C++ library for matrix operations

Candidate Number: 1003876

June 29, 2016

Contents

Code listings

Introduction

A great surge of computational power available at a low price allows scientists to make simulations on scales that were unthinkable a few years ago. Nowadays, every graduate student can run simulations of models on their personal computers that could exist only as concepts not so long ago. This encouraged a shift in most of modern scientific computing from low-level languages like C, C++ in which users need to be aware of their hardware resources, to higher-level languages like MATLAB, Python or R that enable us to write code in much more abstract terms and allows us to focus on our model instead of on the technical side of things. However, there are many computationally demanding areas in science which require writing of well-optimized code and efficient management of computer memory.

The main goal of this special topic is to bring together advantages of both worlds. On the one hand, we want to write code that is fast and this will require us to be aware of how memory is managed. On the other hand we want to write code that is concise and easy to write and read. We will attempt to get our code as close to MATLAB syntax as possible. While full implementation of our code is accessible via the anonymized Github repository[1] we also provide a number of code listings in the main text while describing our library.

[1] `https://github.com/can1003876/cpp`

1 Basic functionality

1.1 Showcase

Our approach is to create one general matrix class which can also represent vectors. This facilitates easier code writing because we do not have to overload our functions for two different classes. Moreover, representing matrices as vectors does not require any significant additional price.

One of the greatest features of C++ is its ability to create templates that allow functions and classes to operate with generic types. In practice, we might want to use our matrix class for many different problems. When solving linear algebra problems we are able to select different precision for the floating point arithmetic. We can choose *float* to save memory or *double* to achieve greater accuracy. Alternatively, we might want to work with networks and use only *boolean* type to save their adjacency matrices. Moreover, if we are interested in using matrices as data tables, we can choose *integers* or even non-numeric type like *strings*.

Of course, most of the functionality developed is only supported for numeric types. Specifically, the default type used here is *double* and all the linear solvers that are implemented to return the solution of this type. We also create functions for *type casting* that are used to convert between *integer* and *double* matrices. This is demonstrated in Code listing 1.

Code listing 1: Conversion between different matrix types

```
// Code listing 1 (lines 26 to 51):
// 2x3 double matrix "A":
TMatrix<double> A(RandomGauss<double>(2, 3, 1.0, 3.0), "A");
A(2,2) = 1; // Change A_{2,2} == 1:

// Create integer matrix B by rounding elements of A:
TMatrix<int> B(transpose(A).toInt(), "B");
cout << A;
//A =
//(0.115494, 4.04615, 7.23705)
//(-0.752548, 1, 0.569711)

cout << B;
//B =
```

```
15 //(0, -1)
   //(4, 1)
   //(7, 1)

   // Transpose B, cast it as double and add it to A:
20 cout << A+transpose(B).toDouble();
   //A+B^T-double =
   //(0.115494, 8.04615, 14.237)
   //(-1.75255, 2, 1.56971)

25 cout << A*B.toDouble();          //name : "A*B-double"
   cout << A.toInt()*B;             //name : "A-int*B"
```

Our matrix class has four different constructors that allow for very flexible usage. To initialize matrix A in the Code listing 1 above, we used one of the overloaded copy constructors in combination with a random gaussian matrix generator. The random gaussian matrix generator takes two integers describing the wanted size of the matrix, mean and standard deviation floats for the pseudo-random number generator. Finally the overloaded copy constructor enables us to specify the name of the matrix.

The matrix class enables us to specify name, but in case we do not choose any name, it generates a default name based on the operations performed on the matrix. In this way, we can see how the matrix was derived. This functionality can be seen on line 20, where we transposed and casted B to *double* type and added it to matrix A. The new matrix is only temporarily saved in memory, but it can be seen that its name is: *'A+B^T-double'*. In addition to this, using other overloaded operators like $+, -$, or $*$ has a similar effect.

1.2 Structure of the matrix class and accessing data

As can be seen in Code listing 2, each matrix object of type **T** has only four private variables which are accessible in various ways only through the public helping functions. The only significant memory requirement comes from storing the data for the matrix in $mData$ variable. This is an array that gets dynamically allocated when the matrix is created.

