École polytechnique fédérale de Lausanne

# Project Report PCSC - Group 12

*Authors:*
Wenyuan Lv, Mariella Kast

December 14, 2017

# 1 Introduction

This is a short guide to our coding project, which had the topic "*Linear Systems*". We will explain how to include and use the library that we have built in the course of this project, as well as point out some limitations and useful hints.
The type of problem this libary solves is

$$Ax = b, \quad b \in \mathbb{R}^n, A \in \mathbb{R}^{n \times n},$$

where we would like to now the solution vector $x$. This library will focus exclusively on quadratic systems, which allows for the use of both direct and iterative solvers. We also provide implementations for representing vectors and matrices in our library as well as do standard computations with them.

# 2 How to compile the library and get started.

After cloning the repository from github with the appropriate command[1], you can see the main structure of this project. There are three parts: the actual *Linear Systems* library which is located in "src", a "demo" folder which shows sample use of the features and the "test" suite. In order for the compilation to run smoothly, one needs an internet connection as the current version of the googletest library is included automatically. The *Linear Systems* library code itself is completely independent of googletest and does not use any of its functionality. Since the compilation is done via cMake files no other adjustements should be needed.
In order to see how to use the library and include it, simply see the demo folder, where examples are given. In general, you will simply need to include the header files of the classes you want to use, the linking is done by Cmake.

# 3 Implemented features

## 3.1 Basic classes

We implemented two basic classes (`Matrix` and `Vector`) that allow for general linear algebra computations. They allow the user to create and manipulate real-valued matrices and vectors. The "standard" algebraic operations are supported, so that the user cannot only create a Matrix and a Vector to solve a linear system, but also do some pre - and post computations of interest. The classes support:

- 0-based indexing with read-only (`vec.at(index)`) or write access `vec(index)`, similarly `mat(row,col)` for matrices.
- Multiplication with another Vector, Matrix or scalar with the `*` operator
- Addition/substraction of another matrix of equal dimensions or a scalar with the `+/-` operators

[1] git clone `https://github.com/jessll/PCSC2017_Group12.git`

- Matrix (and Vector) transpose with `mat.transpose()`
- Vectors additionally support a dot product with another Vector: `dotProduct(vec1,vec2)` (not orienation sensitive), and the computation of its p-norm, in particular the 2-norm by `vec.CalculateNorm()`
- A conversion of a 1D Matrix into a vector with `asVector(mat)`.

## 3.2 The solvers

We provide an assembly of linear systems solvers which inherit from the abstract class `LinSysSolver`. They all allows for reading and writing matrices to file, as well as solving the standard system $Ax = b$, with the command `solver.Solve(A,b)`. To date, this library can only solve quadratic systems, but we implemented all tools with regards to extensibility to general linear system solvers. In the following, we shortly discuss the required parameters and usage of each solver.

### 3.2.1 Iterative solvers

All iterative solvers have the two following parameters: The tolerance of error in the residual `tol` and the maximum number of iterations `maxIter`, which define the accuracy of the solution. If `maxIter` is reached before convergence, the solver will return the current result and print a warning.

**Conjugate Gradient**

The `CGSolver` can only be used for symmetric positive definite (spd) matrices. It does an initial check on symmetry, but might not achieve convergence for non positive matrices. It is possible to provide a (spd) preconditioner in order to speed up convergence. Alternatively, the solver can also create a simple diagonal preconditioner for the user (Jacobi preconditioning).

**Richardson Iteration**

The `RichSolver` can in principle be applied to any quadratic linear system. However, convergence for an arbitrary initial guess is only guaranteed if all eigenvalues of the matrix $A$ are positive. For this solver it is crucial to choose an appropriate parameter $w$ for the iterative step

$$x^{k+1} = x^k + w(b - Ax).$$

In order to guarantee convergence, $w$ should be chosen positive such that $w < \frac{2}{\lambda_{\max}}$, i.e. it depends on the largest eigenvalue of the matrix $A$.

**Jacobi and Gauss Seidel method**

The `JacSolver` and `GSSolver` are both very similar. They can be used to solve quadratic systems of diagonally dominant matrices. Entries on the diagonal can thus not be zero and the solver will throw an error if the input is not sufficiently diagonally dominant.

