PROJECT PROPOSAL FOR

Google Summer of Code 2014

New Algorithms for Boost.numeric.uBLAS

Ganesh Prasad Sahoo
March 11, 2014

1 Abstract

This document is a short description of my vision and ideas for Google Summer of Code 2014 project 'New Algorithms for Boost.numeric.uBLAS'. This project intends implement various matrix decomposition algorithms and matrix solvers for the uBLAS library under Boost C++ libraries.

uBLAS is an elegant library of Boost for linear algebra. But it still needs a wide range of algorithms to be implemented. In uBLAS we have support for LU decomposition and triangular solvers, but we lack support for QR, Cholesky, SVD, Eigen, Hessenberg and Schur decompositions and respective solvers including Iterative linear solvers, eigen solvers and non linear solvers. We also lack alogorithms for matrix diagonalisation, simultaneous diagonalisation, inversion and determinant computation (Though determinant can be easily computed using LU factorization, it would be better to have an unary operation for that). Though it's possible to use Lapack++ algorithms over uBLAS data structures, it would be better to have these algorithms implemented within uBLAS. My project intends to implement these algorithms for Boost.numeric.uBLAS to make it more versatile and user friendly.

2 Personal Details

Name : Ganesh Prasad Sahoo
University : National Institute of Technology, Rourkela, India
Course : Computer Science and Engineering
Degree : Bachelor of Technology (Currently 4th Semester)
Email : sir.gnsp@gmail.com

Personal Contact : ManiSahi, Bhadrak, Odisha, 756181
Mobile : +91-9668588771

3 Availability

- I plan to spend 500 hours for my GSoC project working 50 hours a week on an average. I intend spending more time on testing, debugging and performance analysis, my codes must be clean, effective, optimized and perfectly readable.
- My project can be divided into 3 classes of algorithms : Decompositions, Solvers and Unary Operations. Hence, I intended to complete it in four phases, dedicating the fourth phase completely to testing, debugging, performance analysis, documentation and prerelease code revisions and improvements.
- Documentations are as important as the code itself, so my plans are to use 60% of my time in coding and 40% of my time in writing documentations. My documentations will include Example snippets, basic tutorials and references.
- I intend to start the actual development from 19th of May itself and finish coding by 10th of August. I am already acquainted with the uBLAS library and have thoroughly gone through the code and documentations, so I would prefer to start my project before 19th of

May, if possible and try to implement some more algorithms during the coding period if time permits.

- There are absolutely no factors that will affect my availability during the coding period, if we do not count upon any possibility of accidental illness.

4 Background Information

4.1 Goals and Motivations

I have been acquainted with the Boost.numeric.uBLAS library for last 18 months. I am also getting quite acquainted with other linear algebra libraries like eigen and Lapack++ lately. I have gone through the documentations and source of the uBLAS library quite thoroughly and I understand its design concepts. I have gone through the sources of eigen and Lapack++ too. Therefore I have a clear idea and understanding about what need to be implemented in uBLAS to make it a more versatile library in its field of linear algebra.

Moreover I feel enthusiastic about the FOSS philosophy and I use many open source softwares like Ubuntu, code::blocks etc. I respect the Boost C++ libraries - they are a great wealth of cool libraries covering a wide range of utilities. I understand that if I develop library for Boost, I must write very clean and efficient code. I think it would be a great experience and a valuable service to the open source world, so I want to contribute to Boost.

And my interests lie in the fields of numerical computation and algorithmic coding, so I am genuinely interested in the project I am proposing. I have undertaken several courses relevant to this project during previous semesters, so I feel quite confident and well equipped for this project.

4.2 Education

4.2.1 History

2004 – 2009 :	Passed High school certificate examination (Board of Secondary Education,Odisha) with 93.5% of marks
2009 – 2011 :	Passed Higher Secondary Examination (Council of Higher Secondary Education,Odisha) with 79% of marks
2012 – Present :	Student of B.Tech Computer Science and Engineering at National Institute of Technology, Rourkela, India

4.2.2 Relevant Courses Undertaken

- Mathematics – I (ODEs and Laplace Transforms) Autumn 2012
- Mathematics – II (Linear Algebra, Vector Calculus and Fourier Analysis) Spring 2013
- Data Structures and Algorithms : Spring 2013
- Mathematics – III (Numerical Methods) Autumn 2013
- Numerical Methods Laboratory Course : Autumn 2013
- Elementary Number Theory : Autumn 2013
- Discrete Mathematics : Autumn 2013
- Mathematics – IV (Complex Analysis) Spring 2014
- Principles of Programming Languages : Spring 2014

4.3 Experience

Dec, 2012 :	A small library to support functions like getch(), getche(),clrscr() etc for C/C++ under UNIX shell like environments
Feb, 2013 :	A library to support LISP like list processing techniques in C++

July, 2013 :	Study of Strategy Based Games and development of an Othello game based on improvised MinMax algorithm
Sept, 2013 :	An interpreter for a self-designed BrainF**k like language, coding was minimal but counts for studies into Computability theory
Oct, 2013 – Present :	Development of concepts of 'Two level Symbol Rewrite Systems' and the Programming Language 'I' and its interpreter.
Feb, 2014 :	Gns::Meta-Liscpp library, A complete compile time template metaprogramming library providing a LISP like syntax for template meta programs in C++. Available on git.