We also save two integers corresponding to the matrix size in $mShape$ and a number of all matrix elements in the $mSize$ variable. Every matrix is assigned a name that is saved in a string called $mName$, which is used mostly for printing

matrices to a terminal window.

Code listing 2: TMatrix data structure

```
private:
        T *mData;        // Matrix data (instead of: T **mData)
        int mShape[2];   // Dimensions of the matrix
        int mSize;       // Number of elements in the matrix
        string mName;    // Matrix name
```

It is important to note that we chose to store the matrix data in an array-of-pointers instead of pointer-to-pointers data structure, because the first type of implementation allows for a more efficient memory access. Even though writing optimized *cache-friendly* code is beyond the scope of this project, forcing all of the matrix data to be saved in spatial proximity to each other allows the code to load all the matrix data in one go. This is especially important in the case of a matrix, because we often iterate through all the data. Using two dynamic allocations could introduce alignment holes and spread our data around in the memory which would cause a slower memory access to its elements.

For accessing these variables we use class functions *getData(int i, int j), setData(int i, int j, T input), getName(), setName(string name), getShape(int 0 or 1).* Moreover, to access the matrix elements we can use MATLAB-like notation *A(int i, int j)*, which returns dereferenced pointer to the elements and therefore can be used to set or get the matrix value. More specifically *getData* functions are used in cases when we only have permission to read from the matrix, for example for the functions that take a constant reference to a certain matrix and passing the dereferenced pointer is not an option.

MATLAB-like notation can also be used to return the vector of certain row or column of the matrix. If only one positive integer is specified, *operator()* returns a new matrix that corresponds to the matrix column, and if negative integer is provided it returns the row vector. This can be seen in Code listing 3.

Code listing 3: Accessing matrix elements, columns and rows.

```
// Code listing 2 (lines 53 to 62):
```

```
cout << A;
//A =
//(0.115494, 4.04615, 7.23705)
//(-0.752548, 1, 0.569711)

cout << A(-1);
//A-1row = (0.115494, 4.04615, 7.23705)
cout << transpose(B(1).toInt());          // B = A^T.toInt()
//B-1column^T = (0, 4, 7)
```

One of the issues encountered during the implementation of our matrix class was how to print the matrices to a terminal window in a convenient way. There are two requirements we impose on the printing function. First, we want to prevent any unnecessary copying of the matrix inside the function as this should not be needed. Secondly, we want to be able to print the newly constructed matrices that are not permanently saved in the memory. The issue arouse when a templated class was introduced. However, the issue was solved using a helpful answer at StackOverflow[2].

1.3 Mathematical operations

The elementary mathematical operators $(+, -, *)$ are overloaded for the matrices. All of these binary operators take constant reference to the other matrix of the same data type and return a new matrix that corresponds to the relevant matrix operation. Additionally, we also have assignment $=$ operator that assigns data from one matrix to other, unary $-$ operator and overloaded $*$ for multiplication by scalar from both matrix sides.

Creation of a new matrix results into extra copying of $mSize$ elements every time we perform a matrix operation. However, we often create a temporary matrix, just to immediately reassign its elements to some other matrix. This includes cases when we want to assign $A = A + B$. To deal with this issue we implement a special matrix operation function that is used in this kind of update. In Code listing 4 it can be seen that this results in more than a triple performance increase. This becomes more significant for larger matrices because more data has to be copied.

[2] http://stackoverflow.com/questions/5284473/c-template-ostream-operator-question

Code listing 4: Cost of creating a temporary matrix object.

```
// Code listing 3 (lines 64 to 72):
TMatrix<double> C(RandomGauss<double>(5e3, 5e3, 0.0, 3.0), "C")
    ;
TMatrix<double> D(RandomGauss<double>(5e3, 5e3, 0.0, 3.0), "D")
    ;
clock_t start = clock ();
C = C+D;          // It took: 0.317092 sec.
printf("It took: %Lf sec.\n", (long double) (clock() - start)/
    CLOCKS_PER_SEC );
start = clock ();
C.rMAdd(D);       //It took: 0.085300 sec.
printf("It took: %Lf sec.\n", (long double) (clock() - start)/
    CLOCKS_PER_SEC );
```

2 Additional functions

We also create a number of functions that are very convenient for matrix manipulation and allow us to export data outside of our code. We have *diag* which works in the same way as its MATLAB counterpart. It either takes $n \times n$ matrix, $n \times 1$ or $1 \times n$ vector. If the matrix is inserted it returns a vector that corresponds to its diagonal. In case a vector is inserted it returns an $n \times n$ matrix full of zeros except for the diagonal which matches the inserted vector. If we want to keep only the diagonal of the matrix, we only need to apply *diag* twice: $diag(diag(A))$.

