University of Science and Technology in Cracow

Thesis

Solving large linear systems with sparse matrices - practical part

Łukasz Jastrząb

Course: Mathematics
Specialty: Computational mathematics

Number of album: 202277

Promoter
...

Faculty of Applied Mathematics

Cracow 2014

Statement by author

I the undersigned Łukasz Jastrząb certify that the work was written independently and used (except the knowledge gained in college) only results of the work included in the bibliography.

.....
(Signature of the author)

Statement by the promoter

I declare that the work meets the requirements for graduate work.

.....
(Signature of the promoter)

Contents

Preface

This work is intended to give the user a tool, which is easy to use, and allows solving problems with sparse matrices in most efficient way. The full understanding of the usefulness of this tool requires advanced knowledge of the `C++` language and numerical linear algebra.

Since this is a scientific project, each user has a non-commercial license to use, and there is only one request:

please send any comments, suggestions and improvements to the e-mail: luk.jastrzab@gmail.com

This will help to improve the performance of this tool.

Keywords

sparse matrices, Gauss elimination, Householder decomposition, method of successive relaxation

Introduce

We will begin the presentation of two distinguishable because of the usage schemes, decribed by Zahari Zlatev in [1]. Both schemes are useful in order to solving sparse matrices problems by performing the calculations in most efficient way, and maintaining the object-oriented programming methodology provided by `C++` language. We start with presentation of all relevant fields of both schemes, basing on the example matrix from the general $\mathbb{K}^{5\times5}$ space.

1. `input_storage_scheme`

The first scheme is called `input_storage_scheme` and is used to perform static (ie. not changing the structure) operations on the matrix. Basically, the scheme consists of three arrays containing data for each non-zero element of the matrix (`AORIG[i]` contains value of `i`-th stored element, `RNORIG[i]` - its row number and `CNORIG[i]` - its column number), as shown on the figure below.

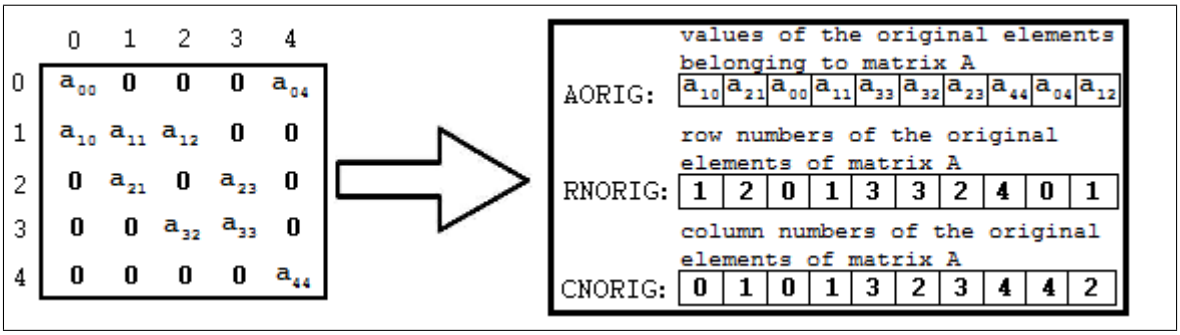


Figure 1: representation of an example matrix in `input_storage_scheme` object

Besides described lists of data, `class input_storage_scheme` contains other basic data which describes the stored matrix, such as constant fields `number_of_rows` and `number_of_columns` of type `size_t` (ie. `unsigned int`), useful field `order`, and field `NNZ` containing number of non-zero elements stored in scheme.

Note that order of storing elements should not be valid due to the effect of the algorithms applied to the scheme

The next important thing is that `class` is declared as a `template` with `typename TYPE`, which allows to store matrix over different algebraic structures (eg. real numbers, complex numbers, quaternions, so on). During describing the usage of particular methods, will be provided more detailed descriptions concerning planning and realizing the counting process.

To complete the basic desription (containing only mentioned methods and fields) of the `input_storage_scheme`, we provide the most precise desription of it as follows:

```
// ------------------------------------------------------------ code begin
template <typename TYPE>
class input_storage_scheme
{
public:
    /// number of added elements - sizeof arrays: AORIG, RNORIG, CNORIG
    const size_t NNZ;
    /// sizes of stored matrix
    const size_t number_of_rows, number_of_columns;
    /// not always real order of a matrix...
    /// just min(number_of_rows,number_of_columns)
    const size_t order;

private:
    /// array of ORIGinal values of the input matrix
    std::vector<TYPE> AORIG;
    /// array of ORIGinal row numbers of the input matrix
    /// (indexed from 0)
    std::vector<int> RNORIG;
    /// array of ORIGinal column numbers of the input matrix
    /// (indexed from 0)
    std::vector<int> CNORIG;

public:
    /// constructor and other public methods...

private:
  /// other private methods and fields

/// friends...
};
// -------------------------------------------------------------- code end
```

2. dynamic_storage_scheme

Second scheme is called `dynamic_storage_scheme` and is much more complicated. This scheme allows to perform dynamic operations such as GE (Gauss Elimination), or make some static operations more efficient (eg. iterating of SOR). Besides basic constants such as `number_of_rows` or `number_of_columns` of type `size_t`, the scheme consists of few particular parts which we will describe on an example matrix from (Figure 1).

