Novosibirsk Center of Information Technologies "UNIPRO"
https://unipro.ru/en/

Software package USPARS for solving sparse SLAE by Gauss method.
version 2.3.1

User Guide

12.06.2026

Table of Contents

Preface 5
Document structure 6
1. Definitions and data types 7
1.1 CSR format for storing sparse matrices 7
1.1.1. General sparse matrix 7
1.1.2. Storing a symmetric sparse matrix in CSR format 8
1.2. Portraits of sparse matrices 9
1.3. Special data types of the USPARS package 10
1.4. Permutations 10
2. User Functionality 11
2.1 uspars_alloc(): creation of USP structure 12
2.2 uspars_init(): preparation for factorization of the coefficient matrix 12
2.3 uspars_fact(): Triangular factorization of sparse matrices 15
2.3.1. Multiple factorization 16
2.4 uspars_solve(): Solving systems of linear equations with a factorized matrix of coefficients 16
2.4.1. Multiple solving 17
2.5 uspars_param_get(): access to some tt fields 17
2.6 uspars_param_set(): editing tt fields 18
2.7 uspars_destroy(): destruction of USP structure 19
3. Error codes 20
3.1 Generating errors 20
3.2 Input data errors (return code 2) 20
3.3 Memory allocation errors (return code 3) 20
3.4 Factorization errors (return code 4) 20
3.5 Errors when reversing factors (return code 5, reserved for future use) 20
3.6 Incorrect order of calling functions (return code 6) 20
3.7 Solution phase errors (Return code 7) 21
3.8 Errors in Metis usage (Return code 8) 21
3.9 Errors in the USPARS version used (return code 9) 21
3.10 Unclassifiable errors (return code 999) 21
A. Examples 22
A.1. Permutations and number of nonzero elements in factors 22
A.2. Measurement of phases operation time 23
A.3. Real symmetric positive-definite coefficient matrix with single precision 23

A.4. Real symmetric positive-definite coefficient matrix with double precision ... 23
A.5. Complex Hermitian positive-definite coefficient matrix with single precision ... 24
A.6. Complex Hermitian positive-definite coefficient matrix with double precision ... 24
A.7. Real symmetric coefficient matrix with single precision ... 25
A.8. Real symmetric coefficient matrix with double precision ... 25
A.9. An attempt to use the Cholesky decomposition for a non-positive-definite matrix ... 25
A.10. Complex symmetric coefficient matrix with single precision ... 26
A.11. Complex symmetric coefficient matrix with double precision ... 26
A.12. Complex Hermitian coefficient matrix with single precision ... 27
A.13. Complex Hermitian coefficient matrix with double precision ... 27
A.14. Real coefficient matrix with single precision ... 28
A.15. Real coefficient matrix with double precision ... 28
A.16. Complex coefficient matrix with single precision ... 29
A.17. Complex coefficient matrix with double precision ... 29
A.18. Real symmetric positive-definite coefficient matrix with single precision, two right-hand sides ... 29
A.19. Real coefficient matrix with single precision, two right-hand sides ... 30
A.20. Iterative refinement ... 31
A.21. Diagonal boosting to ‘work around’ Zero Pivot problem ... 31
A.22. Global pivoting to 'Bypass' Zero Pivot Problem ... 32
A.23. Using Out-Of-Core mode ... 32
A.24. Using long long int (64-bit) interfaces ... 32
B. Linear algebraic and algorithmic foundations ... 34
B.1. Gaussian elimination method ... 34
B.1.1. Real symmetric or complex Hermitian positive definite matrices ... 34
B.1.2. Symmetric (real or complex) and complex Hermitian coefficient matrices} ... 35
B.1.3. Coefficient matrices of general form ... 35
B.1.4. Structure of triangular factors ... 36
B.2. Balancing equation coefficients ... 37
B.3. Matching ... 37
B.4. Atlant ... 37
B.5. Bunch-Kaufman factorization ... 38
C. Practical recommendations for using the USPARS package ... 39
C.1. Using multithreading ... 39
C.2. Iterative refinement ... 39
C.2.1. Simple iterations ... 39