Code listing 5: Extracting and inputing diagonals and export to a CSV file.

```
// Code listing 4 (lines 74 to 89):
TMatrix<int> E(2, 2, 1, "E");
cout << E;
//E =
//(1, 1)
//(1, 1)
cout << transpose(diag(E));
```

```
//diag(E)^T = (1, 1)
cout << diag( diag(E) );
//diag(diag(E)) =
//(1, 0)
//(0, 1)
saveMat(diag( diag(E) + transpose(B(-1)) ), "diag.csv");
// diag.csv can be loaded in MATLAB with: csvread('diag.csv');
cout << sum(E) + norm(diag(E)) - sqrt(2); cout << "\n";
// 4
```

Moreover, we also have the *saveMat* function that takes any matrix as an input and saves it in a *CSV* file that can be easily imported in MATLAB using: $ans = csvread(filename)$. The user is able to specify the name of the output file, but if no name is specified, the matrix will be saved as: $A.getName() + `.csv'$.

It is worth emphasizing that this is not the best way of saving matrices. Ideally we would want to save it in the binary format, which would prevent any loss caused by a string representation of numbers that happens while printing to the output file. However, in our case we are not interested in such great precision and this *CSV* exporter works well for later exported times taken by various linear solvers.

3 Linear Solvers

In this section we are concerned with different algorithms for solving the systems of linear equations: $A\vec{x} = \vec{b}$, where $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n$ and A is of full rank. There are many different solvers and each of them is adequate for a different matrix setups. Our matrix class comes with six different solvers **(i)** LU factorization with backsolve, **(ii)** QR decomposition with backsolve, **(iii)** Jacobi method, **(iv)** Gauss–Seidel method, **(v)** Steepest descent method, **(vi)** Conjugate gradient method and a general *linsolve* solver that attempts to pick the best possible solver for the problem based on the properties of the underlying matrix. Here we ofter a brief overview and numerical comparison of the mentioned methods. A full theoretical background as well as geometrical intuition behind these methods can be found in [2].

3.1 LU and QR decompositions with backsolve

We implement the classical Gaussian elimination which is appropriate especially for small matrix sizes. Solving the system consists of two steps. First, we apply a

sequence of linear transformations to A and b to get A into an upper triangular format. This is referred to as 'triangular triangularization', because these operations are equivalent to multiplying the linear system from left by lower triangular matrix L^{-1}. As a result, we get A to the upper triangular form U.

$$\underbrace{L^{-1}A}_{U}x = L^{-1}b$$

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & \times \end{bmatrix} \cdot \vec{x} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \end{bmatrix}$$

Following this we can solve the system using backwards substitution:

$$\begin{aligned} x_n &= \frac{b_n}{U_{n,n}} \\ x_{n-1} &= \frac{b_{n-1} - U_{n-1,n}x_n}{U_{n-1,n-1}} \\ &\vdots \\ x_1 &= \frac{b_1 - \sum_{i=2}^{n} U_{i-1,i}x_i}{U_{1,1}}. \end{aligned}$$

Similarly, we can use the Gram-Schmidt QR decomposition to get a similar system that can be backsolved in the same way:

$$\underbrace{Q^T A}_{R}x = Q^T b.$$

Again, the Gram-Schmidt decomposition applies a sequence of triangular matrices to the left and therefore it corresponds to triangular orthogonalization. The floating point operations required by these algorithms grow cubically with size of the problem. For LU we need roughly $\mathcal{O}(\frac{2}{3}n^3)$ and the Gram-Schmidt QR requires $\mathcal{O}(\frac{4}{3}n^3)$ floating point operations. Although this makes the use of these methods very limited, it is especially useful for small systems as it finds the solution after a fixed number of steps.

