= 1) {
        cout << "Vector 2 Input modified. All non-digit values are changed to
'1'.\n\n";
    }

    auto endPrep = std::chrono::steady_clock::now();
    std::cout
        << "Prepare Data took "
        << std::chrono::duration_cast<std::chrono::milliseconds>(endPrep -
startPrep).count() << "ms" << endl;

    // dotProduct function call
    try {
        DotProduct dotProduct;

        cout << "Result: " << endl;

        for (int i = 0; i < 1; i++) {

            // Method 1 (Original method)
            auto start = std::chrono::steady_clock::now();
            cout << dotProduct.CalculateValArray(v1, v2, n) << endl;
            auto end = std::chrono::steady_clock::now();
            std::cout
                << "dotProduct CalculateValArray took "
                << std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count() << "ms" << endl;
```

```
            // Method 2
            start = std::chrono::steady_clock::now();
            cout << dotProduct.CalculateImprovedV1(v1, v2, n) << endl;
            end = std::chrono::steady_clock::now();
            std::cout
                << "dotProduct CalculateInnerProduct took "
                << std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count() << "ms" << endl;

            // Method 3
            start = std::chrono::steady_clock::now();
            cout << dotProduct.CalculateImprovedV2(v1, v2, n) << endl;
            end = std::chrono::steady_clock::now();
            std::cout
                << "dotProduct CalculateImprovedV1 took "
                << std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count() << "ms" << endl;

            // Method 4
            start = std::chrono::steady_clock::now();
            cout << dotProduct.CalculateImprovedV3(v1, v2, n) << endl;
            end = std::chrono::steady_clock::now();
            std::cout
                << "dotProduct CalculateImprovedV2 took "
                << std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count() << "ms" << endl;

            // Method OpenBLAS
            start = std::chrono::steady_clock::now();
            cout << dotProduct.CalculateOpenBlas(v1, v2, n) << endl;
            end = std::chrono::steady_clock::now();
            std::cout
                << "dotProduct OpenBlas took "
                << std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count() << "ms" << endl;

        }
    }
    catch (int e) {
        cout << "input error, try again" << endl;
    }
    delete [] v1;
    delete [] v2;

    return 0;
}
```

DotProduct.cpp

```
#include <iostream>
#include <numeric>
#include <valarray>
#include "cblas.h"
#include "DotProduct.h"
using namespace std;
```

```
long float DotProduct::CalculateValArray(float v1[], float v2[], long n) {
    std::valarray<float> myV1(v1, n);
    std::valarray<float> myV2(v2, n);

    long float product = (myV1 * myV2).sum();

    return product;
}

long float DotProduct::CalculateImprovedV1(float v1[], float v2[], long n) {

    long float product = 0;
    product += std::inner_product(v1, v1 + n, v2, 0.0);

    return product;
}

long float DotProduct::CalculateImprovedV2(float v1[], float v2[], long n)
{

    long float product = 0;

    // Loop for calculate cot product
    for (long i = 0; i < n; i++) {

        product += v1[i] * v2[i];
    }

    return product;
}

long float DotProduct::CalculateImprovedV3(float v1[], float v2[], long n)
{
    long float product = 0;
    long float p1 = 0;
    long float p2 = 0;
    long float p3 = 0;
    long float p4 = 0;
    long float p5 = 0;

    // Loop for calculate cot product
    for (long i = 0; i < n; i = i + 5) {
        p1 = v1[i + 0] * v2[i + 0];
        p2 = v1[i + 1] * v2[i + 1];
        p3 = v1[i + 2] * v2[i + 2];
        p4 = v1[i + 3] * v2[i + 3];
        p5 = v1[i + 4] * v2[i + 4];

        product += p1 + p2 + p3 + p4 + p5;
    }

    return product;
}


long float DotProduct::CalculateOpenBlas(float v1[], float v2[], long n)
{
    long float product = 0;

    // Loop for calculate cot product
    product = cblas_sdot(n, v1, 1, v2, 1);

    return product;
}


long double DotProduct::CalculateOpenBlas1(float v1[], float v2[], long n)
{
    long double product = 0;

