Report on Lab 2 of SDTM
By

AMAKOR Augustina Chidinma and KARIMI Soufiane

October 19th

Contents

1 Create Project 1

2 Installation of Eigen 2

3 Eigen Library 2

4 Quick Test 2

5 Testing Performance of Eigen with Time 3

6 Optimization of codes using flags 5

7 Comparing Eigen and Boost.ublas by Time Compilation 6

8 Processors manipulation 8

1 Create Project

We started our project by creating a directory in the `HOME` Directory named `Lab2`, and we also created a `CMakelists.txt` as shown below:

```
(base)karimi@MacBook-Pro-de-Karimi% mkdir Lab2
(base)karimi@MacBook-Pro-de-Karimi% cd Lab2
(base)karimi@MacBook-Pro-de-Karimi% cat > CMakeLists.txt
```

In the `CMakelists.txt`, we included the below codes:

```
cmake_minimum_required(VERSION 2.8.11)
set(CMAKE_BUILD_TYPE Release)
project(Lab2)
```

The file at the root makes sure that the version of `cmake` is currently recent(2.8.11), defines the name of the project and indicates to compile in release mode. Then we compiled by `cmake .` then `make`

2 Installation of Eigen

We untared the `eigen-3.4.0.tar` and add it to our project directory `Lab2`. Then we included it into the `CMakelists.txt` as an external one as shown below:

```
include(ExternalProject)
ExternalProject_Add(
Eigen
SOURCE_DIR SOURCE_DIR ${Lab2_SOURCE_DIR}/eigen-3.4.0
INSTALL_COMMAND echo "Skipping install")
```

We compiled the above using `make` and the eigen library was installed successfully.

3 Eigen Library

We then add the eigen directory using the code below to the `CMakelists.txt`:

```
INCLUDE_DIRECTORIES(${Lab2_SOURCE_DIR}/eigen-3.4.0)
```

4 Quick Test

We created a subdirectory `tests` where we added some tests `test1.cpp` and `test2.cpp` to check Eigen Library. For this, we updated the main `CMakelists.txt` in the `Lab2` directory by adding;

```
add_subdirectory(tests)
```

Also, we create a `CMakelists.txt` in the subdirectory `tests` as shown below:

```
add_executable(test1 test1.cpp)
add_executable(test2 test2.cpp)
enable_testing()
add_test(NAME Test1 COMMAND test1)
add_test(NAME Test2 COMMAND test2)
```

Then we compiled our tests with `make` respectively as shown below:
`test1.cpp`

```
#include <iostream>
#include <Eigen/Dense>

using Eigen::MatrixXd;

int main()
{
  MatrixXd m(2,2);
  m(0,0) = 3;
  m(1,0) = 2.5;
  m(0,1) = -1;
  m(1,1) = m(1,0) + m(0,1);
  std::cout << m*m << std::endl;
}
```

Output:`./test1`

```
 6.5   -4.5
11.25 -0.25
```

`test2.cpp`

```
#include <iostream>
#include <Eigen/Dense>

using Eigen::MatrixXd;

int main()
{
  VectorXd v(2);
  v(0) = 4;
  v(1) = v(0) - 1;
  std::cout << v << std::endl;
}
```

Output:`./test2`

```
4
3
```

Our Eigen Library was successfully imported since we had no error.

5 Testing Performance of Eigen with Time

Here we created new tests of Matrix by Matrix multiplication of size(10,100,1000) and Matrix by vector multiplication of size(10,100,1000) to test the performance of the eigen. Also, we created a benchmark `header.cpp` that contains the `include directives` for easy inclusion into our tests as shown below;
`./tests/header.cpp`

```
#include <iostream>
#include <Eigen/Dense>
# include <boost/timer/timer.hpp>
using namespace Eigen ;
using namespace boost :: timer ;
```