The first part of a `dynamic_storage_scheme class` is called "row-ordered list" and contains (in its base form) values and column numbers of the matrix elements (ie. `ALU[i]` and `CNLU[i]` are respectively the value and the column number of the element stored on `i`-th position). Stored data is organized in compact way and form the so called "row packages". Processing this packages should always be done in such a way, that all of the elements belonging to the same row should lying side by side, and there were no free places between them.

After allocation of dynamic scheme, the row-ordered list, storing elements of example matrix from (Figure 1), should look like this:

Row-Ordered List (ROL)

NROL = 19 - size of row-ordered list
LROL = 9 - last not free position in ROL
CROL = 10 - number of stored elements

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
ALU :	a_{00}	a_{04}	a_{10}	a_{11}	a_{12}	a_{23}	a_{21}	a_{44}	a_{32}	a_{33}	-	-	-	-	-	-	-	-	-
CNLU:	0	4	0	1	2	3	1	4	2	3	-1	-1	-1	-1	-1	-1	-1	-1	-1
	r0		r1			r2		r4	r3										

Figure 2: base representation of the `ROL` in `dynamic_storage_scheme` object

The second part is called "column-ordered list" and contains only row numbers of the matrix elements. The data is organized in similar way as in `ROL` (this time it forms so called "column-packages"), and similar manners of proccessing should be applied to them, ie. all elements belonging to the same column should lying side by side, and there shouldn't be any free places beetwen them. In a considered example (Figure 1) column-ordered list after initialization should look this way:

Column-Ordered List (COL)

NCOL = 16 - size of column-ordered list
LCOL = 9 - last not free position in COL
CCOL = 10 - number of stored elements

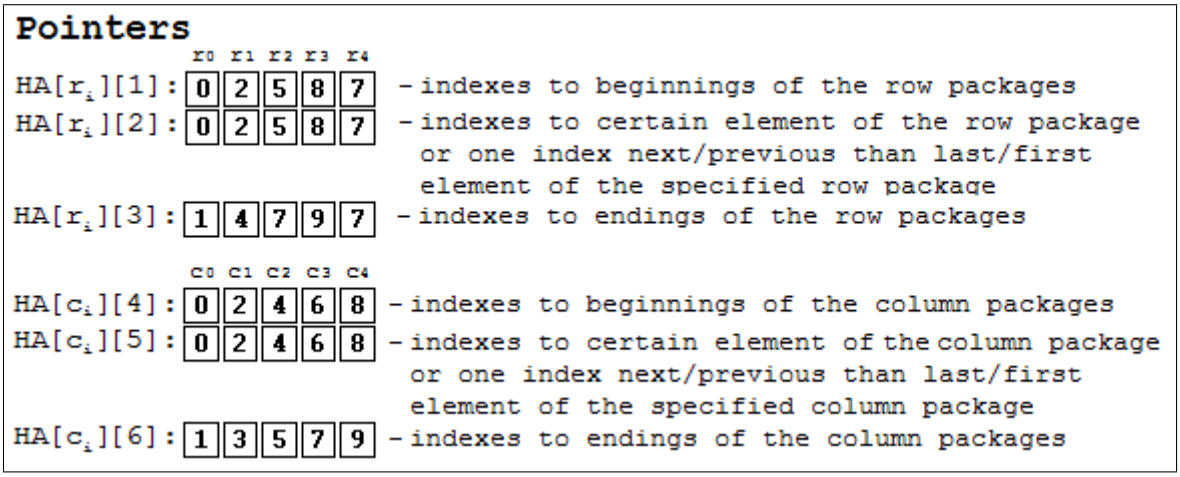
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RNLU:	0	1	1	2	1	3	2	3	0	4	-1	-1	-1	-1	-1	-1
	c0		c1		c2		c3		c4							