3.2 Iterative solvers

On the other hand, we have iterative solvers that can be used for large systems provided that we have some knowledge about the structure of the matrix. These

solvers usually do not have a fixed number of steps (except for a special case of the Conjugate gradient method) and terminate after certain termination criteria are met. In our functions, the user can specify the required precision and the maximum of iterations after which the algorithm stops.

3.2.1 Jacobi and Gauss-Seidel algorithms

We can take the original system of linear equations and rearrange it in the following way:

$$\begin{aligned} Ax &= b \\ (L+U+D)\,x &= b \\ Dx &= b - (L+U)\,x \\ x &= D^{-1}b - \underbrace{D^{-1}\,(L+U)}_{T}\,x, \\ x_{n+1} &= D^{-1}b - Tx_n, \end{aligned}$$

where L, U and D are the lower-triangular, upper-triangular and diagonal parts of the matrix A. Since matrix D is diagonal, inverting it costs only n floating operations.

Now, under the assumption that the spectral radius $\rho(T) = \max|\lambda| < 1$, we can treat this as a contractive linear system with only one stable equilibria that corresponds to the system solution. In general, if a matrix is strictly column or row diagonally dominant, then $\rho(T) < 1$.

$$\sum_{j\neq i} |A_{i,j}| < |A_{i,i}|, \quad \forall i \in [1,\ldots,n] \qquad \textit{(strictly row diag. dom.)}$$

$$\sum_{j\neq i} |A_{j,i}| < |A_{i,i}|, \quad \forall i \in [1,\ldots,n] \qquad \textit{(strictly column diag. dom.)}$$

There is also the other way we can look at the Jacobi algorithm. It is often described as simultaneous displacements method. We have a system, from which we can express x_i:

$$\begin{aligned} \sum_{j=1}^{n} A_{i,j}\,x_j &= b_i && \forall i \in [1,\ldots n] \\ x_i &= \frac{b_i - \sum_{j\neq i} A_{i,j}\,x_j}{A_{i,i}} && \forall i \in [1,\ldots n] \\ x_i^{n+1} &= \frac{b_i - \sum_{j\neq i} A_{i,j}\,x_j^n}{A_{i,i}} && \forall i \in [1,\ldots n] \end{aligned}$$

If the right hand side is contractive mapping, we can iterate until we converge to the solution. Notice also that this algorithm can be easily implemented in parallel.

The Gauss Seidel method has just one subtle difference and that is that it uses the information from the previously computed x_i^{n+1} for the computation of x_j^{n+1}, $j > i$ in the following way:

$$x_i^{n+1} = \frac{b_i - \sum_{j=1}^{i-1} A_{i,j}\, x_j^{n+1} - \sum_{j=i+1}^{n} A_{i,j}\, x_j^n}{A_{i,i}} \qquad \forall i \in [1, \dots n]$$

Even though this has a faster convergence, we lost the ability to implement it in parallel. This corresponds to the successive displacements method.

With a recent gain of popularity of randomized numerical linear algebra there is also a new recently-proposed method that attempts to bring together the parallel implementation of the Jacobi algorithm with a faster convergence of the Gauss Seidel algorithm [1]. This is achieved by approximately expecting what the value for the new x_i^{n+1} is going to be for the parallel computation of the next x_{i+l}^{n+1}

3.2.2 Steepest descent and Conjugate gradient descent

Take a look at the following optimization problem:

$$\min f(x) = \frac{1}{2} x^T A x - bx, \qquad x \in \mathbb{R}^n,$$

If we assume that A is a positive definite matrix (i.e. $\forall x \neq \vec{0} : x^T A x > 0$ and $A^T = A$), this becomes a convex optimization problem with a unique global optima with zero gradient:

$$\nabla f(x) = Ax - b = \vec{0}$$

One could assume that quadratic optimization is harder to solve than the original problem. However, we now can apply the iterative line search methods for non-linear optimization that are able to solve our problem much more quickly. We implement the Steepest Descent and Conjugate Gradient, whose pseudocode can be seen in Algorithm 1 and 2.

Data: $A \succ 0 \in \mathbb{R}^{n\times n}, x_0, b \in \mathbb{R}^n$
$r_0 = b - Ax_0$
while *not termination criteria* **do**
  $\alpha_k = \frac{r_k^T r_k}{r_k^T A r_k}$
  $x_{k+1} = x_k + \alpha_k r_k$
  $r_{k+1} = b - Ax_{k+1}$
end

Algorithm 1: Steepest descent method.