`./tests/test_timer1.cpp`
Initialise a matrix A of size(n,n) where $n \in (10,100,1000)$, do the multiplication $A * A$, and return the time compilation for each size.

```
#include header.cpp
MatrixXd multiplication_matrix(int size){
  MatrixXd A(size,size);
  for (int row = 0; row <size; ++row){
   for (int col = 0; col <size; ++col)
    {
     A(row,col) = row+col;
    }
  }
  return A*A;
}
int main()
{
  cpu_timer timer;
  std::cout << "Here is A*A:\n" << multiplication_matrix(10)
  <<std::endl;
  cpu_times times10 = timer.elapsed();
  timer.start();
  std::cout << "Here is A*A:\n" << multiplication_matrix(100)
  << std::endl;
  cpu_times times100 = timer.elapsed();
  timer.start();
  std::cout << "Here is A*A:\n" << multiplication_matrix(1000)
  << std::endl;
  cpu_times times1000 = timer.elapsed();
  std::cout <<"wall time by size:10,100,1000 : "<<times10.wall<<","
  <<times100.wall<<","<<times1000.wall<<'\n';
  std::cout <<"system time by size:10,100,1000 : "<<times10.system
  <<","<<times100.system<<","<<times1000.system<<'\n';
  std::cout <<"user time by size:10,100,1000 : "<<times10.user
  <<","<<times100.user<<","<<times1000.user<<'\n';
}
```

`./tests/test_timer2.cpp`
Initialise a matrix A of size(n,n) and a vector V of size n where $n \in$ (10,100,1000), do the multiplication $A * v$, and return the time compilation for each size.

```
#include header.cpp
VectorXd multiplication_matrix_vector(int size){
  MatrixXd A(size,size);
  for (int row = 0; row <size; ++row){
   for (int col = 0; col <size; ++col)
    {
     A(row,col) = row+col;
    }
  }
  VectorXd v(size);
  for (int row = 0; row <size; ++row){
       v(row) = row;
  }
  return A*v;
}
int main()
{
  cpu_timer timer;
  std::cout << "Here is A*v:\n" << multiplication_matrix_vector(10)
  << std::endl;
  cpu_times times10 = timer.elapsed();
  timer.start();
  std::cout << "Here is A*v:\n" << multiplication_matrix_vector(100)
  << std::endl;
  cpu_times times100 = timer.elapsed();
  timer.start();
  std::cout << "Here is A*v:\n" << multiplication_matrix_vector(1000)
  << std::endl;\vspace{5mm}
  cpu_times times1000 = timer.elapsed();
  std::cout <<"wall time by size:10,100,1000 : "<<times10.wall<<","
  <<times100.wall<<","<<times1000.wall<<'\n';
  std::cout <<"system time by size:10,100,1000 : "<<times10.system
  <<","<<times100.system<<","<<times1000.system<<'\n';
  std::cout <<"user time by size:10,100,1000 : "<<times10.user
  <<","<<times100.user<<","<<times1000.user<<'\n';
}
```

Here, we tested the performance (time complexity) of all our eigen tests using (`Boost.Timer`).

To do this, we included all the `system and timer` components of the Boost for the individual tests into the `CMakelists.txt` of the `tests` directory as shown below:

```
set(Boost_USE_STATIC_LIBS ON)
find_package(Boost COMPONENTS system timer unit_test_framework REQUIRED)
add_executable(test_timer1 test_timer1.cpp)
add_executable(test_timer2 test_timer2.cpp)

target_link_libraries(test_timer1
        ${Boost_LIBRARIES}
        ${Boost_UNIT_TEST_FRAMEWORK_LIBRARY}
        ${Boost_TIMER_LIBRARY}
        ${Boost_SYSTEM_LIBRARY})
target_link_libraries(test_timer2
        ${Boost_LIBRARIES}
        ${Boost_UNIT_TEST_FRAMEWORK_LIBRARY}
        ${Boost_TIMER_LIBRARY}
        ${Boost_SYSTEM_LIBRARY})
enable_testing()
add_test(test_timer1 test_timer1)
add_test(test_timer2 test_timer2)
```

We compile then our tests by `make` and we get the outputs shown below:
`./test_timer1`

```
wall time by size:10,100,1000 : 308039,18446740,1058655227
system time by size:10,100,1000 : 0,0,50000000
user time by size:10,100,1000 : 0,10000000,920000000
```

`./test_timer2`

```
wall time by size:10,100,1000 : 184896,452254,11537544
system time by size:10,100,1000 : 0,0,0
user time by size:10,100,1000 : 0,0,10000000
```

We remark that the time compilation increases in size.

