# Using Linear Algebra Libraries in C++ and Python

Melvyn Ian Drag

December 15, 2014

**Abstract**

In this paper I will describe what BLAS, LAPACK, Armadillo, Eigen, and OpenMP are. Then I will show how to write C++ code which uses these libraries to perform common linear algebra operations like matrix-matrix, and LU factorization. I will then compare the run_time of the codes to some cache efficient code I have written to (hopefully) show the incredible speed ups possible with these libraries.

Many people on stackoverflow.com are clamoring for a paper such as this one. I plan on posting a link to this paper in response to several questions I have seen online.

I will also show how these libraries are utilized in a correctly built scipy/numpy package, demonstrating that scientific python codes can indeed run fast. In fact, my Python code outperforms my C++ code!

*I looked around and could only find these two webpages with some simple examples. I wish more people would add examples of codes to use these libraries.* - makhlaghi, Stack-overflow.com user.

This paper is for makhlagi, and for all of those like you (including myself) who have been confused about how to use these popular but suprisingly poorly documented libraries! In this paper are some simple sample codes which demonstrate how to incorporate various linear algebra packages into your codes, and then how to compile with them.

# Contents

# Chapter 1

# Introduction

## 1.1 What Are These Libraries?

First, lets clarify what the libraries we are using actually are, since this alone can generate some confusion. We will first consider the C++ libraries.

The **BLAS** (Basic Linear Algebra Subprogams) are a set of highly optimized routines for common linear algebra computations like matrix-vector and matrix-matrix multiplication, Givens rotations, and norm computation. **LAPACK** (Linear Algebra PACKage) is a set of higher linear algebra routines for things like matrix factorizations, eigenvalue problems, and linear system solvers. LAPACK is written to perform most computations using the BLAS library, so that they are fast. **Armadillo** and **Eigen** are open source linear algebra libraries for C++ which are often used in industry. Both of these packages are essentially front ends to the BLAS and LAPACK.

Python programmers - or 'Pythonistas', as they prefer to be called - have a set of guiding principles[1]. One of them is that "There should be one – and preferably only one – obvious way to do it." This is of course in contrast to C++ in which, when you need to do something as rudimentary as multiply two matrices, you have hundreds of available linear algebra libraries, many with similar performance, and many with awfully confusing syntax. This all being said, it should not surprise us that there are very few ways of doing linear algebra computations in Python. We will experiment with the **Numpy** and **Scipy** libraries which are the numerical and scientific computing libraries, respectively, and see how one can also use the **BLAS** and **LAPACK** routines in a .py file.

## 1.2 Format

We will begin each section with a piece of code with no explanation. We will see its results and the comparison between its run time and the run time of competing libraries. Then we will begin to look in depth at the syntax of the code to see the pros and cons.

All of the code in this paper can be found online at `https://github.com/JulianCienfuegos/linear_alg_final`. I encourage you, the reader, to play with this code yourself on your computer. I even include compilation instructions! Julian Aureliano Cienfuegos is the pseudonym and online identity of Melvyn Drag.