C.2.2. BiCGStab ... 40
C.3. Techniques for 'bypassing' the Zero Pivot ... 40
C.3.1. Diagonal boosting ... 42
C.3.2. Global pivoting ... 43
C.4. In-Core (IC) and Out-Of-Core (OOC) modes ... 44
D. Migration from MKL PARDISO to USPARS ... 46
D.1. Interfaces and solution phases ... 46
D.1.1. Memory allocation for the internal structure and default parameters setting ... 46
D.1.2. Running the solution phases ... 46
D.2. Matrix format and factorization type ... 47
D.3. Solver parameters ... 48
References ... 52

Preface

The technical capabilities of modern computers make it possible to solve systems of linear algebraic equations (SLAEs), comprising millions of equations, where coefficient matrices have the property of sparsity. Such SLAEs arise, in particular, as a result of discretization of two- or three-dimensional boundary value problems for partial differential equations. The sparsity of coefficient matrices is a specific property of such SLAEs, used in the design of methods for solving them and development of appropriate software. In programs, based on iterative algorithms, sparsity ensures the comparative cheapness of the matrix-vector multiplication operation.

The Gauss method, based on triangular factorization of the coefficient matrix, is an alternative way to solve SLAEs with sparse coefficient matrices. An incomplete list of libraries developed using the Gauss method includes

- PARDISO
- MUMPS
- Super LU
- WSMP

The Intel®Math Kernel Library should be added to this list, one of the components of which is a version of PARDISO, slightly different from the original one mentioned above. In the context of the review of methods and software products for solving SLAEs by direct methods, it is worth mentioning relatively new ideas for using intermediate data compression in computation process based on low-rank approximation algorithms [1, 2].

The USPARS package is another unique element in this series. Its functionality is intended for use on multicore computing systems with shared memory when solving SLAE using the Gauss method. The package can be used in in-core mode (IC), such that all intermediate data during the computation is stored in RAM, and out-of-core mode (OOC), with the option of storing some of the intermediate data on disk. Using disk memory for temporary storage of intermediate results in OOC mode allows to increase in the SLAE size available for solution on a given computer system. However, it should be noted that the increase in data exchange between RAM and disk during the solution process leads to an increase in computation time.

To ensure high performance of dense matrix operations, specifically for key functions such as GEMM and TRSM, the USPARS solver can be built and supplied linked against various mathematical libraries. This includes standard BLAS-compatible libraries such as OpenBLAS, distributed under the 3-Clause BSD License, or Intel® MKL, governed by the Intel Simplified Software License (ISSL), both of which expressly permit such use and redistribution. Also, the solver can be configured to use any other generic BLAS-compatible library or its own specially optimized internal kernels, referred to as UBLAS.

Support address: uspars-sup@unipro.ru

Document structure

Coefficient matrices of equations are considered to be presented in CSR format (Compressed Sparse Row) compact representation of sparse matrices. A description of this type can be found in Chapter 1. The concept of a portrait of a sparse matrix is also introduced there, and a summary of definitions of specific data types used in the USPARS package is provided.

USPARS functionality is divided into two levels. The main computing functions of the package belong to the lower level, hidden from the user. The purposes of these functions are narrow specialized, and the interfaces are not documented. Users have access to the upper level with the set of functions described in Chapter 2.

The document is supplemented with Appendices:

- Techniques for working with USPARS functionality are illustrated in the examples in Appendix A;
- The mathematical minimum underlying the package algorithms – see Appendix B;
- Practical recommendations for working with the package can be found in Appendix C;
- Help on migration from MKL PARDISO to USPARS can be found in Appendix D;

1. Definitions and data types

To run USPARS for the first time, you only need to know the input format of sparse matrices (see 1.1) and the supported scalar data types (see 1.3). For a better understanding of how the sparse solver works, we recommend that you study 1.2, 1.4, and Appendix B.

1.1 CSR format for storing sparse matrices

Matrices in which the number of non-zero elements is significantly less than the total number of elements are usually called *sparse*[1]. The abbreviation CSR stands for *Compressed Sparse Row*. When writing a matrix to this format, sparse rows are 'compressed'. In order to compress a row, you need to know its length, the number of non-zero elements in it, the positions of non-zero elements and their values. The positions of non-zero elements in a row are integer indices of the corresponding columns, the values of the elements are floating point numbers.