Data: $A \succ 0 \in \mathbb{R}^{n\times n}, x_0, b \in \mathbb{R}^n$
$p_0 = r_0 = b - Ax_0$
$\alpha_0 = \frac{r_0^T r_0}{r_0^T A r_0}$
$x_1 = x_0 + \alpha_0 r_0$
$r_1 = b - Ax_1$
while *not termination criteria* **do**
  $\beta_{k-1} = \frac{r_k^T A p_{k-1}}{p_{k-1}^T A p_{k-1}}$
  $p_k = r_k - \beta_{k-1} p_{k-1}$
  $\alpha_k = \frac{r_k^T p_k}{p_k^T A p_k}$
  $x_{k+1} = x_k + \alpha_k r_k$
  $r_{k+1} = b - Ax_{k+1}$
end

Algorithm 2: Conjugate gradient.

The convergence of the steepest descent depends on the condition number of a system $\kappa = \frac{|\lambda_{max}|}{|\lambda_{min}|}$. A small condition number systems are easier to solve and are called well-conditioned, while a large condition number problems are ill-conditioned and in general harder to solve. The steepest descent is very sensitive to the condition number because its convergence after k steps is bounded by:

$$\|e_{k+1}\|_A \leq \|e_0\|_A \left(\frac{\kappa - 1}{\kappa + 1}\right)^k, \tag{1}$$

where $\|x\|_A = \sqrt{x^T A x}$. On the other hand, the Conjugate gradient minimizes the residual in sequence of Krylov subspaces and is less dependant on the condition number of the matrix [2]:

$$\|e_{k+1}\|_A \leq 2\|e_0\|_A \left(\frac{\kappa^{1/2} - 1}{\kappa^{1/2} + 1}\right)^k. \tag{2}$$

3.3 General solver

Previously described algorithms impose different requirements on the matrix. In MATLAB the user does not need to have knowledge about the underlying methods, because it can recognize the best possible method for a given problem. Here we implement a function with a similar idea. Nota that our *linsolve* function is much simpler as we did not implement many of the pre-conditioners that exist in MATLAB or the upper-Hessenberg triangularization. We check whether the matrix is strongly row or column diagonally dominant and if it is, we use the Gauss-Seidel method.

Otherwise we employ the LU solver. This solver can also be used with the overloaded / operator as can be seen in Code listing 6.

Code listing 6: Linear solver example.

```
// Code listing 6 (lines 91 to 98):
TMatrix<double> Amat(RandomGauss<double>(5, 5, 2.0, 3.0), "Amat
    ");
TMatrix<double> xsol(RandomGauss<int>(5, 1, 3.0, 4.0).toDouble
    (), "xsol");
TMatrix<double> bvec(Amat*xsol, "bvec");
cout << transpose(bvec/Amat);   // calls linsolve(Amat, bvec)
//xvec^T = (10, 4, -4, 5, 2)
cout << transpose(xsol);
//xsol^T = (10, 4, -4, 5, 2)
```

4 Numerical comparison of solvers

To compare the efficacy of the implemented solvers we designed the following experiment. We construct an $N \times N$ random matrix A, such that $A_{i,j} = \mathcal{N}(0, 1)$. In order to get a symmetric matrix, with a strong diagonal we create the matrix T:

$$T_{i,j} = \begin{cases} A_{i,j} + A_{j,i} & i \neq j; \\ A_{i,j} + A_{j,i} + 3N & i = j; \end{cases} \tag{3}$$

This construction ensures that T is symmetric and with an overwhelming probability also diagonally dominant with a non-zero diagonal and therefore it is also positive definite. We construct a solution x in a similar way: $x_i = \mathcal{N}(2, 5)$. The right-hand side of the equation b is then defined as $b = Tx$ and we solve this problem for varying $N \in [50, 100, 150, \ldots 950]$.

In Figure 1 we can see that the exact solvers like LU and QR are faster for smaller systems but slower for very large systems, as expected. On the other hand, iterative solvers are typically slower for small systems and become much faster for large systems compared to LU and QR.