[1]https://www.python.org/dev/peps/pep-0020/

# Chapter 2

# Dense Matrix Multiplication

## 2.1 C++

### 2.1.1 Code

**This is the code we will be discussing in this section.**

```
/*
 * This code surveys matrix multiplication in C++
 *
 *
 * BLAS
 * Tiled MM on one core
 * Tiled MM on multiple cores
 * Armadillo MM
 * Eigen MM
 *
 * are what we will experiment with.
 *
 * Compilation Instructions:
 * g++ -I /usr/local/include/eigen3 CompareMM.cpp -larmadillo -lblas -o MM
 *
 * Where -I /usr/local/include/eigen3 is the path to your Eigen directory.
 *
 */

#include <stdio.h>
#include <ctime>
#include <armadillo>
#include <omp.h>
#include <Eigen/Dense>

#define bSize 64
#define NITER 1
int N = 1024;
```

```
/*
 * The function dgemm() performs one multiplication of the form
 * C = alpha*A*B + beta*C. Below we will set alpha to 1 and beta to zero.
 */

extern ``C" void dgemm_(char *transa, char *transb, int *m, int *n, int* k,
                        double *alpha, double *A, int *lda, double *B, int *ldb,
                        double *beta,  double *C, int *ldc );


void BMM(double **, double **, double **);
void omp_BMM(double **, double **, double **);

using namespace std;
using namespace arma;
using Eigen::MatrixXd;

int main(int argc, char * argv[])
{
/*********************** MAKE MATRICES *************************/
/*                           C++ 2D ARRAY                        */
double ** A_block = new double * [N];
double ** B_block = new double * [N];
double ** C_block = new double * [N];

for(int i = 0; i < N; i++)
{
        A_block[i] = new double [N];
        B_block[i] = new double [N];
        C_block[i] = new double [N];
}
srand(time(NULL));
for(int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
                A_block[i][j] = (double)(rand()%100);
                B_block[i][j] = (double)(rand()%100);
        }

/*                          C++ 1D ARRAY                         */
double * A_blas = new double [N*N];
double * B_blas = new double [N*N];
double * C_blas = new double [N*N];
for (int i = 0; i < N*N; i++)
{
        A_blas[i] = (double)(rand()%100);
        B_blas[i] = (double)(rand()%100);
}

/*                         ARMADILLO MATRICES                         */
mat A = randu<mat>(N, N);
mat B = randu<mat>(N, N);
mat C = zeros(N,N);

/*                          EIGEN MATRICES                            */

MatrixXd A_eigen = MatrixXd::Random(N,N);
MatrixXd B_eigen = MatrixXd::Random(N,N);
MatrixXd C_eigen = MatrixXd::Zero(N,N);

/*                         OTHER VARIABLES                            */
double run_times[5];
clock_t start, stop;
char trans = 't';
double alpha = 1;
double beta = 0;
/*                          BEGIN COMPARE!                            */

printf("Running BMM...\n");
start = clock();
for(int i = 0; i < NITER; i++)
        BMM(A_block, B_block, C_block);
stop = clock();
run_times[0] = (double)(stop - start)/(NITER*CLOCKS_PER_SEC);

printf("Running omp_BMM...\n");
start = clock();
for(int i = 0; i < NITER; i++)
        omp_BMM(A_block, B_block, C_block);
stop = clock();
run_times[1] = (double)(stop - start)/(NITER*CLOCKS_PER_SEC);

printf("Running Armadillo MM...\n");
start = clock();
for(int i = 0; i < NITER; i++)
        C = A*B;
stop = clock();
run_times[2] = (double)(stop - start)/(NITER*CLOCKS_PER_SEC);

printf("Running BLAS...\n");
start = clock();
```

```
for(int i = 0; i < NITER; i++)
        dgemm_(&trans, &trans, &N, &N, &N, &alpha, A_blas, &N, B_blas,\
                &N, &beta, C_blas, &N);
stop = clock();
run_times[3] = (double)(stop - start)/(NITER*CLOCKS_PER_SEC);

printf("Running Eigen...\n");
start = clock();
for(int i = 0; i < NITER; i++)
        C_eigen = A_eigen*B_eigen;
stop = clock();
run_times[4] = (double)(stop - start)/(NITER*CLOCKS_PER_SEC);

printf("BMM: %f\nomp_BMM: %f \nArmadillo: %f\nBLAS: %f\nEigen: %f\n", \
        run_times[0], run_times[1], run_times[2], run_times[3], run_times[4]);

}

void BMM(double ** A, double ** B, double ** C)
{
/* Block Matrix Multiply */
for(int i = 0; i < N; i += bSize)
        for(int k = 0; k < N; k += bSize)
                for(int j = 0; j < N; j += bSize)
                        for(int it = i; it < i+bSize; it++)
                                for(int kt = k; kt < k+bSize; kt++)
                                        for(int jt = j; jt < j+bSize; jt++)
                                        {
                                                C[it][jt] += A[it][kt]*B[kt][jt];
                                        }
}


void omp_BMM(double ** A, double ** B, double ** C)
{
int i, j, k, it, jt, kt;
/* openMP version of BMM */
#pragma omp parallel shared(A,B,C), private(i,j,k, it, jt, kt)
{
#pragma omp for schedule(static)
for(int i = 0; i < N; i += bSize)
        for(int k = 0; k < N; k += bSize)
                for(int j = 0; j < N; j += bSize)
                        for(int it = i; it < i+bSize; it++)
                                for(int kt = k; kt < k+bSize; kt++)
                                        for(int jt = j; jt < j+bSize; jt++)
```

```
                                                        C[it][jt] += A[it][kt]*B[kt][jt];
}
}
```

**Results**

```
(With -O3 optimization)
BMM: 1.654067
omp_BMM: 1.621354
Armadillo: 1.439744
BLAS: 1.430144
Eigen: 0.644094
(With no compiler optimization)
BMM: 7.912810
omp_BMM: 7.815276
Armadillo: 1.446131
BLAS: 1.428944
Eigen: 24.925716
```

### 2.1.2 Tiled Matrix multiplication

This is an implementation of matrix multiplication written in pure C++, which uses tiling to achieve better temporal locality of data. The computer has a fast cache memory closer to the CPU registers than the main memory, and every time a bit of main memory is accessed, the computer brings a whole chunk of adjacent memory to the cache. What tiling achieves is to break the matrix multplication up into multiplications of sub matrices by sub-matrices, such that the chunks which the computer caches are not wasted, but are used over and over again until tey are not needed. For example, consider the multiplication AB, where A and B are NxN matrices. The element in the upper left corner of A will eventually be multiplied by every element in the first row of B. What tiling acheieves is that the indicated element of A is multiplied by many elements of B in a very short period of time, so that this element does not need to be fetched from main memory N times, once for each of the N elements in the first row of B.