The CSR format is one of the most popular sparse matrix storage formats used in sparse matrix libraries. Specifically, the format is used in the PARDISO component of the Intel® MKL library. The CSR format in the USPARS package uses three arrays. There is also known a variation of CSR that uses four arrays.

1.1.1. General sparse matrix

To represent a general sparse matrix in CSR format[2], three linear arrays are needed. For a matrix A of order N having NNZ non-zero elements, these arrays are:

- `ia` - integer array of length `N+1`;
- `ja` - integer array of length `NNZ`;
- `a` - a real or complex array of length `NNZ`. Its representation accuracy (`float` or `double`) determines the accuracy of the matrix representation.

The variables `N`, `NNZ`, as well as the elements of the arrays `ia` and `ja` are `int`. The non-zero elements of the matrix are packed into the listed three arrays as follows. The first element `ia[0]` of the `ia` array is always zero, `ia[1]` is equal to the number of non-zero elements in the first row of the matrix. k=`ia[1]-ia[0]` non-zero elements of the first row are placed as the first k elements of array a in the same order as they appear in the row. Their column indices are placed in the first k elements of the `ja` array. The process is repeated for the second row of the matrix - the column indices of the non-zero elements are packed into the array `ja` in the order they appear in the matrix row, the corresponding values of the non-zero elements go into a, value of `ia[2]` becomes equal to the number of non-zero elements in the first two rows.

Continuing the process for the remaining rows of the matrix, the non-zero elements of the matrix are packed into three arrays. It is easy to understand that the last element of `ia[N]` is equal to `NNZ`. Note that the values of the elements of the `ja` array turn out to be ordered within each row. This requirement is mandatory for the functionality to work correctly.

The difference

`ia[i+1] - ia[i]`, (`i` = 0, 1, …, `N-1`)

is always equal to the number of non-zero elements of matrix A lying in the i-th row. According to this rule, if `ia[i+1] = ia[i]`, then in the `i`-th row of the matrix all elements are equal to zero. The correspondence between the non-zero elements of the matrix and the elements of the array a is established by the following formula

`a[k]` $= A_{i,ja[k]}$ for `ia[i]` ≤ k < `ia[i+1]`, (`i` = 0, 1, …, `N-1`).

[1] Sometimes matrices are called sparse if there are methods for solving SLAEs with such coefficient matrices in which taking into account the structure of non-zero elements leads to a gain in memory and solution time.

[2] The storage of symmetric (real and complex) and complex Hermitian matrices is described in subsection 1.1.2

Comment. In the above description, all indices, including indices of matrix elements, obey the rules of the C language: they start from zero (0-based indices). The rest of this document uses the zero- base index assumption.

The triple of arrays (`ia, ja, a`) used to describe the sparse matrix A in CSR format will be called the *CSR-matrix* A. For future references, we explicitly state the conditions that the elements of the index arrays must satisfy:

1. Ordering of `ia`

$$0 = \mathtt{ia[0]} \leq \mathrm{ia}[1] \leq \mathrm{ia}[2] \leq \ldots \leq \mathrm{ia}[n-1] \leq \mathtt{ia[N]} = \mathtt{NNZ}. \tag{1.1}$$

2. Ordering of `ja` in a row

$$0 \leq \mathtt{ja[k]} < \mathtt{ja[k+1]} < \mathtt{N} \quad \text{for } \mathtt{ia[i]} \leq \mathrm{k} < \mathrm{k}+1 < \mathrm{ia}[\mathrm{i}+1], 0 \leq \mathtt{i} < \mathtt{N}. \tag{1.2}$$

Here is an example of representing a matrix

$$\begin{pmatrix} 1 & 0 & 5 & 0 & 0 & 0 \\ 0 & 2 & 0 & 8 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 3 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \tag{1.3}$$

in CSR format. The contents of the arrays are shown in the following table

```
ia = 0,  2,  4,  6,  7,  10, 12;
ja = 0,  2,  1,  3,  0,  2,  3,  1,  4,  5,  4,  5;
 a = 1., 5., 2., 8., 1., 3., 1., 3., 2., 1., 1., 1.
```

Formally speaking, some of the matrix elements that are represented in the CSR format do not necessarily have to be zero.

