Brown University
Center for Computation and Visualization
HPC Technical Assignment

Joseph R. Cooke III

June 27, 2022

1 Introduction

A C++ application to calculate the ordinary least square estimates using the Armadillo library was developed and benchmarked for performance. The following report outlines the benchmarking procedure used along with the code that corresponding to the timing points in the program.

The outline to the problem was to assume

$$y = X\beta \tag{1}$$

with the estimates to be calculated as

$$\hat{\beta} = (X^T X)^{-1} X^T y. \tag{2}$$

The completed application is used from the command line with the following syntex:

```
estOLS -x XMAT -y yVEC -o Output
```

where estOLS is the program name, XMAT is a text file containing the matrix X in a space or tab delimited form without a header, yVEC is a text file holding the vector y without a header, and *Output* is the file name for the calculated estimates output $\hat{\beta}$.

The required dependencies for the Armadillo package are g++, CMake, OpenBLAS, LAPACK, ARPACK and SuperLU. The dependencies can be installed using apt-get with the following command:

```
sudo apt-get install g++, cmake, libopenblas-dev, liblapack-dev, libarpack2-dev,
    ↪ libsuperlu-dev
```

and the Armadillo package can be installed through apt-get with the following command:

```
sudo apt-get install libarmadillo-dev
```

2 Benchmark

The benchmarking for the application was completed using pregenerated random value matrices, XMAT, and vector, yVEC, saved into .txt format.

Timing was completed using the chrono class in C++ with the following timing metrics to understand where in the program the time was spent. The total elapse time for the program is defined as a measure from the first line of main() to when the timing is displayed in the terminal window which is the last call before the program ends. The XMAT read in time is measured from the opening of the XMAT text file to when the XMAT text file is closed which encapsulates the procedure outlined in Figure 2. The yVEC read in time is measured from the opening of the yVEC text file to when it closes the yVEC text file which is outlined if Figure 3. The calculation of the ordinary least squares estimates is measured from declaring the matrices and vectors required to perform the calculation to when it returns the estimates as outlined in Figure 4. The data writing time is measured from the opening of the text file to it closing the output data file as outlined in Figure 5. The individual functions are timed to know the most costly features of the program for future optimization purposes and to determine cost effectiveness of the program. The test system was an Intel Core i7-8550 CPU with 8 GB of system memory.

Table 1 displays the total elapse times for the program to run under the tested matrices. As observed in Table 1, as the *dim m* increases in size, the more time the application takes to run. To further understand the time spent in the program, Table 2 display the individual timing for each of the components of the program.

XMAT (dim n x dim m)	**Total Elapse Time** (milliseconds)
(20000,500)	3088
(20000,1000)	6434
(20000,2000)	15412
(20000,10000)	170041

Table 1: Total elapse time for estOLS program ran on the test system for the prescribed matrix sizes.

As observed in Table 2, the read in of the yVEC text file doesn't vary in all by the last test because its size doesn't vary. The variance in the last test is not significant to investigate. It is also observed that the writing of the estimates to the text file doesn't account for the majority of the time in the application. The two most costly pieces of code is the reading in of the XMAT file and the calculation of the estimates. The reading in of the XMAT file for the first three tested sizes costs the largest amount of computational resources as expected due to the number of points being read and stored to memory. Similarly, as the size of the X matrix increases, the calculation of the estimates also increases and dramatically increases in the final test size.

Optimization and further work would be most beneficial in the reading in the of XMAT file and the calculation of the estimates.

The data sets in Tables 1 and 2 were also plotted in Figure 1 for visual clarity.

XMAT (dim n x dim m)	**XMAT Read In** (milliseconds)	**yVEC Read In** (milliseconds)	**Estimator Calculation** (milliseconds)	**Data Write** (milliseconds)
(20000,500)	2687	6	330	7
(20000,1000)	5426	6	876	15
(20000,2000)	12338	6	2845	21
(20000,10000)	66092	7	102458	283

Table 2: Individual timing segments for each part of the estOLS program ran on the test system for the prescribed matrix sizes.