My implementation is fairly straight forward. I allocate some square 2d arrays and then multiply them using the traditional matrix multiplication for loops, except that the three inner for loops work only on a subset of data specified by the outer for loops.

### 2.1.3 Parallelized MM using OpenMP

OpenMP is the easiest library to use to parallelize C++ code. All one needs to do is include the headerfile omp.h, add one or two lines of code before a for loop, and then the code will run on all of the cores of a multicore processor. Unfortunatly, in the above example, little performance increase is seen. This is likely due to "False Sharing", which is what occurs when multiple processors concurrently try to write to adjacent locations in global memory. Since each processor has its own cache, and memory is brought to the cache in chunks, if processor 1 is working on A[0] and processor 2 is writing to A[1], no parallelization is possible. Let me explain more. Processor 1 will write its work to A[0], and send the chunk A[0]...A[j] back to global memory, where A[0] ... A[j] is the chunk of A fetched by the cache. Processor 2 will then be unable to send its results back to global memory since processor 2 also fetched A[0]...A[j]. and the value of A[0] which it has does not match the global value. Processor 2 thus checks out the same memory block, redoes its computation, and then sends this value back to memory. Only now have both A[0] and A[1] been updated. The code can thus run slower than the non parallelized code, due to the excess memory transfers. This can be alleviated with relatively simple loop transformations, but I haven't spent time on that. I chose to simply present OpenMP as an option which the interested reader can pursue. If we were to write a clever memory access pattern for the matrix multiplication, there would be no competition from the other libraries, because they all run on one core. This is a major drawback to BLAS, and can be attributed to its having been written a long time ago when multicore architectures were not available. There are some parallel BLAS implementations available online, and even some which work with CUDA, the C-based GPU programming language. These parallel BLAS implementations boast impressive performance but are unfortunately outside of the scope of this paper.

### 2.1.4 BLAS

The BLAS are a set of old FORTRAN routines that have been highly optimized over time. In addition to being fast, they are also portable and can be run on many processors. One interesting aspect of the BLAS is the fact that they may give ¡seemingly¿ erroneous results if used in a C++ code. This is because FORTRAN uses column major storage of matrices, whereas C++ uses row major storage. You will notice in the sample code that dgemm takes a pointer to a 1d array as an arguement. Depending on how you want to output the results, you should enter the entries of your matrices carefully. If you enter them in a row major fashion, your output will be transposed. If you decide to circumvent this by using the TRANS argument to dgemm, you have to be careful with the values of LDX, which are described below.

The arguments to the mysteriously named dgemm are:

1. TRANSX - This is whether or not you want the matrix X transposed. Choices are 'N' for No, 'T' for Transpose, and 'C' for Conjugate Transpose.

2. M - This specifies the number of rows in A and C

3. N - This specifies the number of columns in B and A

4. K - This specifies the number of columns of A and, consequently, the rows of B.

5. LDX - This specifies the leading dimension of matrix X. This depends upon whether or not you use transposition. For example, if TRANSA = 'T', then LDA=N. If TRANSA='N', then LDA=M.

6. ALPHA, BETA - These are coefficients which correspond the the operation performed by dgemm(), namely, $C = \alpha A * B + \beta C$. In the sample code, $\beta$ is set to zero so we simply perform a matrix multiplication.

7. A, B, C - These are the matrices we are using. These must be matrices of double precision floats, stored as a 1D array.

### 2.1.5 Armadillo

Armadillo is a widely used linear algebra package which is largely built upon the BLAS and LAPACK, and which offers a convenient MATLAB-like syntax to the user. You will notice in the above code that sprawling lines of code are needed for the tiled matrix multiplication, and that the confusing syntax which is required to implement dgemm makes it frightening to the enduser. All of this complication boils down to four easy to read lines when the matrix product is coded using Armadillo. Operators are nicely overloaded in Armadillo, too, such that if one wants to see the contents of matrix A one can use the familiar cout ¡¡ A ¡¡ endl, and the output is nicely formatted. In addition, matrix multiplication uses *, which makes it easy to code (even the super user-friendly language Python 'requires' users to use dot(A,B) to compute the product A*B).

Since Armadillo is essentially a front end to the BLAS, we should see very little performance difference between it and the pure BLAS code. Indeed, the overhead generated by Armadillo is quite small, and the run times are about the same for both.

### 2.1.6 Eigen

The Eigen syntax is not terribly complicated, but is more confusing than that of Armadillo. In addition, downloading and linking Eigen is slightly more complicated in that you have to include a header file and provide a symlink to the directory where certain functions are stored when you compile. For the user not thoroughy comfortable with a UNIX environment, this is a serious hassle. The good news is, however, that Eigen provides not only common linear algebra operations, but also comes with a whole suite of overloaded operators which make using the library a real pleasure. These are the reasons why Eigen is used by so many production quality software development teams. The performance if Eigen is very mysterious, however. It seems that Eigen requires a great deal of overhead which O3 optimization miraculously gets rid of - notice that the run times with and without optimization are two orders of magnitude apart! While Eigen is easy to use, the fact that the run time with and without optimizatin is so disparate make one wonder what exactly the compiler is throwing away and if it is safe. O3 optimized Eigen code runs much faster than optimized BLAS code, and seeing how Eigen is based on BLAS, one ought to feel some unease with the shortcuts taken by the compiler. In short, Eigen may not be trustworthy.