The steepest descent is much slower. This is caused by how we create our random systems. More specifically, they are ill-conditioned. Since the Steepest descent

convergence depends on the conditional number, when we generate a system with a large condition number we get a very slow convergence.

There is also a fundamental difference in the implementation of the LU, QR, Jacobi, Gauss-Seidel and Conjugate gradient, Steepest descent methods. While the first group is implemented using element-wise operations, the second group is formulated using matrix operations. We can recall from Section 1.3 that every matrix operation requires us to initialize and create a new space for a resulting matrix. This new matrix is immediately copied to the memory address of another matrix and deleted. This causes the Steepest descent and Conjugate gradient to be less efficient compared to other mentioned methods. All of the mentioned solvers achieve similar L_∞ error around machine tolerance of 10^{-16}.

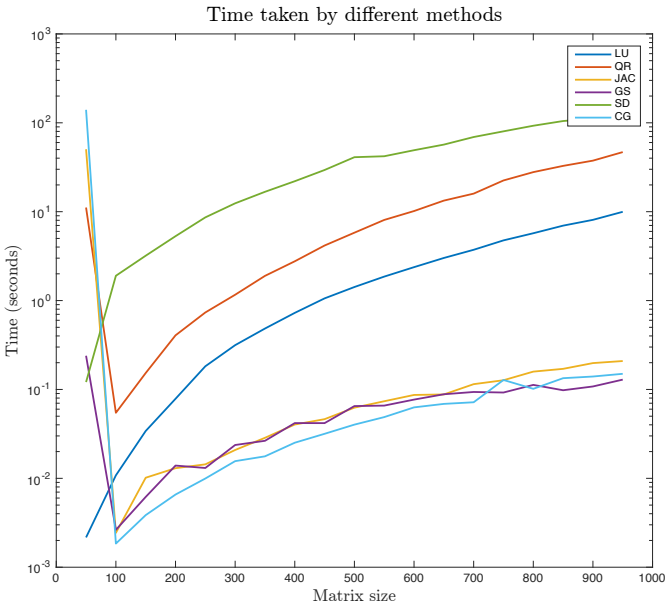

Figure 1: Time taken until convergence for different solvers.

Conclusion

We were able to implement six linear solvers and numerically compare them. These solvers performed as expected, with the only difference being in the Steepest descent and Conjugate gradient which are slightly slower due to the implementation through matrix operations. However, typically we were able to solve a system up to 2000×2000 in a reasonable time, which is performance comparable with MATLAB.

In this analysis topic we implemented C++ library for matrix operations. This required an additional focus to efficient memory management and using basic functionalities of C++, such as classes and templates. Overall we wrote over a thousand lines of code. However, there still remain certain additional functions we could have developed, for example for saving matrices in the binary format or making fast assignment operations instead of copying matrices. This could have been done by an additional overloading of operators for pointers. Furthermore, it would be interesting to take a closer look at ways in which we could parallelize parts of our code.

References

[1] R. M. Gower and P. Richtárik. Randomized Iterative Methods for Linear Systems. pages 1–32, 2015.

[2] L. N. Trefethen and D. Bau III. *Numerical linear algebra*, volume 12. 1997.

Appendix

Here we present the main script, which was partly included in the main text. For the full code including matrix class implementation please see: `https://github.com/can1003876/cpp`.