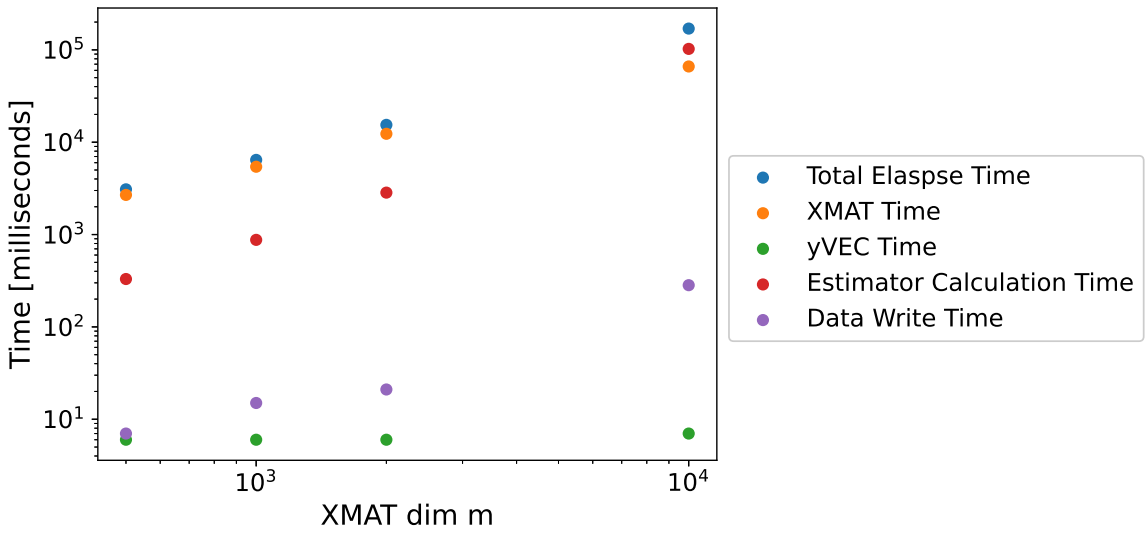

Figure 1: Timing data from Tables 1 and 2 from runs on test system.

3 Code

The following section contain the functions that are called within main() of the program estOLS. These code blocks are here for reference to the benchmarking of the individual components discussed in the aformentioned section.

```
mat ReadXMAT(string xMAT_input){
        // Opening XMAT text file
        ifstream inputX;
        inputX.open(xMAT_input,ios::in);

        // Declaring storage matrix
        mat X;

        // Storing data from text file to storage matrix
        X.load(inputX, raw_ascii);

        // Closing input file
        inputX.close();

        // Returning filled matrix
        return X;
}
```

Figure 2: ReadXMAT function

```
vec ReadyVEC(string yVEC_input){
        // Opening yVEC text file
        ifstream inputY;
        inputY.open(yVEC_input,ios::in);

        // Declaring storage vector
        vec Y;

        // Storing data from text file to storage vector
        Y.load(inputY, raw_ascii);

        // Closing input file
        inputY.close();

        // Returning filled vector
        return Y;
}
```

Figure 3: ReadyVEC function

```
vec CalculateBeta(mat X, vec Y){
        // Declaration of matrix and vector variables
        mat XT, XTX;
        vec Beta;

        // Taking the transpose of the input matrix X
        XT = X.t();

        // Completing the matrix multiplication
        XTX = XT*X;

        // Calculating Beta
        Beta = (inv(XTX))*XT*Y;

        // Returning Beta
        return Beta;
}
```

Figure 4: CalculateBeta function

```
void DataWrite(string output_name, vec Beta){
        // Opening output file
        ofstream outputData;
        outputData.open(output_name,ios::out);

        // Need to specify arma because size() exists in both namespaces
        // Setting bounds for for loop to save data
        int m = arma::size(Beta)[0];

        // Writing to output file
        for(int i = 0; i < m; i++){
                outputData << Beta[i] << endl;
        }

        // Closing output file
        outputData.close();
}
```

Figure 5: DataWrite function