## 2.2 Python

### 2.2.1 Code

**Here is the code we will be discussing.**

```
""" In this script we will see how to compute matrix − matrix products
using numpy and scipy.linalg.BLAS

From the output we see the python methods are similar.
"""

from numpy import matrix, array, dot, zeros
from numpy.random import random
from scipy.linalg.blas import dgemm
```

```
import time

N = 1024
NITER = 30

A_array = random((N,N))
B_array = random((N,N))
C_array = zeros((N,N))

start = time.time()
for it in range(NITER):
        C_array = dot(A_array,B_array)
stop = time.time()

print"Time for 2d array dot product is:", (stop - start)/NITER

A_mat = matrix(A_array)
B_mat = matrix(B_array)
C_mat = matrix(C_array)

start = time.time()
for it in range(NITER):
        C_mat = A_mat*B_mat
stop = time.time()
print "Time for Matrix multiplication is:", (stop-start)/NITER

alpha = 1
start = time.time()
for it in range(NITER):
        C_array = dgemm(alpha, A_array, B_array)
stop = time.time()
print "Time for dgemm is:", (stop-start)/NITER
```

**Results**

```
Time for 2d array dot product is: 0.299135661125
Time for Matrix multiplication is: 0.304835136731
Time for dgemm is: 0.367184797923
```

### 2.2.2 A quick comparison

It is easy to talk about Python code, because it is so succinct. In this code I compare matrix multiplication using numpy 2d arrays, numpy 2d matrices, and the low-level BLAS function dgemm. One right away notices that the array and matrix multiplications run in exactly the same amount of time. This is because numpy matrices are nothing more than numpy arrays with a bit more structure. One can quickly see this since the matrices above are instantiated by copying arrays. Indeed, the Numpy source code shows that Matrices and Arrays are fundamentally the same object.

Why do both exist, then? I am not sure why Numpy includes a Matrix class. The numpy website actually recommends that users stay away from matrices as they are not as heavily tested as the matrices are. In fact, the numpy website offers a quick answer the the question “Should I use arrays or matrices?” “Short Answer: Use Arrays”.[1] In the above code one can see that the matrix multiplication is done with arrays using the dot() function, and with matrices using the convenient *. Since matrix multiplication is so much more intuitive with the matrix class, one might be tempted to use matrix all the time instead of the array class. Do so at your own risk, because the Numpy developers say not to use the use matrices, even though they created matrices and put them within the reach of the user. See figure 1.

You can also see that dgemm is a bit slower than the built in Python functions. This is rather strange due the fact that the numpy routines are built upon BLAS and LAPACK. This, just like the Eigen example, needs much further investigation. The really interesting bit of information is that the *Python code runs much faster than the C++ code* in either case! This is exactly the opposite of our intuition and dispells the common stereotype that Python is slow and useless for scientific computing. In these examples, the Python optimizer is able to far outdo the g++ compiler! (Unfortunaely there is no way to link Eigen and Armadillo to the icc compiler, so I was unable to see how fast my code would be in that case. Typically, ICC far outdoes g++).