6 Optimization of codes using flags

We compared our previous time of code compilation by updating the `CMakelists.txt` in the main directory with

```
set(CMAKE_CXX_FLAGS "-g")
```

using test1 results with the following flags
-g

```
wall time by size:10,100,1000 : 491531,21054504,1033193165
system time by size:10,100,1000 : 0,0,50000000
user time by size:10,100,1000 : 0,10000000,910000000
```

-O2

```
wall time by size:10,100,1000 : 555540,19407981,1034552431
system time by size:10,100,1000 : 0,0,50000000
user time by size:10,100,1000 : 0,10000000,920000000
```

-O3

```
wall time by size:10,100,1000 : 201318,13564393,1033549776
system time by size:10,100,1000 : 0,0,50000000
user time by size:10,100,1000 : 0,10000000,910000000
```

-O2 -msse2

```
wall time by size:10,100,1000 : 194555,12009848,1026738239
system time by size:10,100,1000 : 0,0,50000000
user time by size:10,100,1000 : 0,10000000,900000000
```

-O3 -msse2

```
wall time by size:10,100,1000 : 213191,12832374,1030161815
system time by size:10,100,1000 : 0,0,50000000
user time by size:10,100,1000 : 0,10000000,900000000
```

Using the flags, we observed that `-O2 -msse2` and `-O3 -msse2` are equivalent, so is also `-O2` and `-O3`. `-g` performs better than the two former but the best to optimize for code size and execution time is `-O2 -msse2`.

7 Comparing Eigen and Boost.ublas by Time Compilation

Using this time `Boost.ublas`, we created new tests of Matrix by Matrix multiplication of size(10,100,1000) and Matrix by vector multiplication of size(10,100,1000) to test the performance of the ublas.
Also, we created a benchmark `header2.cpp` that contains the `include directives` for easy inclusion into our tests as shown below;
`./tests/header2.cpp`

```
#include <boost/timer/timer.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
using namespace boost::timer;
using namespace boost::numeric::ublas;
```

`./tests/test_ublas1.cpp`
Initialise a matrix A of size(n,n) where $n \in$ (10,100,1000), do the multiplication $A * A$, and return the time compilation for each size.

```
#include header2.cpp
matrix<double> multiplication_matrix(int size){
  matrix<double> A(size,size);
  for (int row = 0; row <size; ++row){
   for (int col = 0; col <size; ++col)
    {
     A(row,col) = row+col;
    }
  }
  return prod(A,A);
}
int main()
{
  cpu_timer timer;
  std::cout << "Here is A*A:\n" << multiplication_matrix(10)
  <<std::endl;
  cpu_times times10 = timer.elapsed();
  timer.start();
  std::cout << "Here is A*A:\n" << multiplication_matrix(100)
  << std::endl;
  cpu_times times100 = timer.elapsed();
  timer.start();
  std::cout << "Here is A*A:\n" << multiplication_matrix(1000)
  << std::endl;
  cpu_times times1000 = timer.elapsed();
  std::cout <<"wall time by size:10,100,1000 : "<<times10.wall<<","
  <<times100.wall<<","<<times1000.wall<<'\n';
  std::cout <<"system time by size:10,100,1000 : "<<times10.system
  <<","<<times100.system<<","<<times1000.system<<'\n';
  std::cout <<"user time by size:10,100,1000 : "<<times10.user
  <<","<<times100.user<<","<<times1000.user<<'\n';
}
```

`./tests/test_ublas2.cpp`
Initialise a matrix A of size(n,n) and a vector V of size n where $n \in (10,100,1000)$, do the multiplication $A * v$, and return the time compilation for each size.

```
#include header2.cpp
vector<double> multiplication_matrix_vector(int size){
  matrix<double> A(size,size);
  for (int row = 0; row <size; ++row){
   for (int col = 0; col <size; ++col)
    {
     A(row,col) = row+col;
    }
  }
  vector<double> v(size);
  for (int row = 0; row <size; ++row){
      v(row) = row;
  }
  return prod(A,v);
}
int main()
{
  cpu_timer timer;
  std::cout << "Here is A*v:\n" << multiplication_matrix_vector(10)
  << std::endl;
  cpu_times times10 = timer.elapsed();
  timer.start();
  std::cout << "Here is A*v:\n" << multiplication_matrix_vector(100)
  << std::endl;
  cpu_times times100 = timer.elapsed();
  timer.start();
  std::cout << "Here is A*v:\n" << multiplication_matrix_vector(1000)
  << std::endl;
  cpu_times times1000 = timer.elapsed();
  std::cout <<"wall time by size:10,100,1000 : "<<times10.wall<<","
  <<times100.wall<<","<<times1000.wall<<'\n';
  std::cout <<"system time by size:10,100,1000 : "<<times10.system
  <<","<<times100.system<<","<<times1000.system<<'\n';
  std::cout <<"user time by size:10,100,1000 : "<<times10.user
  <<","<<times100.user<<","<<times1000.user<<'\n';
}
```