    // Loop for calculate cot product
    product = cblas_sdot(n, v1, 1, v2, 1);

    return product;
}
```

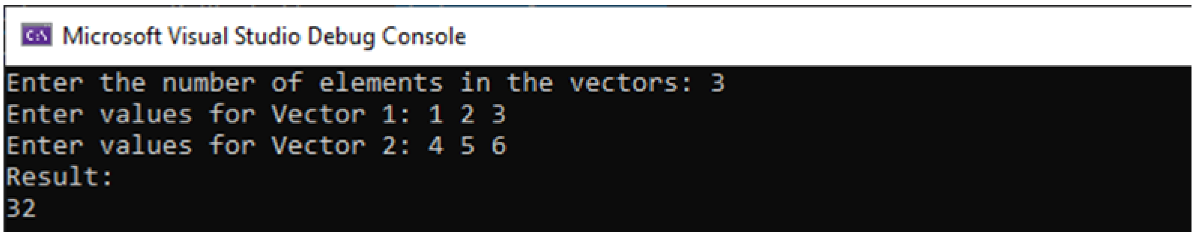
Part2 - Result & Verification

#1. Please implement a function which can compute the dot product of two vectors. The type of vector elements is float (not double). The function must be robust and cannot crash when you input something incorrectly. The definition is: dot_product = v1[0] · v2[0] + v1[1] · v2[1] + ... + v1[n-1] · v2[n-1].

Test case #1.1 - Normal Integer Value :

```
Sample Input and Output:
Input:  Vector1(v1): 1 2 3
        Vector2(v2): 3 4 5
Output: 32
```

Screen-short for case #1.1:



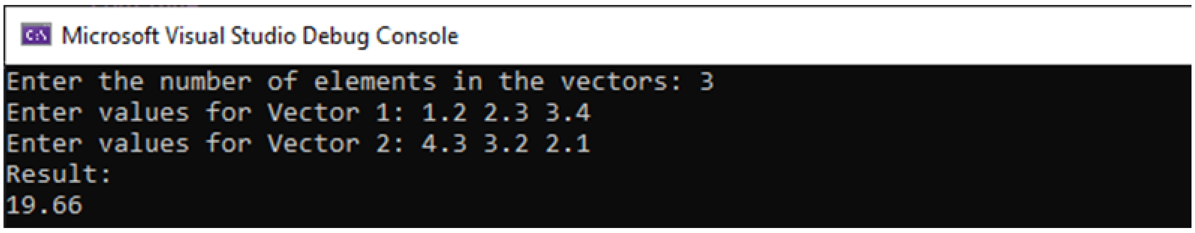
Microsoft Visual Studio Debug Console

```
Enter the number of elements in the vectors: 3
Enter values for Vector 1: 1 2 3
Enter values for Vector 2: 4 5 6
Result:
32
```

Test case #1.2 - Normal Floating Number :

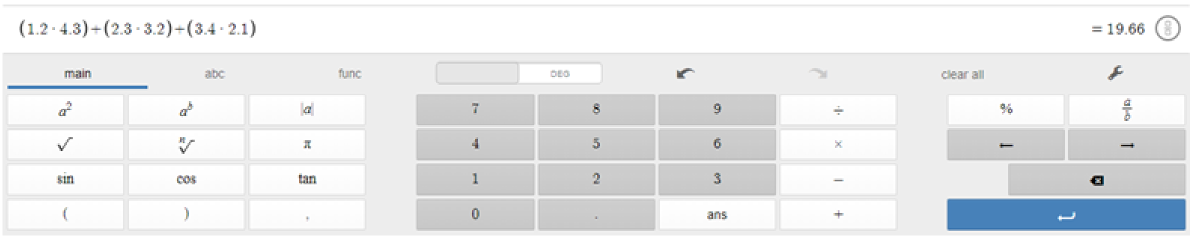
```
Sample Input and Output:
Input:  Vector1(v1): 1.2 2.3 3.4
        Vector2(v2): 4.3 3.2 2.1
Output: 19.66
```

Screen-short for case #1.2 :