[1]`http://wiki.scipy.org/NumPy_for_Matlab_Users`

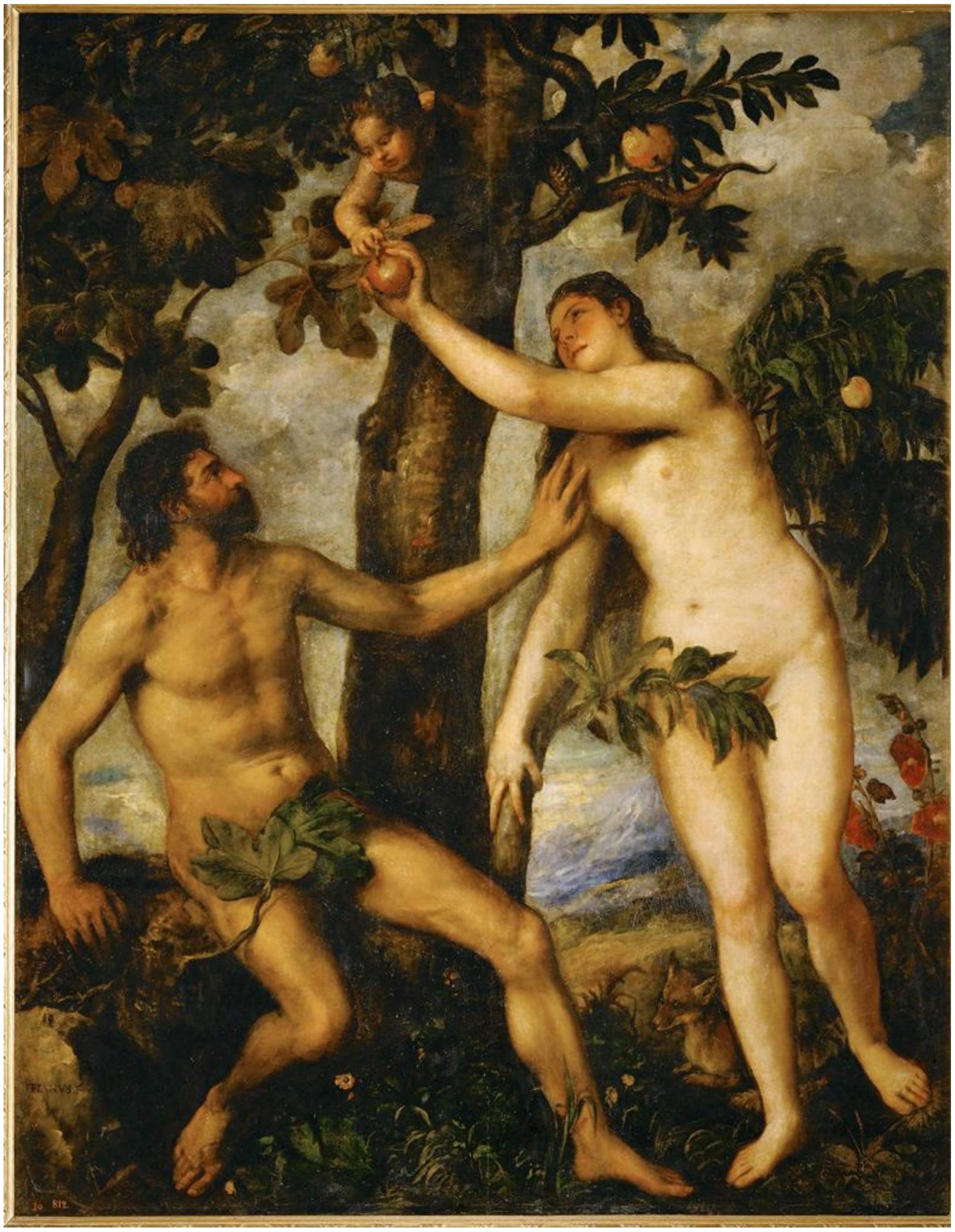

Figure 2.1: Look But Don't Touch

# Chapter 3

# LU Factorization

## 3.1 C++

### 3.1.1 Code

**Here is the code we will be using**

```
/*
 * This code surveys C++ linear system solvers which use the LU
 * Factorization. We will consider:
 *
 * BLAS
 * An LU factorization and solver which I wrote.
 * Armadillo
 * Eigen
 *
 * Compile with: g++ -I /usr/local/include/eigen3 CompareLU.cc \
 *              -larmadillo -llapack -O3 -o CompareLU
 */

#include <iostream>
#include <cmath>
#include <ctime>
#include <cstdlib>
#include <armadillo>
#include <Eigen/Dense>

int N = 1024;
const int num_iters = 10;

extern "C" void dgetrf_(int* M, int* N, double* a, int* lda, \
        int* ipiv, int* info);
extern "C" void dgetrs_(char *TRANS, int *N, int *NRHS, double *A, \
        int *LDA, int *IPIV, double *B, int *LDB, int *INFO );
```

```
using namespace std;
using namespace arma;
using Eigen::MatrixXd;
using Eigen::PartialPivLU;

void lufac(double **, double **);
void fSub(double **, double *, double *);
void bSub(double **, double *, double *);
void MVMult(double **, double *, double *);

int main()
{
        /********************* ALLOCATE MEMORY **************************/
        /*                      ARRAYS FOR MY SOLVER
*/
        double ** A = new double * [N];
        double ** P = new double * [N];
        double ** L = new double * [N];
        double ** U = new double * [N];
        double * b = new double [N];
        double * b_hat = new double [N];
        double * y = new double [N];
        double *x = new double [N];

        for(int i = 0; i < N; i++)
        {
                A[i] = new double [N];
                P[i] = new double [N];
                L[i] = new double [N];
                U[i] = new double [N];
        }

        srand((signed)time(NULL));

        for (int i = 0; i < N; i++)
        {
          for (int j = 0; j < N; j++)
          {
                  A[i][j] = rand() % 100 + 1;
                  L[i][j] = 0;
                  U[i][j] = 0;
                  if(i == j) // Make the identity matrix
                        P[i][j] = 1;
                  else
                        P[i][j] = 0;
          }
        }

        for (int i = 0; i < N; i++)
        {
                b[i] = rand() % 100 + 1;
        }
        /*                      BLAS SOLVER ARRAYS
*/
        double * A_blas = new double [N*N];
        double * b_blas = new double [N];
        int ipiv[N];
        for (int i = 0; i < N*N; i++)
                A_blas[i] = (double)(rand()%100);
        for (int i = 0; i < N; i++)
                b_blas[i] = (double)(rand()%100);
        /*                    ARMADILLO MATRICES
*/
        mat A_arma = randu<mat>(N, N);
        mat P_arma, L_arma, U_arma;
        mat b_arma = zeros<mat>(N,1);
        mat x_arma = zeros<mat>(N,1);

        /*                      EIGEN MATRICES
*/
        MatrixXd A_eigen = MatrixXd::Random(N,N);
        MatrixXd x_eigen = MatrixXd::Zero(N,1);
        MatrixXd b_eigen = MatrixXd::Random(N,1);
        /*                      OTHER VARIABLES
*/
        char trans = 'N';
        int dim = N;
        int nrhs = 1;
        int LDA = dim;
        int LDB = dim;
        int info;
        double run_times[4];
        clock_t start, stop;

        /************************ MY SOLVER ***************************/
        printf("Running My Solver...\n");
        start = clock();
        for(int iter = 0; iter < num_iters; iter++)
        {
                lufac(A,P);

                /* Get U */
        for (int i = 0; i < N; i++)
        {
                for(int j = i; j < N; j++)
                {
                        U[i][j] = A[i][j];
                }
        }

        /* Get L */
        for(int i = 0; i < N; i++)
        {
                L[i][i] = 1;
                for(int j = i-1; j >=0; j--)
                {
                        L[i][j] = A[i][j];
                }
        }

        /*Now we multiply b by P and solve PAx = Pb => LUx = Pb*/
        MVMult(P, b, b_hat);

        /* Solve Ly = b_hat */
        fSub(L, y, b_hat);

        /* Solve Ux =  y */
        bSub(U, x, y);

        /* Now we can do what we want with the x! */
}
stop = clock();
run_times[0] = (double)(stop - start)/(num_iters*CLOCKS_PER_SEC);