Figure 3: base representation of the `COL` in `dynamic_storage_scheme` object

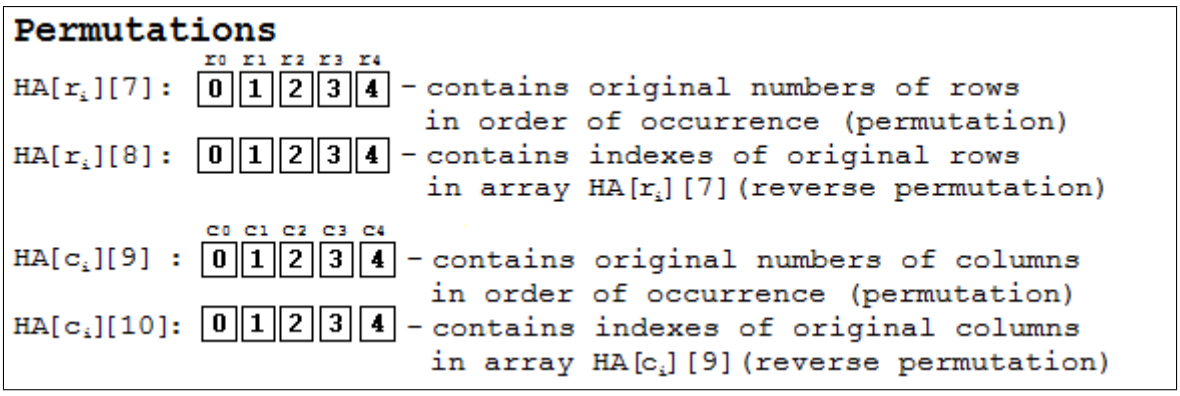
Note that there is additional `FREE` (marked by `-1`) places in both arrays. These places are intended to storing further elements arising during dynamic (ie. changing the structure) algorithms.

The performance of any calculation based only on described lists is impossible without the third part of `dynamic_storage_scheme class`, which is called the "integrity array", and consist of following data:

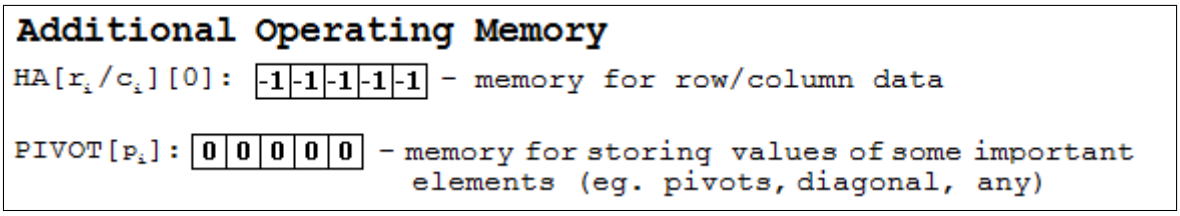
- indexes of the specific elements in `ROL` and `COL`



- permutations and reverse permutations of rows and columns



- additional memory for efficient performance of the algorithms



Implementation more complex calculations requires proper handling of all data forming part of the described modules. For this purpose has been written few useful methods that we will be discussing later in this documentation.

Remark. Introduced parts are in its basic initial form. Dynamic operations applied to the scheme can cause dispersal of row/column packages, hence begin/midle/end

pointers of those packages will be changed too. There should also be mentioned that midle pointers (`HA[*][2]` for ROL and `HA[*][5]` for COL) are used to separate for two groups all elements in particular row/column package. For example in GE it is used to separate active and inactive elements in a row/column package.

Described lists of elements could be initiated in a slightly different way, ie. values of elements (array `ALU`) can be stored together with their numbers of rows in column-ordered list. In this case row-ordered list will contain only column numbers of the matrix elements. Therefore this two parts of the `dynamic_storage_scheme` created on the matrix example from (Figure 1) should look this way:

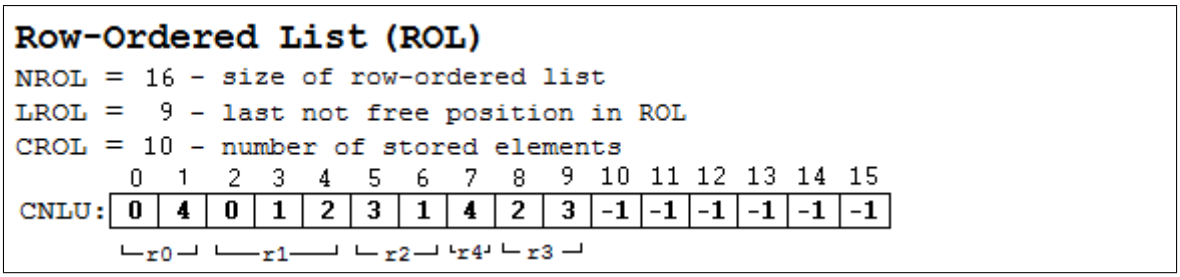


Figure 4: second representation of the `ROL` in `dynamic_storage_scheme` object

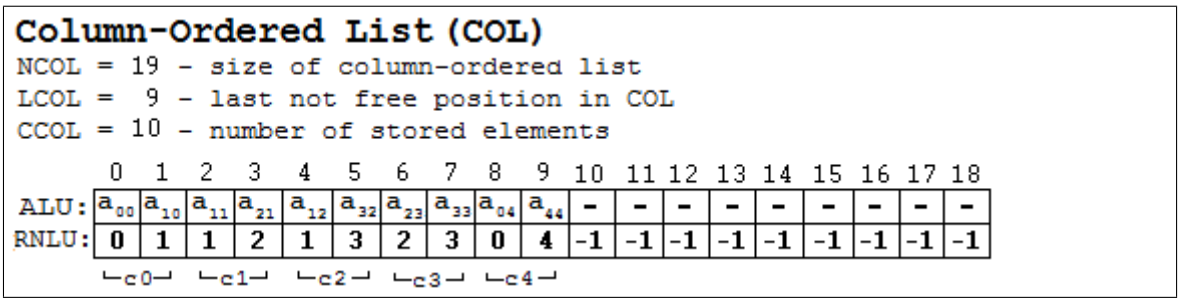


Figure 5: second representation of the `COL` in `dynamic_storage_scheme` object

Note that parts of integrity arrays should stay the same and only association of matrix elements values is changed. Both forms are distinguishable and marked as `ROL_INIT` and `COL_INIT` respectively. The `dynamic_storage_scheme` contains a special field of type `DYNAMIC_STATE` (`typedef enum`), that stores information of current state of the scheme. This field may contain one of the following values:

- `ROL_INIT` - base input state, used by most methods eg. `LU_decomposition`, or `iterative_preparation`. If we want to solve the system of linear equations (or count the determinant, eigenvalues, so on), it is likely that we will need to initialize a dynamic scheme to that state.
- `COL_INIT` - alternative state of the input, which can provide the performance of other algorithms.
- `ITERATIVE` - state of the scheme characterized by proper set of both lists and the integrity array contents, for optimal performance of SOR-iterative methods.

- LU_DECOMPOSED - state in which the matrix is decomposed into the product of the factors L and U, where L and U are lower and upper triangular matrices respectively.
- QR_DECOMPOSED - state in which the matrix is decomposed into the product of the factors Q and R, where Q is a orthogonal matrix and R is a upper triangular matrix.

To complete the basic description of the dynamic_storage_scheme we will provide its declaration as follows:

```
// ------------------------------------------------------------ code begin
template <typename TYPE>
class dynamic_storage_scheme
{
private:
    //======== MATRIX - BASIC INFORMATIONS ========
    /// sizes of stored matrix
    const size_t number_of_rows, number_of_columns;
    /// mostly  = min(number_of_rows, number_of_columns)
    const size_t order;

    //============== Row-Ordered List (ROL) =============
    size_t NROL;    /// size of row-ordered list
    size_t LROL;    /// last not free position in ROL
    size_t CROL;    /// number of non-zeros actualy stored in ROL
    TYPE *ALU;      /// array of values of the elements stored in scheme
    int *CNLU;      /// array of column numbers

    //=============== Column-Ordered List (COL) ===============
    size_t NCOL;    /// size of column-ordered list
    size_t LCOL;    /// last not free position in COL
    size_t CCOL;    /// number of non-zeros actualy stored in COL
    int *RNLU;      /// array of row numbers

    //============== INTEGRITY ARRAYS =============
    size_t NHA;     /// size of integrity arrays
                    /// NHA = max(number_of_rows, number_of_columns)
    TYPE *PIVOT;    /// table of pivots PIVOT[NHA]

    int **HA; /// Array of pointers and permutations HA[NHA][11]
    /**
      HA[i][0] - place to store working indexes during algorythmization

      HA[r][1] - first element in r-th row in ROL
      HA[r][2] - midle pointer in r-th row in ROL
      HA[r][3] - last element in r-th row in ROL
      HA[c][4] - first element in c-th column in COL
```

```
      HA[c][5] - midle pointer in c-th column in COL
      HA[c][6] - last element in c-th column in COL

      // permutations part
      HA[r][7] - original number of row, which is on r-th position now
      HA[r][8] - index in HA[*][7] where is placed r-th original row
      HA[c][9] - original number of column, which is on c-th position now
      HA[c][10] - index in HA[*][9] where is placed c-th original column
    */

    DYNAMIC_STATE dynamic_state;    /// variable indicating
                                    /// current state of the scheme

public:
    /// constructor and other public methods...

private:
    /// private methods and fields

/// FRIENDS...
};
// ------------------------------------------------------------ code end
```

Dynamic scheme contains also further fields, which we will discuss later during describing the associated methods.