4.4 Skills

Languages :	C, C++, C++11, C#, Java, Pascal, Python 3, LISP (GCL), SWI-PROLOG, Haskell, Google Go. (Among others : SQL, PL/SQL and VHDL).
C/C++ IDEs :	Dev Cpp,Code::Blocks,Visual studio C++ Express
Compilers :	gcc, MinGW, Visual C++, DMC
Debuggers :	gdb
Profilers :	gprof
Documentation :	Docxygen
Version Control :	git

4.5 Ratings

C++ :	4.5	(I am well acquainted with features and techniques)
C++ STL :	4.5	(I use it almost every time I code in C++)
Boost :	3	(I am acquainted with a handful of libraries only)
Subversion :	1	(I am a git user, never exactly used subversion)
Git :	4	(well, I use it)

5 Project Proposal

5.1 Introduction

The goal of this project is to provide a set of unary operations, factorization algorithms and solvers to make the Boost.numeric.uBLAS library more versatile and user friendly. The list of the to be implemented algorithms is given below :

1 Unary Operators :
   (a) Determinant
   (b) Inverse (For Square Martices)
   (c) Moore-Penrose Pseudo Inverse (For non-Square Matrices) *
   (d) Diagonalization
2 Factorization Algorithms
   (a) QR factorization
   (b) Cholesky factorization
   (c) SVD factorization
   (d) Eigen factorization
   (e) Hessenberg factorization *
   (f) Schur factorization *
3 Solvers
   (a) Iterative Linear Solver (using Gauss-Seidel method)
   (b) Eigen Solver
   (c) QR Solver
   (d) Nonlinear Solver *
   (e) Constrained Conjugate Gradient *
   (f) Minimal Residual Solver *

In the following sections I'll briefly describe these operators and algorithms and my ideas for their implementations. Then I'll be dividing the development period in 3 phases and propose a timeline and set milestones for this project. (The algorithms marked with * are to be implemented if and only if the other implementations are complete ahead of timeline)

5.2 Factorization Algorithms

5.2.1 QR Factorization

I intend to implement the QR Factorization using Householder Transformations. Householder reflections can be used to calculate QR decompositions by reflecting first one column of a matrix onto a multiple of a standard basis vector, calculating the transformation matrix, multiplying it with the original matrix and then recursing down the (i, i) minors of that product.

Prototypes :

```
template<class M, class T>
    typename M::size_type lu_factorize (M &m, T &u);
template<class M, class T>
    typename M::size_type lu_factorize (M &m, M &o, T &u);
```

Where
M : class of matrices
T : class of upper triangular matrices
m : Matrix to be decomposed
o : Orthogonal Matrix
u : Upper Triangular Matrix

5.2.2 Cholesky Factorization

Cholesky Decomposition takes a Hermitian positive definite matrix and factorizes it to the product of a lower triangular matrix with real and positive diagonal entries and its conjugate transpose. I intend to use the Cholesky - Banachiewicz algorithm for this.

Prototypes :

```
template<class H, class T>
    typename H::size_type lu_factorize (H &m, T &a);
template<class H, class T>
    typename H::size_type lu_factorize (H &m, T &a, T &b);
```

Where H : class of Hermitian matrices
T : class of upper triangular matrices
m : Matrix to be decomposed
a : Lower Triangular Matrix
b : conjugate transpose of a

5.2.3 SVD Factorization

The SVD of a matrix M is typically computed by a two-step procedure. In the first step, the matrix is reduced to a bidiagonal matrix. This takes $O(mn^2)$ floating-point operations (flops), assuming that $m \geq n$. The second step is to compute the SVD of the bidiagonal matrix. This step can only be done with an iterative method. However, in practice it suffices to compute the SVD up to a certain precision, like the machine epsilon. If this precision is considered constant, then the second step takes O(n) iterations, each costing O(n) flops.

My plans are to implement the 1st step using householder transformations, this will take comparatively less time to code, as I would have implemented householder transformations during the development of QR decomposition. The second step is similar to that of eigenvalue decompositions. So implementing this will also reduce the implementation time for Eigen decompositions. *(links to some algorithms are given in the reference)*

Prototypes :

```
template<class M, class D>
    typename M::size_type SVD_factorize (M &m, D &s, M &v);
template<class M, class D>
    typename M::size_type SVD_factorize (M &m, M &u, D &s, M &v);
```

Where M : class of matrices
D : class of diagonal matrices
m : Matrix to be decomposed
u, v : Unitary matrices
s : Diagonal matrix

5.2.4 Eigen Decomposition

Eigen Decomposition can be done in two steps, first step is to calculate the eigenvalues and second step is to determine the corresponding eigen vectors. I plan to implement the first step using power iteration method and the second step using the iterative linear solver based on Gauss-Seidel method. But for this decomposition to be implemented, the unary operator inverse must be implemented beforehand. Because that'll make the testing procedure easy.