For example, the next three arrays

```
ia = 0,  2,  5,  7,  9,  12, 14;
ja = 0,  2,  1,  3,  4,  0,  2,  1,  3,  1,  4,  5,  4,  5;
 a = 1., 5., 2., 8., 0., 1., 3., 0., 1., 3., 2., 1., 1., 1.
```

represent the same matrix (1.3), but now its so-called portrait (see section 1.2) is symmetrical. For these purposes, two zero elements of the matrix were included in the 'non-zero' family.

1.1.2. Storing a symmetric sparse matrix in CSR format

If a matrix has symmetry properties (symmetry $A_{ij} = A_{ji}$ of a real or complex matrix, or Hermitian $A_{ij} = \overline{A_{ji}}$ of a complex matrix), all its elements are uniquely restored from the elements of any (upper or lower) triangle of the matrix. In the USPARS package, the upper triangle, including the diagonal, is chosen to represent such matrices.

Let a symmetric matrix A of order N be represented in CSR format. The description presented in section 1.1.1 requires minor adjustments. For symmetric matrices, NNZ denotes the number of non-zero elements of matrix A lying in its upper triangle, including the diagonal.

Packing the elements of the upper triangle of the matrix into three CSR arrays is similar to the description from the previous section. The difference

`ia[i+1] - ia[i], (i=0, 1, …, N-1)`

is equal to the number of non-zero elements of matrix A lying in the `i`-th row of the upper triangle, including the diagonal.

Conditions for elements of index arrays representing a symmetric or Hermitian matrix (see (1.1), (1.2)):

3. Ordering of `ia`

$$0 = ia[0] \le ia[1] \le ia[2] \le \dots \le ia[N-1] \le ia[N] = NNZ; \qquad (1.4)$$

4. Ordering of `ja` in a row and belonging to the upper triangle

$$i \le ja[k] < ja[k+1] < N \ \ for \ \ ia[i] \le k < k+1 < ia[i+1], 0 \le i < N \qquad (1.5)$$

Let's finish this section with an example of representing a symmetric matrix

$$\begin{pmatrix} 4 & 0 & 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & -3 & -2 & 0 \\ 2 & 0 & 1 & 0 & 0 & 0 \\ 0 & -3 & 0 & 1 & 0 & 2 \\ 0 & -2 & 0 & 0 & 4 & 0 \\ 0 & 0 & 1 & 2 & 0 & 8 \end{pmatrix}$$

in CSR format

```
ia = 0,   2,   5,   6,   8,   9, 10;
ja = 0,   2,   1,   3,   4,   2,   3,   5,   4,   5;
 a = 4., 2., 3.,-3.,-2., 1., 1., 2., 4., 8.
```

Note. For the order of the matrix `N`, the number of non-zero elements `NNZ`, elements of the arrays `ia` and `ja`, it must be possible to represent them correctly. In the case of using 32-bit integers, these parameters should not exceed the value $2^{31} = 2147483648$. It is easy to understand that as the size of the matrices increases, a limitation on the number of non-zero elements most quickly occurs. This parameter is not independent in the description of the matrices, however, the values of elements of the `ia` array can reach the number of non-zero elements. To overcome this difficulty, 64-bit integers can be used in USPARS (see section 2.2.).

1.2. Portraits of sparse matrices

The so-called *portraits of sparse matrix* is a convenient tool for visualizing the structure of non-zero elements in a matrix. Figure 1.2.1 shows such portraits for the case of low-order matrices. The points represent the positions of non-zero elements in the matrix. When generating portraits, we assumed that the diagonal elements were non-zero, and this was clearly reflected in the drawings.

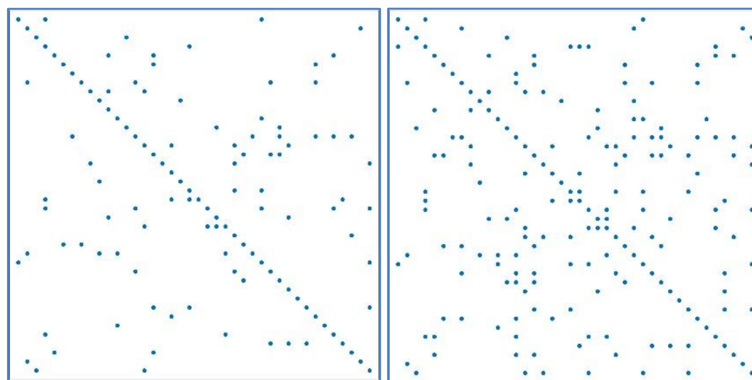

Fig. 1.2.1 Left: asymmetrical portrait of a sparse matrix.
Right: symmetrization of the portrait depicted on the left.