3. Planning the calculations

Before detailed description of particular methods we should say something about overall methodology of performing the calculation that are intended to solve the linear systems.

Consider following diagram, where:

- thick lined are objects,
- thin rectangles means methods,
- clouds contains information about parameters passed by the user,
- dotted rectangle groups different methods, or different states of the same object.

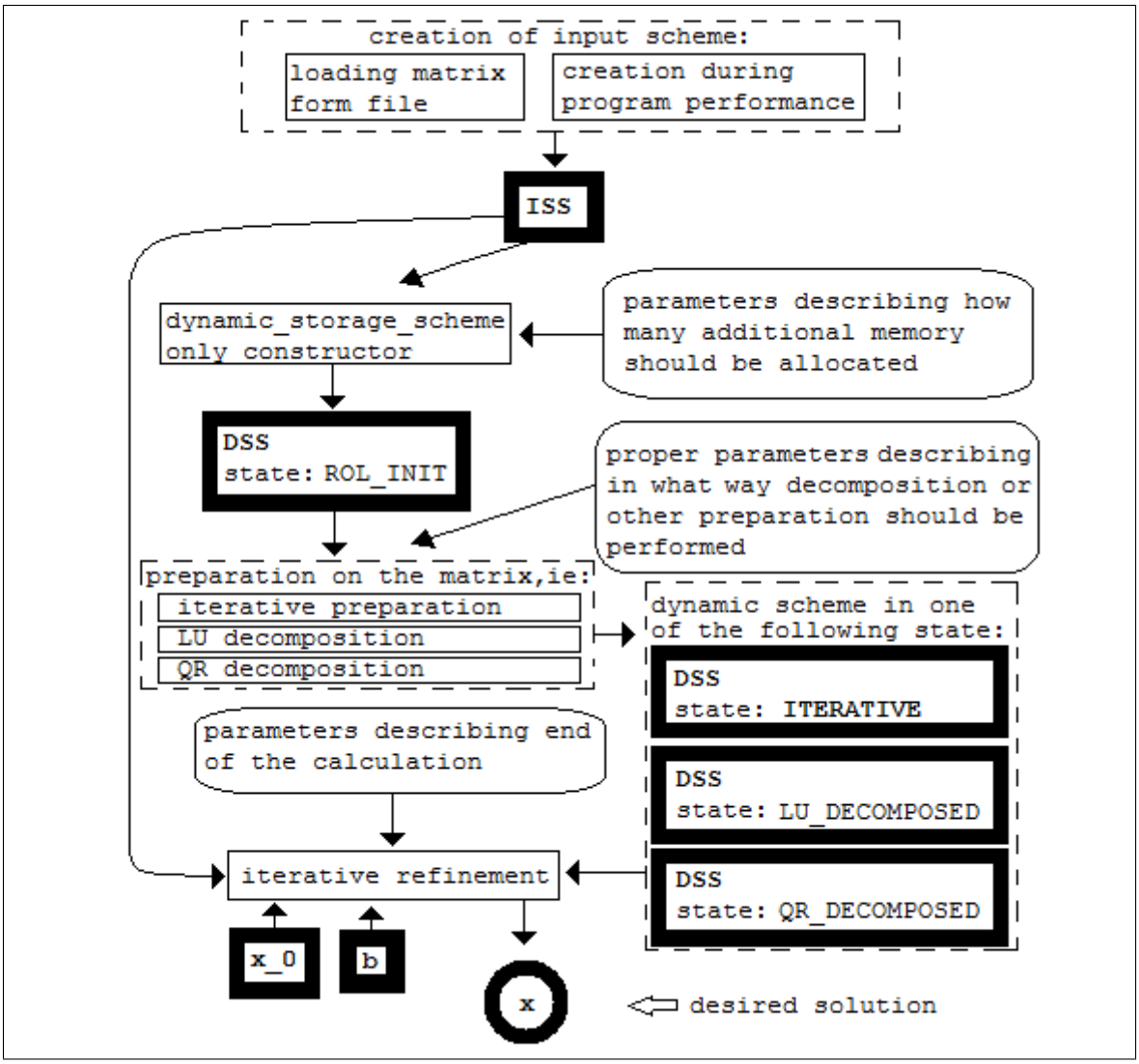


Figure 6: general approach to solving linear systems

As we can see on the draw above, the calculation process can be separated for a different variants of following general parts:

1. Creation of the `input_storage_scheme`.
2. Creation of the `dynamic_storage_scheme`.
3. Preparations on the `dynamic_storage_scheme`.