Prototypes :

```
template<class M, class D>
    typename M::size_type SVD_factorize (M &m, D &l);
template<class M, class D>
    typename M::size_type SVD_factorize (M &m, M &q, D &l);
```

Where M : class of matrices
D : class of diagonal matrices
m : Matrix to be decomposed (Square matrix)
q : Square matrix with eigen vectors as columns
s : Diagonal matrix with eigenvalues as diagonal enties

5.2.5 Hessenberg Factorization *

Hessenberg Factorization of a nxn real square matrix can be implemented using n-2 Householder transformations. For complex matrices and general cases, the QR algorithm is used.
(To be implemented if all other algorithms are done ahead of proposed timeline)

5.2.6 Schur Factorization *

Hessenberg factorization is the first step of Schur factorization. It can be done with the QR algorithm with multiple shifts.
(To be implemented if all other algorithms are done ahead of proposed timeline)

5.3 Solvers

5.3.1 Iterative linear solver

I intend to implement the iterative linear solver using Gauss-Seidel method. The future plans are to implement the Jacobi method. As LU factorization and triangular solver are already present in uBLAS, this algorithm would take very less amount of time to implement.

5.3.2 Eigen Solver

I plan to implement the eigen solver after completing the iterative linear solver and the eigen decomposition due to the dependencies of the algorithms. Again, this solver would take minimal amount of time to implement, once the iterative linear solver and eigen decomposition are implemented.

5.3.3 QR Solver

As the name suggests, this solver is dependent upon the QR algorithm. So, it is to be implemented after the QR algorithm. Provided the dependencies, this algorithm would take minimal amount of time to be implemented.

The non linear solver, constrained conjugate gradient and the minimal residual solver are to be implemented if all other algorithms are implemented completely before the GSoC pencil down date.

5.4 Unary Operators

5.4.1 Determinant

I plan to implement an iterative method of finding determinant with $O(n^2)$ time complexity for nxn square matrices with no extra memory usage. It is to be implemented after the factorizations and solvers are implemented. If

possible, I would like to write an LU based determinant function too, to compare the complexities of both implementations.

5.4.2 Inverse

Inverse operator can be easily implemented using an improvised iterartive linear solver. This operator is to be optimized for every type of storage used for every class of matrices. As inverse is an widely used term in linear algebra, it would be better to have it optimized as much as possible. Future plans are to implement the Moore-Penrose pseudo inverse for non square matrices.

5.4.3 Diagonalisation

This is the easiest thing to implement after eigen decomposition and eigen solver.

6 Timeline and Milestones

The project is divided into 3 phases : #1. Base Algorithms, #2. Dependent Algorithms, #3. Operators and Intensive Testing. The detailed timeline is given below.

Present – May 18 :	Getting closer with the boost community, exploring existing libraries, discussing with the mentor about implementations, figuring out optimization strategies for each class of algorithms and preparing a detailed design.
Phase #1	
May 18 – June 15	Implementation of Base Algorithms : - Householder Transformation (7 days) - Iterative Linear Solver (5 days) - QR Decomposition Algorithm (7 days) - Cholesky Decomposition (7 days)

Phase #2

June 16 – July 16	Implementation of dependent algorithms: - Eigen Decomposition (7 days) - Eigen Solver (4 days) - Diagonalisation (4 days) - QR Solver (7 days) - SVD decomposition (8 days)

Phase #3

July 17 – August 8	Implementation of Operators and intensive testing - Determinant (7 days) - Inverse (7 days) - Intensive testing and performance analysis (5 days) - Code and Documentation revision and improvements (3 days)
August 9 – August 11	Preparation of final report.
Future Plans	Implementation of the algorithms marked by * in the list.

I'll be using 60% of the time in coding and 40% in writing documentation, tesing and performance analysis. I'll be submitting my codes weekly for reviews and advice.

7 References

QR Decomposition : http://en.wikipedia.org/wiki/QR˙algorithm#The˙practical˙QR˙algorithm
Cholesky Decomposition : http://en.wikipedia.org/wiki/Cholesky˙decomposition
SVD factorization : http://www.cs.utexas.edu/users/inderjit/public˙papers/HLA˙SVD.pdf
Eigen factorization : http://en.wikipedia.org/wiki/Eigendecomposition˙of˙a˙matrix
Power Iteration Method : http://en.wikipedia.org/wiki/Power˙method
Gauss-Seidel iteration method : http://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel˙method