In the portrait on the right, the dots are arranged symmetrically relative to the diagonal of the matrix, while the portrait on the left does not have this property. It is said that the matrix, whose portrait is depicted on the right, has a *symmetrical portrait* . This does not mean that it itself is symmetrical — for this it is necessary that the matrix elements in symmetrical positions have the property of symmetry.

1.3. Special data types of the USPARS package

In the list of arguments to the functions of the USPARS package, pointers to arrays of floating-point numbers are converted to the type `void*`.

To transfer information about the actual type of array elements to the function, an additional parameter of the enumeration type `usp_scalar_type` is used, the list of possible values of which is presented in Table 1.3.1.

Table 1.3.1: Possible values for `usp_scalar_type` type parameters

Value	**Matching the standard type**
`USP_FLOAT`	`float`
`USP_DOUBLE`	`double`
`USP_COMPLEX_FLOAT`	`_Complex float`
`USP_COMPLEX_DOUBLE`	`_Complex double`

The USPARS package uses several types of triangular factorization of sparse matrices. The type of factorization is determined by the fact that the matrix belongs to one class or another (see section B.1). Each factorization type is associated with the enumeration type variable `usp_factorize_type` having the values presented in Table 2.2.2.

The functions `uspars_param_get()` and `uspars_param_set()` (see sections 2.5, 2.6) are used to extract some data obtained as a result of calculations, or to edit the values of parameters that control the progress of the computational process. The corresponding list is defined by the enumeration type variable `usp_param_data` (see Tables 2.5.2 and 2.6.2).

1.4. Permutations

Permutation matrices are often used in the theory and practice of algorithms for SLAEs with sparse coefficient matrices. A square $N{\times}N$ matrix P is called a *permutation matrix* if there is exactly one non-zero element in each row and column, and all non-zero elements are equal to one. In other words, the matrix P is obtained by rearranging the **rows** of the unit matrix. This fact can be written in the form

$$P_{ij} = \begin{cases} 1 & \text{for } j = p_i \\ 0 & \text{for } j \neq p_i \end{cases}, \qquad (j = 1,2,\dots,N), \tag{1.6}$$

where the concept of a *permutation vector* p with components p_i is used. It is clear that the vector p completely defines the matrix P.

Multiplying the permutation matrix P by vector a

$$\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} = P \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix}$$

gives vector b with components

$$b_i = a_{p_i}, \qquad (i = 1,2,\dots,N), \tag{1.7}$$

where the components of the permutation vector p are defined by formulas (1.6).

If the permutation matrix is multiplied on the right by row

$$[d_1 \quad d_2 \quad \cdots \quad d_N] = [c_1 \quad c_2 \quad \cdots \quad c_N]P,$$

then the corresponding formulas have the form

$$d_j = c_{q_j}, \qquad (j = 1,2,\ldots,N), \tag{1.8}$$

where the permutation vector q is defined by the permutation matrix P^t, which is the inverse of P:

$$PP^t = I.$$

From the formulas (1.7), (1.8) it is easy to obtain formulas for the elements of the AP and PA. Finally, in order to reduce the occupancy of the triangular factors obtained at the factorization stage, the so-called symmetric permutations $\tilde{A} = PAP^T$ matrices are used. The elements of the matrices A and $\tilde{A}$ are related by the following formulas

$$\tilde{a}_{ij} = a_{p_i p_j} \qquad (1 \le i,j \le N). \tag{1.9}$$

2. User Functionality

Among the arguments of each of the user-level functions of the USPARS package there is a special argument (of type void**) - the internal USP structure of the solver. It is necessary to create a variable of this type once and pass it to each called USPARS function. The user does not have access to or control over the fields of this structure.