4. Performing the iterative refinement.

All these steps are described in details in second chapter (User Guide), where description of all public methods can be found.

The chapter third (Development Guide) contains description of all private methods that are already written and can be useful in order to expand the storing schemes of further functionalities.

Chapter 1

User guide

We start with a description of the methods with following motivating example of solving linear system using Gauss elimination. Let's look at the code below:

```
// ------------------------------------------------------------ code begin
1     // size of the problem
2     const size_t N = 100;
3
4     // vectors of the equation Ax=b
5     double x[N], b[N];
6
7     // input scheme containing matrix A
8     std_math::input_storage_scheme<double> ISS(N,N);
9
10    // random initialization of the matrix A
11    for (int row = 0; row < N; ++row)
12    {
13        for (int col = 0; col < N; ++col)
14        {
15            double val = get_random_element();
16            if (val != 0)
17                ISS.add_element(val,row,col);
18        }
19    }
20
21    // creation of the dynamic scheme containing matrix A
22    std_math::dynamic_storage_scheme<double> DSS(ISS,5,0.7,std_math::ROL_INIT);
23
24    // attempt to decompose the matrix A
25    try
26    {
27        DSS.LU_decomposition(std_math::MARKOWITZ_COST,10,2,0.001,true);
28    }
```

```
29    catch (std::exception e)
30    {
31        std::cout << e.what();
32        return 0;
33    }
34
35    // random initialization of the vector b
36    for (int row = 0; row < N; ++row)
37        b[row] = get_random_element();
38
39    // calculate the initial approximation (x_0)
40    DSS.solve_LU(x,b);
41
42    // perform interative refinement
43    DSS.iterative_refinement(ISS,x,b,0.00000000001,10);
// ------------------------------------------------------------ code end
```

In the above fragment of code we can easily distinguish four mentioned in "Planning the calculations" points. Namely:

1. Lines 8-19 are responsible for creation of the `input_storage_scheme`. In this particular example we create further elements by function `get_random_element`.
2. Line 22 is responsible for creation of the `dynamic_storage_scheme`. The creation is done by the only constructor that accepts few parameters inter alia already created the input scheme. Other parameters will be described in further section.
3. Lines 25-33 are responsible for preparations of the dynamic scheme. The member function `LU_decomposition` may throw some exceptions which will be further described along with parameters that this function uses.
4. Line 43 performs iterative refinement. Function `iterative_refinement` gets necessary data and two parameters describing end of the calculations, which will be described later.

1.1. Creation of the `input_storage_scheme`

We already know how to allocate the input scheme in most common way. First we used constructor:

```
input_storage_scheme(size_t number_of_rows, size_t number_of_columns)
```

and then we fill the scheme using method:

```
void add_element(TYPE value, size_t row, size_t column)
```

This method doesn't check if stored element is non zero and doesn't check if there was duplication of elements (ie. two elements of the same row and column). The developer must take care that there were no mentioned cases. It is worth to mention here that order of storing further elements is not valid due to performance of any algorythms, and numbers of rows and columns should be indexed from 0!

There is another way of allocation the input scheme. Namely, we can load it from

file. Further more, there are two methods of loading scheme. Following piece of code demonstrates how to do it:

```
// ------------------------------------------------------------ code begin
1     // declaration of the used namespace
2     using namespace std_math;
3
4     // declaration of two input schemes
5     input_storage_scheme<double> ISS1, ISS2;
6
7     // first way of loading scheme from file
8     load_scheme_from_file("example_scheme.txt", &ISS1);
9
10    // second way of loading scheme from file
11    load_matrix_from_file("example_matrix.txt", &ISS2);
12
13    // print schemes to console
14    std::cout << ISS1;
15    std::cout << ISS2;
// -------------------------------------------------------------- code end
```

In line 5 we allocate two input schemes using default constructor:

```
input_storage_scheme()
```

then in lines 8 and 11 we are loading the schemes from files, using two different methods. The difference between these two methods lies in format of the input files. First of these methods, namely:

```
template <typename TYPE2>
friend void load_scheme_from_file(const char* file_name,
                                  input_storage_scheme<TYPE2>* ISS)
```

uses files of following format:

```
number_of_rows
number_of_columns
value1   row1   column1
value2   row2   column2
value3   row3   column3
...
```

so the example content of the file may looks like this:

```
5
5
2.34    0   0
4.34    1   2
3.23    1   1
34      2   2
```

```
45      3   3
2.34    4   4
2.09    3   4
```