Here, we tested the performance (time complexity) of all our ublas tests using (`Boost`). To do this, we included all the `system and timer and Boost Libraries` for the individual tests into the `CMakelists.txt` of the `tests` directory as shown below:

```
set(Boost_USE_STATIC_LIBS ON)
find_package(Boost COMPONENTS system timer unit_test_framework REQUIRED)
add_executable(test_timer1 test_timer1.cpp)
add_executable(test_timer2 test_timer2.cpp)
add_executable(test_ublas1 test_ublas1.cpp)
add_executable(test_ublas2 test_ublas2.cpp)
target_link_libraries(test_timer1
        ${Boost_LIBRARIES}
        ${Boost_UNIT_TEST_FRAMEWORK_LIBRARY}
        ${Boost_TIMER_LIBRARY}
        ${Boost_SYSTEM_LIBRARY})
target_link_libraries(test_timer2
        ${Boost_LIBRARIES}
        ${Boost_UNIT_TEST_FRAMEWORK_LIBRARY}
        ${Boost_TIMER_LIBRARY}
        ${Boost_SYSTEM_LIBRARY})
target_link_libraries(test_ublas1
        ${Boost_LIBRARIES}
        ${Boost_TIMER_LIBRARY}
        ${Boost_FILESYSTEM_LIBRARY}
        ${Boost_SYSTEM_LIBRARY}
        ${Boost_UNIT_TEST_FRAMEWORK_LIBRARY}
                       )
target_link_libraries(test_ublas2
        ${Boost_LIBRARIES}
        ${Boost_TIMER_LIBRARY}
        ${Boost_FILESYSTEM_LIBRARY}
        ${Boost_SYSTEM_LIBRARY}
        ${Boost_UNIT_TEST_FRAMEWORK_LIBRARY}
                       )
enable_testing()
add_test(test_timer1 test_timer1)
add_test(test_timer2 test_timer2)
add_test(test_ublas1 test_ublas1)
add_test(test_ublas2 test_ublas2)
```

We compiled our tests by `make` and we got the outputs shown below:
`./test_ublas1`

```
wall time by size:10,100,1000 : 199058,14802150,1897406463
system time by size:10,100,1000 : 0,0,60000000
user time by size:10,100,1000 : 0,0,1580000000
```

`./test_ublas2`

```
wall time by size:10,100,1000 : 176577,177500,11335523
system time by size:10,100,1000 : 0,0,0
user time by size:10,100,1000 : 0,0,0
```

we remark that for large size matrix(1000), the compilation using Eigen is faster, which is not true for small sizes(10,100).

8 Processors manipulation

We execute our tests using 1,2,3,4 processors with the commands below:

```
taskset 0X1 make
taskset 0X2 make
taskset 0X3 make
taskset 0X4 make
```

and we observe the following outputs for the test `test_timer1`: `Using one processor`

```
wall time by size:10,100,1000 : 217583,14074826,2193977452
system time by size:10,100,1000 : 0,0,30000000
user time by size:10,100,1000 : 0,10000000,1530000000
```

`Using two processors`

```
wall time by size:10,100,1000 : 150798,14689499,2131581800
system time by size:10,100,1000 : 0,0,40000000
user time by size:10,100,1000 : 0,10000000,1500000000
```

`Using three processors`

```
wall time by size:10,100,1000 : 677481,26196988,2122697280
system time by size:10,100,1000 : 0,0,70000000
user time by size:10,100,1000 : 0,10000000,1460000000
```

`Using four processors`

```
wall time by size:10,100,1000 : 391060,26761858,2124255717
system time by size:10,100,1000 : 0,0,90000000
user time by size:10,100,1000 : 0,10000000,1450000000
```

We remark that the time of compilation decreased when augmenting the number of working processors.