/* We can clean up a bit */
for (int i = 0; i < N; i++)
{
        delete[] A[i];
        delete[] P[i];
        delete[] L[i];
        delete[] U[i];
}
delete[] A;
delete[] P;
delete[] L;
delete[] U;
delete[] b;
delete[] b_hat;
delete[] x;
```

```
delete[] y;

/********************** BLAS SOLVER *****************************/
printf("Running BLAS...\n");
start = clock();
for(int iter = 0; iter < num_iters; iter++)
{
        for (int i = 0; i < N; i++)
                b_blas[i] = (double)(rand()%100);
        dgetrf_(&dim, &dim, A_blas, &LDA, ipiv, &info);
        dgetrs_(&trans, &dim, &nrhs, A_blas, &LDA, \
                ipiv, b_blas, &LDB, &info);
}
stop = clock();
clock_t start1 = clock();
for(int i = 0; i < num_iters; i++)
{
        for (int i = 0; i < N; i++)
                b_blas[i] = (double)(rand()%100);
}
clock_t stop1 = clock();
run_times[1] = (double)(stop - start)/(num_iters*CLOCKS_PER_SEC);
run_times[1] -= (double)(stop1 - start1)/(num_iters*CLOCKS_PER_SEC);
delete[] A_blas;
delete[] b_blas;

/*********************** ARMADILLO ******************************/
printf("Running Armadillo...\n");
start = clock();
for(int iter = 0; iter < num_iters; iter++)
{
        lu(L_arma, U_arma, P_arma, A_arma);
        x_arma = solve(trimatu(U_arma), solve(trimatl(L_arma), \
                P_arma*b_arma));
}
stop = clock();
run_times[2] = (double)(stop - start)/(num_iters*CLOCKS_PER_SEC);

/************************* EIGEN ********************************/
printf("Running Eigen...\n");

start = clock();
for(int iter = 0; iter < num_iters; iter++)
{
        x_eigen = A_eigen.lu().solve(b_eigen);
}
```

```
        stop = clock();
        run_times[3] = (double)(stop - start)/(num_iters*CLOCKS_PER_SEC);

        /******************** PRINT RESULTS *****************************/
        printf("My Solver: %f\nBLAS: %f \nArmadillo: %f\nEigen: %f\n", \
                run_times[0], run_times[1], run_times[2], run_times[3]);
}

void lufac(double ** A, double ** P)
{
        for(int i = 0; i < N-1; i++) // loop over all columns except the last.
        {

                double max = abs(A[i][i]);
                int maxIdx = i; //row i has the max right now
                for(int j = i+1; j < N; j++) //loop over rows
                {
                        if(abs(A[j][i]) > max){maxIdx = j;} // Find max entry
                }
                if(maxIdx != i) // Perform a swap if the max isn't on the diagonal.
                {
                        for(int col = 0; col < N; col++) //swap rows of A and P
                        {
                                double temp = A[i][col];
                                A[i][col] = A[maxIdx][col];
                                A[maxIdx][col] = temp;
                                double temp2 = P[i][col];
                                P[i][col] = P[maxIdx][col];
                                P[maxIdx][col] = temp2;
                        }
                }

                for (int j = i+1; j < N; j++)
                {
                        A[j][i] = A[j][i]/A[i][i];
                }
                for(int j = i+1; j < N; j++)
                {
                        for (int k = i+1; k < N; k++)
                        {
                                A[j][k] = A[j][k] - A[j][i]*A[i][k];
                        }
                }
        }
}
```

```
void bSub(double ** U, double * x, double * b)
{
        for(int i = N-1; i >=0 ; i--)
        {
                for(int j = i+1; j < N; j++)
                {
                        x[i] -= x[j]*U[i][j];
                }
                x[i] = (x[i] + b[i])/U[i][i];
        }

}

void fSub(double ** L, double * x, double * b)
{
        for(int i = 0; i < N; i++)
        {
                for (int j = 1; j < i; j++)
                {
                        b[i] -= L[i][j]*x[j];
                }
                x[i] = b[i]/L[i][i];
        }
}

void MVMult(double ** M, double * x, double * b)
{
        for(int i = 0; i < N; i++)
        {
                b[i] = 0; /* This could be done elsehwere*/
        }
        /*Could this be optimized?*/
        for(int i = 0; i < N; i++)
        {
                for(int j = 0; j < N; j++)
                {
                        b[i] += M[i][j]*x[j];
                }
        }
}
```