```
Microsoft Visual Studio Debug Console
Enter the number of elements in the vectors: 3
Enter values for Vector 1: 1.2 2.3 3.4
Enter values for Vector 2: 4.3 3.2 2.1
Result:
19.66
```

Calculation Checking by using DESMOS Scientific Calculator :

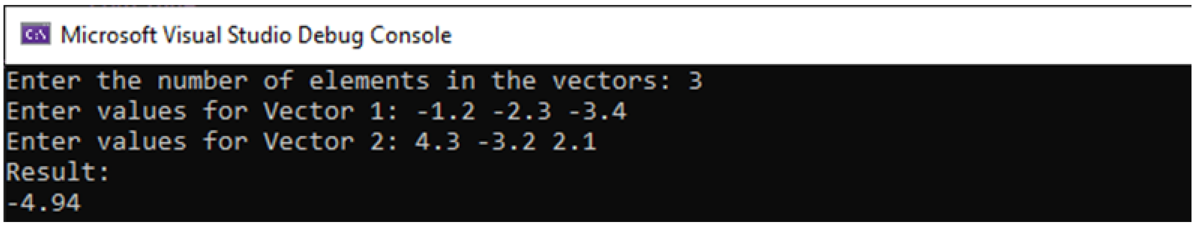


$(1.2\cdot 4.3)+(2.3\cdot 3.2)+(3.4\cdot 2.1)$ $= 19.66$

Test case #1.3 - Normal Floating Number with Negative Value:

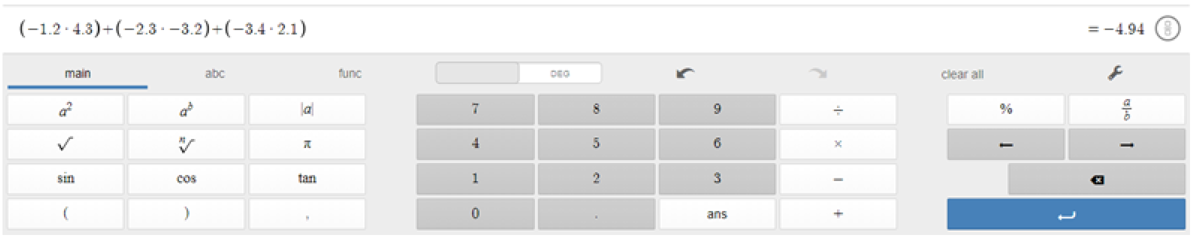
```
Sample Input and Output:
Input:  Vector1(v1): -1.2 -2.3 -3.4
        Vector2(v2): 4.3 -3.2 2.1
Output: -4.94
```

Screen-short for case #1.3 :



```
Microsoft Visual Studio Debug Console
Enter the number of elements in the vectors: 3
Enter values for Vector 1: -1.2 -2.3 -3.4
Enter values for Vector 2: 4.3 -3.2 2.1
Result:
-4.94
```

Calculation Checking by using DESMOS Scientific Calculator :

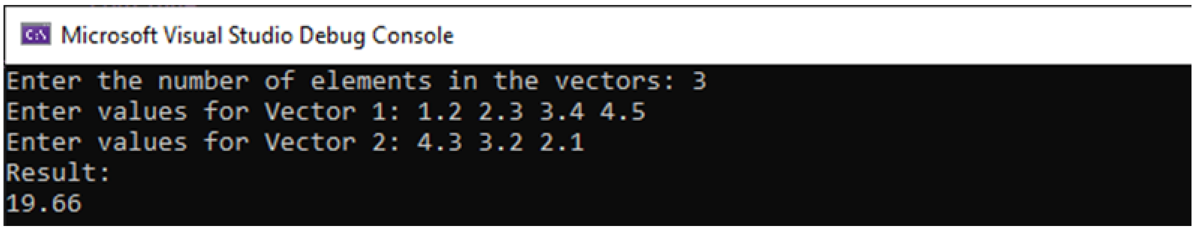


$(-1.2\cdot 4.3)+(-2.3\cdot -3.2)+(-3.4\cdot 2.1)$ $= -4.94$

Test case #1.4 - Number of Input Element in Array Exceeded :