Second method, ie.:

```
template <typename TYPE2>
friend void load_matrix_from_file(const char* file_name,
                                  input_storage_scheme<TYPE2>* ISS)
```

uses files of format:

```
number_of_rows
number_of_columns
value11  value12  value13 ... value1n
value21  value22  value23 ... value2n
value31  value32  value33 ... value3n
...
valuem1  valuem2  valuem3 ... valuemn
```

The content of the example file may looks in a following way:

```
5
5
7.14     5.21        0         0         0
   0        0     2.12         0     20.02
   0     8.09        0         5         0
   0        2     13.6         0         0
   4        0        0      8.56         0
```

Last two lines (`14` and `15`) of the shown code passes the allocated schemes to the output stream by overloaded operator `<<`:

```
template <typename TYPE2>
friend std::ostream& operator<< (std::ostream& out,
                                 const input_storage_scheme<TYPE2>& ISS)
```

Format of the output passed to the stream is the same as the file format of the input to the function `load_scheme_from_file`.

The last method of the creation the input scheme is to copy it from the other, already created scheme. To do this, we use the standard copy constructor, namely:

```
input_storage_scheme(input_storage_scheme<TYPE>& ISS)
```

This method copies all fields of the input scheme and forms a scheme identical to the given scheme.

1.2. Creation of the dynamic_storage_scheme

We will now proceed to the presentation of how to create a dynamic scheme. We can do this by only constructor declared as follows:

```
template <typename TYPE>
dynamic_storage_scheme<TYPE>::
dynamic_storage_scheme( const input_storage_scheme<TYPE>& ISS,
                        double mult1,
                        double mult2,
                        DYNAMIC_STATE _dynamic_state
                       )
```

As we can see from the above declaration, following variables should be passed to the dynamic scheme constructor:

1. `ISS` - the input scheme, dynamic scheme is always created based on the input scheme. The input scheme is also needed to perform iterative refinement (as can be seen at Figure 6), so there is also a need to create the input scheme.
2. `mult1` - this parameter decides about size of the major list (ie. list containing `ALU` array). It gives `NROL = ISS.NNZ * mult1` in case of `ROL_INIT` or `NCOL = ISS.NNZ * mult1` in case of `COL_INIT`.
3. `mult2` - this parameter decides about size of the minor list (ie. list without `ALU` array). It gives `NCOL = NROL * mult2` in case of `ROL_INIT` or `NROL = NCOL * mult2` in case of `COL_INIT`. This variable has its default value set to `0.7`, it means that in mostly cases 70% of memory of the major list is needed to perform the algorythms.
4. `_dynamic_state` - this enumerate decides in which way the scheme should be allocated ie. row-ordered list as major or column-ordered list as major (respectively `ROL_INIT` or `COL_INIT`).

Note that `mult1` is very important parameter which decides how many additional memory should be allocated in order to perform dynamic operations on the scheme. For example if `GE` failed on non-singular matrix it is most likely that not enough memory was allocated.

1.3. Preparations on the dynamic_storage_scheme

This chapter is intended to present methods that prepare the dynamic scheme to solving linear system using iterative refinement.

1.3.1. LU_decomposition

Standard and most popular method for solving general systems is to decompose the system matrix into the factors LU using Gauss Elimination (GE). Performance of this method has been already presented on the motivating example, now we discuss carefully parameters which it applies, but first of all the declaration of this method:

```
template <typename TYPE>
void dynamic_storage_scheme<TYPE>::
LU_decomposition( PIVOTAL_STRATEGY strategy,
                  size_t _search,
                  double _mult,
                  double eps,
                  bool pre_sort
                 )
```

Parameters:

1. `strategy` - meaning of this parameter was detailed discussed in the theoretical part of the thesis, now we are obliged to say that it's standard enum, and it takes one of the following values:
   - `ONE_ROW_SEARCHING`
   - `MARKOWITZ_COST`
   - `FILLIN_MINIMALIZATION`

   it is worth to mention here that `MARKOWITZ_COST` is recomended in most cases.
2. `_search` - this parameter determines how many active parts of rows should be search in order to find the PIVOT. It has been proved that pivotal strategy based only on one row searching is numerically correct.
3. `_mult` - it is a stable parameter ie. it decides haw big (in absolute value) elements in comparision with largest we allow to be a PIVOT. For instance it allows elements no less than `_mult` times smaller then the bigest in considered row.
4. `eps` - this parameter is intended to discard small elements arised during elimination process. It should be much smaller than unity.
5. `pre_sort` - this parameter decides if pre sorting of rows should be performed. If true than rows will be sorted by quicksort algorithm in ascending order of number of non-zeroes. It is set by default on true if `ONE_ROW_SEARCHING` strategy is in use.