Code listing 7: Structure of the TMatrix class.

```
template <typename T>
class TMatrix {
private:
        T *mData;       // Matrix data (instead of: T **mData)
        int mShape[2];  // Dimensions of the matrix
        int mSize;      // Number of elements in the matrix
        string mName;   // Matrix name
public:
        // Functions:
        TMatrix(int sizeRows, int sizeColumns, string Name);
        // Constructor with specified elements;
        TMatrix(int sizeRows, int sizeColumns, T elements, string
            Name);
        TMatrix(const TMatrix<T>& rOther);
        // Copy Constructor with specified name
        TMatrix(const TMatrix<T>& rOther, string Name);
```

```
        virtual ~TMatrix();             // Destructor

        // Getting/Changing data
        T& operator()(int i, int j);
        // if only one index is specified, returns column/row
           vector.
        TMatrix<T> operator()(int ind);
        void swap(int i1, int j1, int i2, int j2);
        // Swap two values in the matrix
        void setElement(int i, int j, T value);
        const T getElement(int i, int j) const;
        void transpose();

        // Overloaded explicit casts for type conversion
        TMatrix<int> toInt();
        TMatrix<double> toDouble();

        // Matrix operations
        TMatrix<T>& operator=(const TMatrix<T>& rMatr);
        bool operator==(const TMatrix<T>& rMatr);
        TMatrix<T> operator+(const TMatrix<T>& rOther);
        TMatrix<T> operator-(const TMatrix<T>& rOther);
        TMatrix<T> operator-();
        TMatrix<T> operator*(const TMatrix<T>& rOther);
        TMatrix<T> operator*(T scalar);
        //      TMatrix<T> operator/(const TMatrix<T>& rOther);
        void rMAdd(const TMatrix<T>& rOther);

        // Getting/Changing information
        void setName(string newName);
        const string getName() const;
        const int getShape(int dim) const;
        const int getShape() const;
        const int length() const;

        friend ostream& operator << <> (ostream& output, const
           TMatrix& rMatr);
};
```