**Results**

```
(With O3 optimization, time in seconds)
My Solver: 0.653896
BLAS: 0.427258
Armadillo: 0.447107
```

```
Eigen: 0.285615
(Without O3 optimization)
My Solver: 2.659308
BLAS: 0.430189
Armadillo: 0.456686
Eigen: 9.694578
```

### 3.1.2 My LU solver

My solver is clunky, large and unoptimized. We should not expect great performance, and indeed we do not see great performance. Without optimization, however, this code outperforms Eigen! An interesting project would be to read the Eigen source code, because its performance with the g++ compiler is quite mysterious. It doesn't make any sense to me that a C++ library dedicated to linear algebra is beaten by a code I wrote in just a few minutes.

### 3.1.3 BLAS

In order to solve a linear system using the BLAS, one has to use two functions. The first one is dgetrf , which factors the matrix A, and the next one is dgetrs, which solves the system given a right hand side $\vec{b}$. In fact, dgetrs can solve for multiple right hand sides! dgetrf factors A using partial pivoting and overwrites LU onto A. It also outputs an array IPIV, which is the pivot matrix P. Here are the inputs for dgetrf:

1. M - The number of rows of A
2. N - The number of columns of A
3. A - An array of doubles. Note: This routine does not work for floats.
4. LDA - Leading dimension of A. This seems a bit redundant, as M is the leading dimension of A. This parameter may be necessary for some reason which is not yet clear to me.
5. INFO - The function stores a zero in INFO for successful completion, or an error code otherwise.

And here are the inputs for dgetrs:

1. TRANSX - The same as for dgemm
2. N - The order of the matrix A. A is a square matrix.
3. NHRS - Number of Right Hand Sides. This solver can solve multiple systems given multiple b's
4. A, b - The matrices in Ax = b. b does not have to be a column. b can store multiple right hand sides.
5. LDX - Leading dimension of matrix X.
6. IPIV - the pivot matrix for A.
7. INFO - The same as above.

### 3.1.4 Armadillo

The Armadillo library does not have an LU solver, but it does have a solver routine which uses an unspecified technique to solve the linear system. Therefore, I had to use a bit of trick to solve the system. I used the Armadillo lufactor routine, and then used the solve() routine to solve the corresponding triangular systems. Below you can see that the lu routine takes in three blank arrays to store the L U and P arrays, and the array A to be factored. Then on the next line you can see how I chained together two solve()s in order to solve the triangular systems. Armadillo does not overwrite A and in this way loses a bit of performance due to memory transfers, but even for a square matrix of order 1024, the effect of the extra matrices is negligible when one compares this routine to the performance of the pure BLAS routine.

```
lu(L_arma, U_arma, P_arma, A_arma);
x_arma = solve(trimatu(U_arma), solve(trimatl(L_arma), P_arma*b_arma));
```

Once again, Armadillo offers a clear, concise syntax and relatively fast code.

### 3.1.5 Eigen

I had some brief difficulty using the LU solver from Eigen. I had to dig around online forums in order to find out how to factorize the matrix and then again to figure out how to solve the system. It turns out that the two functions can be chained together, as is seen in the above code. Fortunately, in the end, this makes for very clean code. The next difficulty I came across was deciding what part of the Eigen namespace to use. The partial pivoting routine is called lu(), but in order to use this routine one needs the line **using Eigen::PartialPivLU**, which is somewhat unintuitive. I found the Armadillo documentation much easier to navigate, and had no issue when importing specific Armadillo functions.