```
Sample Input and Output:
Number of elements in a vector: 3
Input:  Vector1(v1): 1.2 2.3 3.4 4.5
        Vector2(v2): 4.3 3.2 2.1
Output: 19.66
```

Screen-short for case #1.4 :


Microsoft Visual Studio Debug Console

```
Enter the number of elements in the vectors: 3
Enter values for Vector 1: 1.2 2.3 3.4 4.5
Enter values for Vector 2: 4.3 3.2 2.1
Result:
19.66
```

Solution:

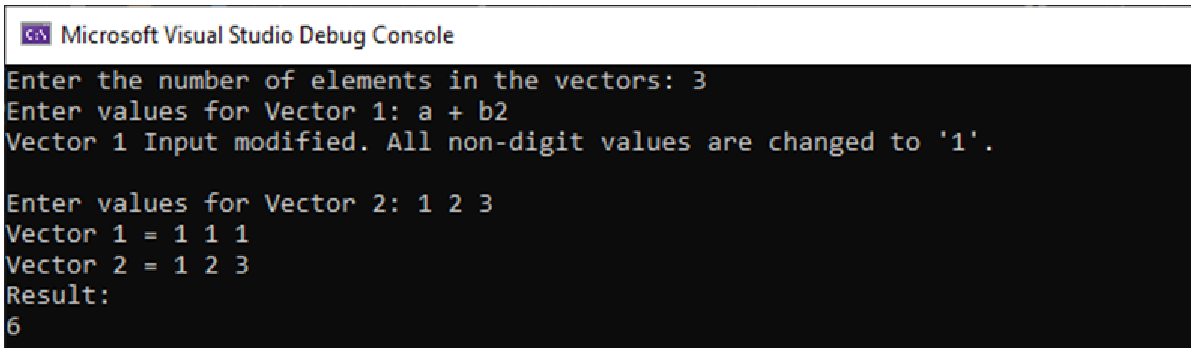
```
std::cin.ignore(32767, '\n');
```

Function “cin.ignore()” above is used before inserting the input for Vector 2 which can clear the exceeded input of Vector 1 (“4.5” in this test case #1.4) to avoid the unwanted input been added into Vector 2.

Test case #1.5 - Invalid Input (Alphabets and Symbols) :

```
Sample Input and Output:
Number of elements in a vector: 3
Input:  Vector1(v1): a + b2
        Vector2(v2): 1 2 3
Output: 6
```

Screen-short for case #1.5 :


Microsoft Visual Studio Debug Console

```
Enter the number of elements in the vectors: 3
Enter values for Vector 1: a + b2
Vector 1 Input modified. All non-digit values are changed to '1'.

Enter values for Vector 2: 1 2 3
Vector 1 = 1 1 1
Vector 2 = 1 2 3
Result:
6
```

Solution:

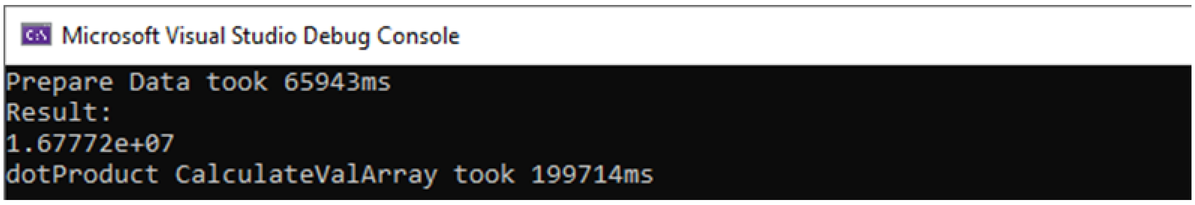
- Check the validation of input after the insertion of input.
- Change the input that is invalid (non digit input) to “1” which will not affect the calculation of dot product.

#2. Please measure how many milliseconds (or seconds) for vectors which have more than 200M elements.

Test case #2:

```
Input:
250000000(250M) random generated floating numbers
```

(One of the) Screen-short for case #2:


Microsoft Visual Studio Debug Console

```
Prepare Data took 65943ms
Result:
1.67772e+07
dotProduct CalculateValArray took 199714ms
```

	Test 1	Test 2	Test 3	Average
Time taken for Calculation (ms)	199714	195551	204436	199900.3333

Conclusion:

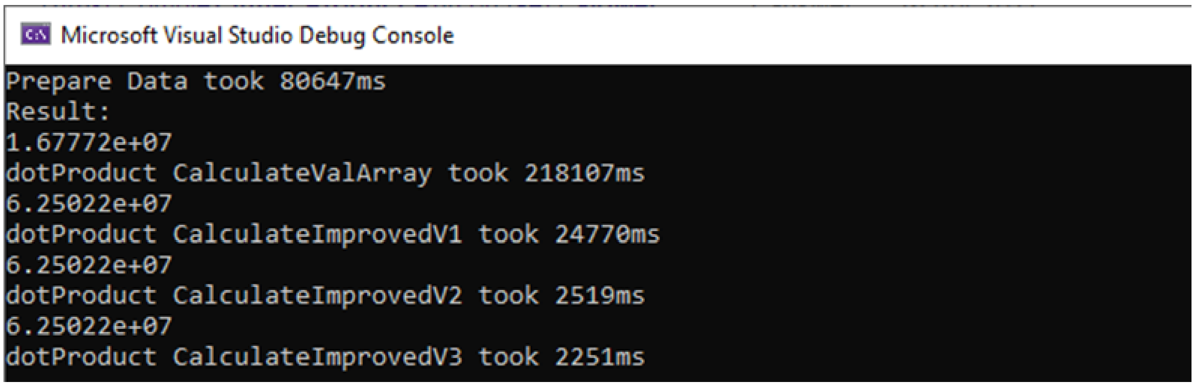
The function takes average about 199900.3333 milliseconds (3 minutes 19 seconds) to calculate the answers of the dot product when each vector contains 250M elements.

#3. Improve the efficiency of your source code. Can it be 10 times faster than your first version? Please report your methods and the improvements.

Test case #3:

```
Input:
200000000(200M) random generated floating numbers
```

(One of the) Screen-short for case #3:


Microsoft Visual Studio Debug Console

```
Prepare Data took 80647ms
Result:
1.67772e+07
dotProduct CalculateValArray took 218107ms
6.25022e+07
dotProduct CalculateImprovedV1 took 24770ms
6.25022e+07
dotProduct CalculateImprovedV2 took 2519ms
6.25022e+07
dotProduct CalculateImprovedV3 took 2251ms
```

Time taken	First Version	Improved1	Improved2	Improved3
Test 1 (ms)	218107	24770	2519	2251
Test 2 (ms)	160504	11724	1540	1198
Test 3 (ms)	169005	19909	2601	2615
Average	182538.6667	18801.0000	2220.0000	2021.3333

#3.1. First Version Method:

- Using standard library in C++

```
long float DotProduct::CalculateValArray(float v1[], float v2[], long n) {

    std::valarray<float> myV1(v1, n);
    std::valarray<float> myV2(v2, n);
    long float product = (myV1 * myV2).sum();

return product;
}
```

- It probably because of the function is spending time on the overhead of :
  - Allocated new blocks for the resulting
  - Created new for insertion of new results
  - Paged in the memory for new
  - Deallocating old which replaced by the new that containing the results of the calculation

#3.2. Improved Method 1:

- Using <inner_product> function from the numeric header

```
long float DotProduct::CalculateImprovedV1(float v1[], float v2[], long n) {

    long float product = 0;
    product += std::inner_product(v1, v1 + n, v2, 0.0);

    return product;
}
```

- Using <inner_product> is faster than the first version which using
  - This might because of it saved a lot of time without allocating, creating, initializing, and deallocating the .
- It is still slow compared to others method
  - The algorithm is using an accumulator which is deduced from the initial value

#3.3. Improved Method 2:

- Using naïve implementation of dot product

