Design documentation for the NPRG041 semestral project

Ondřej Boška

April 15, 2023

Contents

1 Theme

This project acts as a library for solving systems of linear equations. There are 4 algorithms implemented to solve such systems: plain Gaussian elimination, LU decomposition with partial pivoting, QR factorization and Gauss-Seidel iterative method. All functions are programmed with templates and are able to work with most of the numerical types from the standard library or implemented by the user. To demonstrate this, the project comes with testing data for double and std::complex<double> types and 2 special number types are implemented, fractions and finite groups.

2 User documentation

2.1 Representation of the matrix

The Matrix <T> type is used to store matrices and the equation system. The type T must implement a specific *Numerical* concept. That enforces the support of common arithmetic operations $+, -, *, /$, comparison operator $<$, unary minus, abs(), checking for equality with an integer using the $==$ operator and the constructor T(int).

To create a matrix, user has multiple options.

- Empty constructor creates an empty matrix with dimensions 0×0, which can be later resized with the *resize()* method.
- *Matrix(int row_count, int column_count)* creates an empty matrix with given dimensions and initializes it with default values.
- static method *Matrix<T>::identity(int n)* returns an identity matrix of dimensions $n \times n$.

After creating a Matrix, elements can be changed with the [][] or (,) operator or with *set_value(i, j, val)* method. Elements can be accessed also by the [][] and (,) operators or with the *get_value(int i, int j)* method. The second parameter of the (,) operator has default value of 1, which can be useful if the Matrix stored is a vector with dimension $n \times 1$.

Additionally, from the *matrix_loader* class, two additional functions can be used to create a Matrix from data given in an input stream. If no input stream is given as a parameter, the standard input is used.

Function *load_from_stream(istream stream)* reads data in the following format:

row_count column_count

a_{11} a_{12} ... a_{1n}

a_{21} a_{22} ... a_{2n}

...

a_{n1} a_{n2} ... a_{nn}

Function *load_compact(istream stream)* might be more suitable for sparse matrices with most of theirs elements equal to zero. It reads data in following format:

$row\_count\ column\_count\ number\_of\_items$

$i_1\ j_1\ val_1$

$i_2\ j_2\ val_2$

...

$i_n\ j_n\ val_n$

After the matrix is initialized, following operations are possible on the matrix:

- *resize(int i, int j)* re-sizes the matrix to dimensions $i \times j$. If the dimensions given are smaller then the previous dimensions, data outside of the new dimensions are lost. If the dimensions are larger, elements are initialized with the default value of T.
- *get_row_count()* and *get_column_count()* methods can be used to retrieve information about the dimensions of the matrix.
- *get_row(int i)* returns the row at index i as a vector. Note that the vector is returned by a reference, so changes to the vector will be visible in the matrix. Calling the *get_row* method is equivalent to using a single [] indexer. *set_row(int i, std::vector<T> row)* sets the row at index i to the given row vector. If the size of the vector given is different from the column count, the method will raise an exception.
- *transpose()* returns a new matrix, that is a transposition of the matrix the method was called on.
- *copy_from(Matrix source)* replaces the dimensions and all values of the matrix this is called on with dimensions and values of Matrix source. This is equivalent to creating a deep copy of the source matrix.
- addition and multiplication is supported with the standard + and $*$ operators. If those operators are called on two matrices with incompatible dimensions, an exception is thrown.
- *is_square()* returns true if the matrix the method was called on has the same number of rows and columns.
- *print(ostream stream)* prints the matrix to given output stream. If no stream is given, the matrix is printed to the standard output.

2.2 Solving systems of linear equations

For solving systems of linear equations, four functions described below are present in the *LinSolver* class. All of them take matrices of dimensions $n \times n+1$ and return a matrix of dimensions $n \times 1$ representing the vector of solutions.