In order to solve a system of linear equations with a sparse coefficient matrix, you need to sequentially call four functions of the USPARS package:

1. `uspars_alloc()` creates a USP structure for transferring data between functions;
2. `uspars_init()` prepares the coefficient matrix for factorization;
3. `uspars_fact()` performs the actual factorization of the coefficient matrix;
4. `uspars_solve()` forward and back substitution to determine the unknowns.

We especially note that the specified order

`uspars_alloc()` ⇒ `uspars_init?()` ⇒ `uspars_fact()` ⇒ `uspars_solve()` (2.1)

of calling the functions is predetermined by the fact that each function in this sequence prepares input data for the next function. Violation of the order (2.1) of calling functions leads to a specific error, but the user has the right to reuse the data calculated at previous stages to restart `uspars_fact()`/`uspars_solve()` (see 2.6). The USP structure is used to transfer data between solution stages.

Note 2.0.1. *If you need to solve several systems of linear equations with the same coefficient matrix, but with different right-hand sides, then this can be done after the* `uspars_fact()` *function has worked by calling the* `uspars_solve()` *function several times with the corresponding vectors of the right-hand sides.*

Each of the listed functions returns a return code. The value of return code indicates that the calculations were completed correctly, or an error. To avoid abnormal behavior, it is **strongly** recommended to analyze the return code before calling the next function.

2.1 `uspars_alloc()` : creation of USP structure

Syntax

```
int uspars_alloc(void **tt)
```

Include

`uspars.h`

Description

The function is used to create a USP structure `**tt`, used by the package functions for data exchange. For examples of using the `uspars_alloc()`, see the `examples` directory.

Return Codes

Code	What does it mean
0	Successful completion
3	Memory allocation error

2.2 `uspars_init()` : preparation for factorization of the coefficient matrix

Syntax

```
int uspars_init(void **tt, int n, int *ia, int *ja, void *a,
                usp_scalar_type utype, usp_factorize_type ftype,
                int *options)
```

Include

`uspars.h`

Description

The `uspars_init()` function is intended to prepare for factorization of the coefficient matrix of a system of linear equations

$$Ax = f.$$

Before calling the `uspars_init()` function, the `uspars_alloc()` function should be called (see section 2.1), which creates the USP structure `**tt`. The sequence of actions of `uspars_init()` is presented in the following list.

a) Pointers to arrays defining the coefficient matrix in CSR format (see section 1.1), are entered in the corresponding fields of the USP structure `**tt`.

b) Balancing (or scaling) of the coefficient matrix by multiplying it by specially selected diagonal matrices (see subsection B.2). Balancing can be turned on or off (for switching methods, see the description of the parameter `options[USP_SCALING]`). Instead of matrix A, the following matrix appears

$$A_1 = \begin{cases} D_l A D_r & \text{balancing turned on;} \\ A & \text{balancing turned off.} \end{cases}$$

c) Symmetric permutation of rows and columns of the matrix in order to reduce the degree of filling with non-zero elements of triangular factors. The result is a matrix $A_3 = PA_2P^t$ ($P = I$, if no permutations are used).

d) Symbolic factorization of matrix A_3. As a result, the portrait of the sparse matrix L (and the matrix U in the case of LU decomposition), and, accordingly, the amount of memory required to store its non-zero elements become determined. Allocation of memory for arrays of intermediate data and non-zero elements of triangular factors. Pointers to these arrays are stored in the fields of the `tt` structure variable.

Table 2.2.1: Arguments of the `uspars_init()` function

type	name	Assignment
in/out	`tt`	Pointer -to-pointer to USP structure
input	`n`	Matrix order, >0
input	`ia, ja, a`	Three arrays defining the CSR matrix (see section 1.1) of coefficients A.[1] The type of the pointer `a` is set to type `void*`, information about the true type of the elements of this array is transmitted through the value of the `utype` parameter
input	`utype`	Information about the type of non-zero matrix elements (see Table 1.3.1)
input	`ftype`	Information about the type of matrix factorization (see next)
input	`options`	An array (of length `USP_OPTIONS`) of options used by the USPARS package functions. The value of the `USP_OPTIONS` parameter is defined in `uspars.h`

The following table shows the correspondences between the factorization type, the permissible characteristics of matrix A, and the internal USPARS type passed as an argument.