Left us only to discuss about some special cases that may occur in decomposition process. Function `LU_decomposition` may throw one of the following exceptions:

1. `LU_decomposition: ROL_INIT state is required`
2. `LU_decomposition: matrix is not squared`
3. `LU_decomposition: obtained singular matrix`
4. `LU_decomposition: not enough memory in ROL`
5. `LU_decomposition: not enough memory in COL`

First case occurs when we try to decompose already modified scheme or scheme in `COL_INIT` state (`ROL_INIT` state is required by function `LU_decomposition`). Second case occurs when we try to decompose not squared matrix (ie. `number_of_columns != number_of_rows`). Third exception means that matrix is singular or become singular during decomposition. In this case there is a possibility to try to manipulate `eps` parameter in order to obtain non singular factors if input matrix is not singular. Last two exceptions means that there is a need to allocate more memory in particular list.

1.3.2. `iterative_preparation`

Function of the following declaration:

```
template <typename TYPE>
void dynamic_storage_scheme<TYPE>::iterative_preparation(void)
```

is a function that prepares dynamic scheme for SOR iteration method. It shuffles elements inside the lists and sets proper pointers in order to make SOR iteration most efficient.

It is essential to mention here that SOR method is convergent only for a specific kind of matrices

Study following part of code to see how it works:

```
// ------------------------------------------------------------ code begin
1       using namespace std_math;
2
3       input_storage_scheme<double> ISS;
4       load_matrix_from_file("example_matrix1.txt", &ISS);
5
6       const int N = ISS.order;
7
8       dynamic_storage_scheme<double> DSS(ISS, 1, 1, ROL_INIT);
9
10      try
11      {
12          DSS.iterative_preparation();
13      }
14      catch (std::exception e)
15      {
16          std::cout << e.what();
17          return 0;
18      }
19
20      double *x, *b;
21      x = new double[N];
22      b = new double[N];
23      srand((size_t)time(NULL));
24      for (int i = 0; i < N; ++i)
25      {
26          x[i] = 0;
27          b[i] = get_random_number();
28      }
29
30      DSS.iterative_refinement(ISS, x, b, 0.000000001, 20);
31
32      delete[] x;
33      delete[] b;
// ------------------------------------------------------------ code end
```

Again we can easily distinguish stages that are described by the Figure 6 as follows:

1. Lines `3-4` - creation of the input scheme.
2. Line `8` - creation of the dynamic scheme, note that no additional memory is needed.
3. Lines `10-18` - an attempt of preparing the scheme to the SOR itereations. Function `iterative_preparation` throws an exception when it does not find at least one diagonal element, which means that at least one diagonal element is equal to zero. It also throws an exception when we try to prepare already disturbed scheme (ie. `dynamic_state != ROL_INIT`).
4. Lines `20-28` - preparation of the needed data such as first approximation `x` and right side vetor `b` of the equation `Ax=b`.
5. Line `30` performance of the iterative refinement.

1.4. Performing the iterative refinement

Now we proceed to detailed description of the method that is essential in order to get an accurate solution. Method is declared as follows:

```
template <typename TYPE>
void dynamic_storage_scheme<TYPE>::
iterative_refinement( const input_storage_scheme<TYPE>& ISS,
                      TYPE *x,
                      TYPE *b,
                      double acc,
                      size_t max_it
                    ) const
```

First of all, see that method is declared as `const`, so it can not change the structure of the dynamic scheme, and it takes following parameters:

1. `ISS` - the input scheme which was a base during creaetion of the dynamic scheme, this scheme also can not be modified by `iterative_refinement` method.
2. `x` - in/out parameter, first approximation (in) and desired solution (out).
3. `b` - right side vector of the equation $Ax = b$.
4. `acc` - required accuracy, iterative calculations stops if norm of the residual vector is equal or less than this parameter.
5. `max_it` - maximum permissible number of iterations.

Tt is worth to mention here that this method has involved another (the third) conditional of the end of iterative refinement process. Let us assume that r_i and r_{i+1} are residual vectors for iterations x_i and x_{i+1} respectively. If for some i we have $||r_i|| < ||r_{i+1}||$ than calculations should be stopped and iteration x_i has to be taken as solution.

Note also that `iterative_refinement` function does not need any information about method of itration. Algorithm itself recognizes state of the matrix and applies proper algorithm.

1.5. Solving systems with complex numbers

Chapter 2

Development guide

Bibliography

[1] Zahari Zlatev *Computational methods for general sparse matrices,* Kluwer Academic Publishers 1991