- *solve_elimination(Matrix system)* uses basic and straight-forward Gaussian elimination to solve given system of equations. If the system has infinite solutions, an exception will be thrown saying that the system has infinite solutions. If the system has no solution, exception will be thrown also.
- *solve_lu(Matrix system)* solves the system using the LU decomposition and forward and backward substitution. This function also uses partial pivoting to increase the numerical stability. If the system has infinite solutions, an exception will be thrown saying that the system has infinite solutions. If the system has no solution, exception will be thrown also.
- *solve_qr(Matrix system)* solves the system using the QR decomposition and backward substitution. This function requires for the type T to also implement the *sqrt()* method. If the system has infinite solutions, the function will either return one of them or throw an exception saying the system has infinite solutions. If it has no solution, an exception will be thrown saying the system has no solutions.
- *solve_gauss_seidel(Matrix system, int max_steps, T accuracy)* solves the system using the Gauss-Seidel iterative method. The max_steps parameter defines how many iterations can be done at most. The accuracy parameter defines how precise should the results be, that is the difference between the last 2 iterations. If the accuracy is satisfied, the function returns the result sooner than max_steps are used. Default value for the *max_steps* variable is 10000 and T() for the *accuracy* parameter. If the system has infinite solutions, the function will try to converge to one of them. If it has no solution, the function will fail to converge.

2.3 Decomposing matrices

In the *LinSolver* class, the user can use following methods to decompose a matrix:

- *LU_decompose(Matrix input, Matrix L, Matrix U, Matrix Pvec)* takes the input matrix and to given L, U and Pvec matrices stores the calculated LU decomposition. As Pvec, using *permutation_vector(int size)* is recommended. Function *permuation_vector_to_matrix(Matrix Pvec)* then returns a permutation matrix P such that $P \times A = L \times U$, where L is a lower triangular matrix and U is an upper triangular matrix. Important note: LU_decompose function will make modifications to given input matrix, if you do not want the source matrix destroyed, please make a copy. Reason for this is that making the copy inside of the function would make the LU_ solve function slower.
- *QR_decompose(Matrix input, Matrix Q, Matrix R)* takes the input matrix and to given Q and R stores the calculated QR decomposition such that $A = Q \times R$, where Q is an orthogonal matrix and R is an upper triangular matrix. This function requires for the type T to also support the *sqrt()* method.

2.4 Custom number types

Two additional number types satisfying the *Numerical* concept, *Fraction* and *FiniteGroup<N>* are available in the project.

The Fraction type uses signed long for storing the numerator and unsigned long for storing the denominator. A number represented by this type can be created with the constructor *Fraction(long n, unsigned long d)*. If empty constructor is called, a Fraction representing the value zero is created. The type is immutable, so it is not possible to change the value after initialization. However, the fraction can be normalized with the *normalize()* method, which divides both numerator and denominator with their greatest common divider. The Fraction type also support << and >> operators for streams, basic arithmetic operations between both another Fraction or int, all comparison operators and a explicit double conversion method.

The FiniteGroup type represents a finite cyclic group with a parameter

N, which says with what modulo should the modular arithmetic operations be calculated. Creating a value can be done with the constructor *FiniteGroup(int n)*. If an empty constructor is called, a FiniteGroup representing the value zero is created. This type also supports basic arithmetic operations, comparison operators and stream operators.

3 Technical documentation

3.1 Data types

3.1.1 Matrix

The Matrix<T> type uses std::vector of std::vectors to store elements of the matrix. The row count and column count is stored in an integer and is not visible from outside, it can be retrieved however with *get_row_count()* and *get_column_count()* methods. The values of these variables can be changed only by calling the *resize(int m, int n)* method.

If a Matrix of type T is moved, empty Matrix of type T is left in its place, with the vector empty and both row and column count changed to zero.

3.1.2 Fraction

The Fraction type uses long for storing the numerator and unsigned long for storing the denominator. This implies that the type is capable of storing both positive and negative values and that the information about the sign of the fraction is stored in the numerator.

The Fraction type is a struct and has a copy constructor that copies values of the numerator and denominator.

The type supports all comparison operators by implementing the starship operator. The equality operators are implemented separately. All basic arithmetic operators are also implemented.

3.1.3 FiniteGroup

The FiniteGroup<N> takes integer N as a parameter to perform arithmetic operations modulo N. The value is stored in a private integer variable.

Two private methods are present in the FiniteGroup type. The *extended_gcd* method computes the extended euclidean algorithm for number x and y to find a and b such that $ax + by = gcd(x, y)$. The *inverse* method

then uses the extended euclidean algorithm to find the inverse value to the value stored in the FiniteGroup. That is needed for the division operator. However, the inverse value to a given value may not exist (if N is not a prime number). In that case, the method throws an exception.

The type supports all comparison operators by implementing the starship operator. The equality operators are implemented separately. All basic arithmetic operators are also implemented.