```
long float DotProduct::CalculateImprovedV2(float v1[], float v2[], long n)
{

    long float product = 0;
    for (long i = 0; i < n; i++) {
        product += v1[i] * v2[i];
    }
    return product;
}
```

- Using naïve implementation of dot product is faster than the first version which using and <inner_product>
  - Saved time without allocating, creating, initializing, and deallocating the .

    - Not using an accumulator

#3.3. Improved Method 3:

- Using for loop in the calculation to carry out batch calculation

```
long float DotProduct::CalculateImprovedV3(float v1[], float v2[], long n)
{
    long float product = 0;
    long float p1 = 0;
    long float p2 = 0;
    long float p3 = 0;
    long float p4 = 0;
    long float p5 = 0;

    // Loop for calculate cot product
    for (long i = 0; i < n; i = i + 5) {
        p1 = v1[i + 0] * v2[i + 0];
        p2 = v1[i + 1] * v2[i + 1];
        p3 = v1[i + 2] * v2[i + 2];
        p4 = v1[i + 3] * v2[i + 3];
        p5 = v1[i + 4] * v2[i + 4];

        product += p1 + p2 + p3 + p4 + p5;
    }

    return product;
}
```

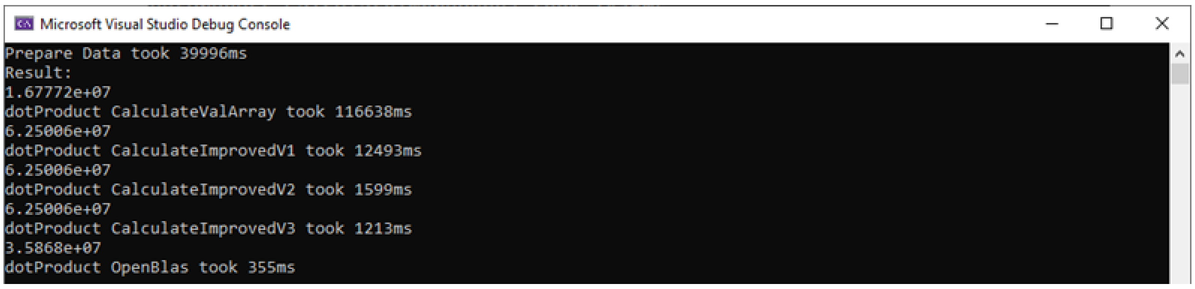
- This method is the fastest among four
    - Batch calculation helped to save the time
    - Calculated 5 elements at once instead of calculate it one by one

#4. Compare your implementation of dot product with OpenBLAS https://github.com/xianyi/OpenBLAS Please report if the two implementations return the same dot product if the input vectors are very long, and which implementation is faster and how faster it is.

Test case #4:

```
Input:
200000000(200M) random generated floating numbers
```

Screen-short for case #4:



```
Prepare Data took 39996ms
Result:
1.67772e+07
dotProduct CalculateValArray took 116638ms
6.25006e+07
dotProduct CalculateImprovedV1 took 12493ms
6.25006e+07
dotProduct CalculateImprovedV2 took 1599ms
6.25006e+07
dotProduct CalculateImprovedV3 took 1213ms
3.5868e+07
dotProduct OpenBlas took 355ms
```

Results:

- The results for all implementations of Improved Method 1, Improved Method 2 and Improved Method 3 are the same
- First version is different because of
    - is a fixed sized array or vector so that the results (product) will be incorrect if the answer is larger than the maximum value of float.
    - The results are always **1.67772e+07** which is the maximum value of floating number.
- The results of OpenBLAS Method are different which the result are always around **3.58e+07**.

Efficiency of Method:

	OpenBLAS Method
First Version	About **330 times** faster than first version
Improved Method 1	About **35 times** faster than Improved Method 1
Improved Method 2	About **4.5 times** faster than Improved Method 2
Improved Method 3	About **3.5 times** faster than Improved Method 3