The main difference of the two methods conists in how the solution vector $x$ is updated: The Jacobi solver calculates every entry of $x^{k+1}$ based only on the previous guess $x^k$, whereas the Gauss-Seidel method uses the values of $x^{k+1}$ where they have been computed already.

#### 3.2.2 Direct solvers

Direct solvers try to decompose the matrix $A$ as two triangular matrices. After that, it will be easy to solve the linear system by a forward substitution method.

**LU Decomposition**

Let Matrix $A$ be a square and non-singular matrix. An LU factorization refers to the factorization of $A$ into two factors, a lower triangular matrix $L$ and an upper triangular matrix $U$: $A = LU$.

The main idea of LU decomposition is Gaussian elimination. So we need a trick since a zero on the diagonal will cause error during the computation. So we try LU factorization with Partial Pivoting, which means we exchange the largest value in the column with the diagonal in each iteration. This implementation helps to deal with more general cases.

**Cholesky Decomposition**

Cholesky Decomposition method is used for symmetric positive definite matrices, which we decompose into two triangular matrices: $A = LL^T$. If the input matrix $A$ is not symmetric positive definite, the code cannot decompose the matrix and will throw an exception.

# 4 Usage and standard program flow

The idea is to include the project as a library header file. The user can create and manipulate matrices and vectors as they need them. Also, the system supports input from keyboard to generate matrices. In order to read/write matrices from/to file or from the console, one needs to generate a solver object. Any solver is initialized with reasonable parameters if none are specified in the constructor. Before solving the system, all parameters of the solver like maximum number of iterations/ tolerance can be changed. The solving call then simply looks like x =solver.solve(A,b). For the iterative method, in case the solution does not converge, the solvers still return their current result, but provide the user with a warning. The results, can then be written to file or used further in the program as desired.

In summary, there are only three simple steps:

1. Create solver of desired type
2. Create Matrix and vector we want to solve for:
   - Assign values in matrix and vector after initialization.
   - Read from keyboard.
   - Read from files.
   - Do further linear algebra manipulation as desired.
3. Solve and maybe save results to file.

## 4.1 Remarks and tips

- Make sure type of matrix is suitable for the solver.

- We assume that solve can only be applied to a combination of Matrix, Vector. If you have a 1D matrix `m`, that should behave as a vector, you can convert it with `asVector(m)`, which will generate a vector from your matrix.
- For user-friendliness, the dotProduct ignores the orientation of the vectors. If you want an "orientation sensitive" dotproduct, you can do $< a, b >= a^T b$ by providing the correct dimensions of $a$ and $b$ and then using matrix multiplication, see the demo for an example.
- The file format for writing to file is the following: The first two numbers in the file specify the dimensions of the matrix, all other values are values of the matrix.

# 5 The test suite

We use googletest for testing our code. The Readme explains how to run the tests. The basic classes Vector and Matrix are simply tested for functionality.

## 5.1 Testing the solvers

In order to test the solving capabilities of the different implemented methods, it is necessary to generate larger linear systems. In order to control the properties of the randomly created matrices, we provide a short MATLAB file that generates spd matrices, a random right-hand side and calculates a reliable reference solution with MATLAB's "Backslash" operator. The results are then written to file and can be used in the test suite.
We tested each solver on a small $10 \times 10$ and a large $100 \times 100$ system and checked for convergence and accuracy by comparing to the MATLAB solutions.

## 5.2 Testing the exceptions

Our library uses a very simple exception class based on the text book. In order to test the corner cases and invalid operations for the methods, it is also necessary to check whether exceptions are thrown appropriately. So we also test the "exceptionial" behaviour for each method.

# 6 Perspectives

We wrote this library in order to allow for easy extensions, for example to solving systems with non-quadratic matrices. Another user-friendly extension, would be to choose a more suitable solver automatically, i.e. instead of throwing an exception when the solver encounters a problem, one could give a warning and use a default solver that works quite robustly. Since non of our solvers are uniformly applicable, this would require the implementation of a more advanced solver like MINRES that can be applied to any kind of quadratic linear systems.