3.2 Algorithms

3.2.1 LU Decomposition

The lower-upper (LU) decomposition is a decomposition of a square matrix A into a product $PA = LU$, where P is a permutation matrix, L is a lower triangular matrix and U is a upper triangular matrix. The implemented version is *LU decomposition with partial pivoting*, which adds the need for the P matrix, but is more numerically stable and can factorize all square matrices (The P matrix is encoded in a permutation vector in our case).

The algorithm for computing the decomposition is very similar to the Gaussian elimination algorithm. The algorithm goes column by column, first it finds the row with the element with the largest absolute value and switches it with the current row. This part is called partial pivoting. The switches made are stored in a permutation matrix. After finding such row, it goes row by row in the same manner as Gaussian elimination, but instead of trying to make all the elements under the pivot zeros, it stores there the change made to the current row. It is possible to view the U matrix as the row echelon form of the input matrix and the L matrix as all the elementary row operations performed on the matrix (more exactly the product of all the inverted elementary row operations matrices, which works because the inverse of a lower triangular matrix is also lower triangular, similarly with the multiplication).

The resulting matrix is equal to $L + U - E$ and it is very simple to separate the matrices. There is an additional condition for the L matrix, all the elements on the diagonal have to be equal to 1. This makes the decomposition unique.

This decomposition can be used (and is commonly used) to solve systems of linear equations. In that case, the left sides of the equations are used as the A matrix and the vector of right sides is used as the permutation vector. After computing the decomposition, we first solve for the vector y such that $Ly = b$ by forward substitution and the we solve $Ux = y$ by backward substitution.

3.2.2 QR decomposition

The QR decomposition is a decomposition of a matrix A into a product $A = QR$, where Q is an orthogonal matrix and R is a upper triangular matrix. This decomposition works with any square matrix.

In this project, this decomposition is implemented using the Householder transformations method (which tends to be more stable then using Gram-Schmidt process). The proof of this method is done by induction. For detailed explanation of the proof, I would recommend the book *Lineární algebra nejen pro informatiky*, from which I took inspiration when I was coding this algorithm. The construction of the proof then gives us the algorithm itself.

If the matrix A is invertible, the decomposition is unique. For other square matrices, the decomposition exists, but uniqueness is not guaranteed.

This factorization can also be used to solve systems of linear equations. For system $Ax = b$, we have $QRx = b$, from that we get $Rx = Q^{-1}b = Q^Tb$. This means that to solve for the vector x, we only need to multiply the vector of the right sides b with the transposition of matrix Q and then perform a single backward substitution.

3.3 Exceptions

All exceptions explicitly thrown by this program inherit from the *LinSolveBaseException* class. This is done so that the user can choose to catch only exceptions thrown by this library. The base class is a basic exception type and only allows to store a message of std::string. This string can be retrieved using the *what()* method.

Exceptions inheriting from this base exception include:

- *MatrixException*: Such exceptions are thrown when there are errors in basic matrix operations (incorrect dimensions when multiplying and adding, trying to replace some row with a row that has incorrect number of columns).
- *NumberTypeException*: exceptions thrown by operations done on Fraction and FiniteGroup types, for example when dividing by zero, or creating a fraction with denominator 0.
- *SystemSolverException*: exceptions thrown by errors caused when solving a system of linear equations, for eg. when the system is not solvable, or when the system is not square.

4 Main method

This project also comes with a *main.cpp* file. In this file a simple main function is implemented which reads one system of equations from the standard input and then tries to solve it using all of the implemented functions. Additionally, *std::chrono* library is used to measure the run time of each function.

To change the type of the matrix loaded, change the *using test_type* value. Note that some types might not implement the *sqrt()* function, so QR_solve might not work with them. If that happens to be the case, comment the try block with qr_solve out. Also, to make the LU_solve work with std::complex, a comparison operator for it is implemented in the *complex_extensions.h* file. Without it, pivotation would not be possible. The comparison is done simply by comparing the absolute values of the two complex numbers.

Note that not all functions must return correct solutions to every system. For example, the Gauss-Seidel is unable to compute the solution when there is a zero present on the matrix diagonal. This documentation also comes with some test data present in attached .txt file.

5 Conclusion

If the user plans to use multiple number types in his problem, this library might work well. However, if there is no need for switching number types,

making the functions specialized would allow for more optimizations and better performance overall. Also, the fraction type does not seem suited at all to be used in functions like the Gaussian elimination because it tends to overflow rather quickly (even when using pivotization and normalizing the fraction after every operation).