As we saw with the Eigen matrix multiplier, the lu solver is very slow without optimization, and then inexplicably fast when compiler optimization is used. I would like to read the Eigen source code to see why this happens. It could be that Eigen uses some form of parallelism, or that it has some other trick.

## 3.2 Python

### 3.2.1 Code

**Here is the code we are considering**

```
""" Remember the Python zen: There should be only one obvious way to do
things. We are going to compare the low-level LAPACK functions
degtrf and dgetrs to the built in scipy LU routines. """

from numpy import array
from scipy.linalg import lu_solve, lu_factor
from scipy.linalg.lapack import dgetrf, dgetrs
from numpy.random import random
import time

# Make arrays
```

```
NUM_ITER = 10
N = 1024
A = random((N,N))
b = random((N,1))

# Solve using scipy.linalg
start = time.time()
for it in range(NUM_ITER):
        (LU_and_piv) = lu_factor(A)
        (x) = lu_solve(LU_and_piv, b)
stop = time.time()
print "Time for scipy routine is", (stop - start)/NUM_ITER

# Solve using scipy.linalg.lapack
start = time.time()
for it in range(NUM_ITER):
        (LU, piv, info) = dgetrf(A)
        (x) = dgetrs(LU, piv, b)
stop = time.time()
print "Time for LAPACK routine is", (stop - start)/NUM_ITER
```

**Results**

```
Time for scipy routine is 0.147462892532
Time for LAPACK routine is 0.144656896591
```

### 3.2.2 A brief discussion of the Results

True to the python philosophy, there is really only one way to solve a problem using LU factorization in Python. Much of scipy is built upon the BLAS and LAPACK, and it turns out that the scipy lu_solve() and lu_factor() routines are nothing more than front ends to the LAPACK functions dgetrf and dgetrs. As you can see above, the run time is the same for the LAPACK and the Scipy routines. Indeed, one can read the Scipy source code to confirm this suspicion. The entire Scipy source code is on GitHub if you are curious. Interestingly enough, the Python codes outperform the equivalent C++ codes again. This could be due to the g++ compiler, or it could be attributable to some other reason. The deserves more experimentation on more computers, and with other compilers. This experiment would be particularly interesting to perform with the ICC compiler.

# Chapter 4

# Conclusion

In the end we see that Armadillo and the BLAS offer incredible speed ups over the cache efficient code which one can write in C++. This is because for the routines we considered, Armadillo is simply a front end to the BLAS. Should we choose to not use compiler optimization (say, because we don't trust -O3), then the best bet is probably to use Armadillo. This library offers speed and a MATLAB like syntax which is very easy to understand. If, on the other hand, we would like to use a compiler optimizer, then (at least on my machine), Eigen has shown itself to be the clear winner, running in roughly one third of the time of the BLAS. Without optimization, however, Eigen has a great deal of overhead which slows it down considerably. In addition, Eigen has a rather confusing syntax, and is a bit more difficult to both install and compile with. According to many sources, Eigen is often used in market-quality software. But, for the common student, it is probably not worth the expense of learning how to use.

If we turn our attention to Python, there is no contest. The Python motto is "Everything is an object", and this is clear when you try to implement libraries which are not specifically tailored to Python in the language. We see that the numpy routines offer incredible speed ups over the pure BLAS functions. This is because, although at their lowest level, the Python routines do often implement the BLAS routines, they do so in a carefully optimized fashion. In a code which I have already shared on my github account I showed how Python matrix multiplication is not sped up, even through the cache-efficiency tricks which work in C. This is due to the fact that Python has a built in optimizer which is very smart about how to handle complicated Python objects. I would say that the most interesting result of this whole paper is that Numpy and Scipy can far outperform C++ codes.

# References

1. Armadillo Documentation and Download `http://arma.sourceforge.net/`

2. BLAS Documentation `http://www.netlib.org/blas/`

3. Eigen Documentation `http://eigen.tuxfamily.org/index.php?title=Main_Page`

4. LAPACK Documentation `http://phase.hpcc.jp/mirrors/netlib/lapack/`

5. A "Hands On" Introduction to OpenMP `http://openmp.org/mp-documents/omp-hands-on-SC08.pdf`

6. Parallel BLAS `http://www.netlib.org/scalapack/pblas_qref.html`

7. Scipy LAPACK Documentation `http://docs.scipy.org/doc/scipy-dev/reference/linalg.lapack.html`

8. Scipy.linalg Documentation `http://docs.scipy.org/doc/scipy-0.14.0/reference/linalg.html`

9. Relevant Stack Overflow Pages, `http://stackoverflow.com/questions/`...

   - `15451958/is-there-more-clear-way-to-do-matrix-of-random-numbers`
   - `10112135/understanding-lapack-calls-in-c-with-a-simple-example`
   - `17940721/armadillo-c-lu-decomposition`