Code listing 8: ProjectMatlib.cpp

```
//============================================================
// Name         :ProjectMatlib.cpp
// Author       :1003876
// Version      :
```

```cpp
// Copyright   :Your copyright notice
// Description :Showcase of TMatrix class + solver comparisons
//============================================================

#include <stdio.h>
#include <stdlib.h>
#include <cassert>
#include <string>
#include <iostream>
#include <math.h>
#include <time.h>

#include "TMatrix.h"
#include "func_main.h"


using namespace std;

int main(void) {


// Code listing 1 (lines 26 to 51):
// 2x3 double matrix "A":
TMatrix<double> A(RandomGauss<double>(2, 3, 1.0, 3.0), "A");
A(2,2) = 1; // Change A_{2,2} == 1:

// Create integer matrix B by rounding elements of A:
TMatrix<int> B(transpose(A).toInt(), "B");
cout << A;
//A =
//(0.115494, 4.04615, 7.23705)
//(-0.752548, 1, 0.569711)

cout << B;
//B =
//(0, -1)
//(4, 1)
//(7, 1)

// Transpose B, cast it as double and add it to A:
cout << A+transpose(B).toDouble();
//A+B^T-double =
//(0.115494, 8.04615, 14.237)
//(-1.75255, 2, 1.56971)
```

```
cout << A*B.toDouble();          //name : "A*B-double"
cout << A.toInt()*B;             //name : "A-int*B"

// Code listing 2 (lines 53 to 62):
cout << A;
//A =
//(0.115494, 4.04615, 7.23705)
//(-0.752548, 1, 0.569711)

cout << A(-1);
//A-1row = (0.115494, 4.04615, 7.23705)
cout << transpose(B(1).toInt());        // B = A^T.toInt()
//B-1column^T = (0, 4, 7)

// Code listing 3 (lines 64 to 72):
TMatrix<double> C(RandomGauss<double>(5e3, 5e3, 0.0, 3.0), "C");
TMatrix<double> D(RandomGauss<double>(5e3, 5e3, 0.0, 3.0), "D");
clock_t start = clock ();
C = C+D;         // It took: 0.317092 sec.
printf("It took: %Lf sec.\n", (long double) (clock() - start)/
    CLOCKS_PER_SEC );
start = clock ();
C.rMAdd(D);      //It took: 0.085300 sec.
printf("It took: %Lf sec.\n", (long double) (clock() - start)/
    CLOCKS_PER_SEC );

// Code listing 4 (lines 74 to 89):
TMatrix<int> E(2, 2, 1, "E");
cout << E;
//E =
//(1, 1)
//(1, 1)
cout << transpose(diag(E));
//diag(E)^T = (1, 1)
cout << diag( diag(E) );
//diag(diag(E)) =
//(1, 0)
//(0, 1)
saveMat(diag( diag(E) + transpose(B(-1)) ), "diag.csv");
// diag.csv can be loaded in MATLAB with: csvread('diag.csv');
cout << sum(E) + norm(diag(E)) - sqrt(2); cout << "\n";
// 4

// Code listing 6 (lines 91 to 98):
TMatrix<double> Amat(RandomGauss<double>(5, 5, 2.0, 3.0), "Amat");
```

```
TMatrix<double> xsol(RandomGauss<int>(5, 1, 3.0, 4.0).toDouble(),
    "xsol");
TMatrix<double> bvec(Amat*xsol, "bvec");
cout << transpose(bvec/Amat);   // calls linsolve(Amat, bvec)
//xvec^T = (10, 4, -4, 5, 2)
cout << transpose(xsol);
//xsol^T = (10, 4, -4, 5, 2)

// Code listing 7 (lines 100 to 168):
// Start/end size, and step in iterations
int aN = 50, bN = 1000, dN = 50;
int MAX_ITER = 5000;
// Matrices to save times and errors
TMatrix<double> times(6,(int) (bN-aN)/dN, "times");
TMatrix<double> errors(6,(int) (bN-aN)/dN, "errors");

int k = 1; long double time = 0;
for (int N=aN; N<=bN; N=N+dN){
                TMatrix<double> T(RandomGauss<double>(N, N, 0.0,
                    1.0), "T");
                TMatrix<double> S(N, N, 3*N, "S");
                S = diag(diag(S));      // diagonal matrix full of
                    3*N elements
                T = T + transpose(T) + S;       // positive
                    definite-diagonally dominant matrix
                TMatrix<double> x(RandomGauss<double>(N, 1, 2.0,
                    5.0).toDouble(), "x");
                TMatrix<double> b(T*x, "b");

//              LU solver
                start = clock ();
                TMatrix<double> xLU(solveLU(T, b, false), "xLU");
                time = (long double) (clock() - start)/
                    CLOCKS_PER_SEC;
                printf("Finished solveLU. L2 error is: %1.3e and
                    it took: %Lf sec. \n", norm(x-xLU), time);
                times(1,k) = time;
                errors(1,k) = norm(x-xLU);

//              QR solver
                start = clock ();
                TMatrix<double> xQR(solveQR(T, b), "xQR");
                time = (long double) (clock() - start)/
                    CLOCKS_PER_SEC;
                printf("Finished solveQR. L2 error is: %1.3e and
```

```
                    it took: %Lf sec. \n", norm(x-xQR), time);
130                 times(2,k) = time;
                    errors(2,k) = norm(x-xQR);

    //              JACobi solver
                    start = clock ();
135                 TMatrix<double> xJAC(solveJAC(T, b, MAX_ITER), "
                        xJAC");
                    time = (long double) (clock() - start)/
                        CLOCKS_PER_SEC;
                    printf("Finished solveJAC. L2 error is: %1.3e and
                        it took: %Lf sec. \n", norm(x-xJAC), time);
                    times(3,k) = time;
                    errors(3,k) = norm(x-xJAC);
140
    //              GaussSeidel solver
                    start = clock ();
                    TMatrix<double> xGS(solveGS(T, b, MAX_ITER), "xGS"
                        );
                    time = (long double) (clock() - start)/
                        CLOCKS_PER_SEC;
145                 printf("Finished solveGS. L2 error is: %1.3e and
                        it took: %Lf sec. \n", norm(x-xGS), time);
                    times(4,k) = time;
                    errors(4,k) = norm(x-xGS);

    //              Steepest descent solver
150                 start = clock ();
                    TMatrix<double> xSD(solveSD(T, b, MAX_ITER), "xSD"
                        );
                    time = (long double) (clock() - start)/
                        CLOCKS_PER_SEC;
                    printf("Finished solveSD. L2 error is: %1.3e and
                        it took: %Lf sec. \n", norm(x-xSD), time);
                    times(5,k) = time;
155                 errors(5,k) = norm(x-xSD);

    //              Conjugate gradient solver
                    start = clock ();
                    TMatrix<double> xCG(solveCG(T, b, MAX_ITER), "xCG"
                        );
160                 time = (long double) (clock() - start)/
                        CLOCKS_PER_SEC;
                    printf("Finished solveCG. L2 error is: %1.3e and
                        it took: %Lf sec. \n", norm(x-xCG), time);
```

```
                    times(6,k) = time;
                    errors(6,k) = norm(x-xCG);

                    k++;
            }
    saveMat(times);
    saveMat(errors);

    printf("Fin !\n");
    return 0;
    }
```