$A = LL^T$	for	$A \in R^{n\times n}$	$A = A^T > 0$	`ftype`	`==`	`USP_LLH;`	(2.2)
$A = LDL^T$	for	$A \in R^{n\times n}$	$A = A^T$	`ftype`	`==`	`USP_LDLT;`	(2.3)
$A = LL^H$	for	$A \in C^{n\times n}$	$A = A^H > 0$	`ftype`	`==`	`USP_LLH;`	(2.4)
$A = LDL^H$	for	$A \in C^{n\times n}$	$A = A^H$	`ftype`	`==`	`USP_LDLH;`	(2.5)
$A = LDL^T$	for	$A \in C^{n\times n}$	$A = A^T$	`ftype`	`==`	`USP_LDLT;`	(2.6)
$A = LU$	for	A – general square matrix		`ftype`	`==`	`USP_LU.`	(2.7)

List of *valid values* for elements of the `options` array:

- `options[USP_MSGLVL]`
  - 0: zero level of logging (no printing to the screen).
  - 1: low level of logging - only basic warnings and factorization status in percent for large matrices.
  - 2: maximum level of logging (statuses and modes of internal functions, runtime profiling, ...).
  - 3: redundant level of logging, needed mainly for debugging[2]. It comes with extra memory overhead and reduced performance

- `Options[USP_RUN_MODE]` defines the mode of operation of the solver, namely a combination of reordering and factorization optimized for specific problem sizes. By default, the mode is selected automatically.

[1] Indices of matrix elements, elements of arrays `ia, ja` represent shifts relative to the beginning of the corresponding arrays and start from zero (zero-based indexing). The elements of arrays `ia, ja` satisfy conditions (1.1), (1.2) in the general case, conditions (1.4), (1.5) in the symmetric or Hermitian case.

[2] This level can be useful when communicating with the developers of the USPARS package if questions arise about a specific program execution by providing the information it produces.

- 0: auto select (***default value***).
- 1: Nested Dissection (ND) permutation and Panel Left Looking factorization. It is a well-parallelized mode optimized for large systems.
- 2: Approximate Minimum Degree (AMD) permutation and Row Right Looking factorization. The mode is designed for small systems.
- 3: Approximate Minimum Degree (AMD) permutation and Panel Left Looking factorization.
- 4: Nested Dissection (ND) permutation and Row Right Looking factorization.

- `options[USP_OMP_THREADS]` sets the number of OpenMP threads used by USPARS in internal parallel regions.
  - -1: use the ***default value***, equal to `omp_get_max_threads()` at the moment of the `uspars_init()` call.
  - `k`, `where 1 <= k <= omp_get_max_threads()`: use k OpenMP threads inside USPARS.

  If `options[USP_OMP_THREADS]` is less than 1 or greater than `omp_get_max_threads()`, it is replaced by the default value. USPARS does not call `omp_set_num_threads()` when processing this option, so the global OpenMP setting of the process is not changed.
- `options[USP_CHECK_LU]` enables additional diagnostics of factorization quality.
  - 0: diagnostics is disabled (***default value***).
  - 1: after factorization, maximum factorization error values and matrix/factor statistics are computed.
  - 2: additionally, a check with a random vector is performed and estimates related to the accuracy of applying the computed factorization are formed.

  Diagnostics requires additional time and memory, so it should remain disabled in production runs. To print diagnostic information, use `options[USP_MSGLVL]` >= 2.
  For the `uspars_init3264()` interface, `options[USP_CHECK_LU]=2` is automatically replaced by 1. For `options[USP_RUN_MODE]=2` and `options[USP_RUN_MODE]=4`, factorization quality diagnostics is automatically disabled.

- `options[USP_SCALING]` determines whether or not to enable balancing of the coefficient matrix (see section B.2). The value of this parameter is taken into account only when using LU-, LDLT- and LDLH factorizations. In the case of LLH-factorization, the value of this parameter is ignored, balancing is not applied.
  - 0: do not use balancing (***default value for LLH***).
  - 1: use balancing (***default value for LU, LDLT, LDLH***).

- `options[USP_ITER_REF]` - limit the number of iterative refinement steps. If `options[USP_ITER_REF]` == 0, then iterative refinement is not used (***default value***). For iterative refinement to be enabled, the value of `options[USP_ITER_REF]` must be positive. Iterative refinement stops when the limit on the number of steps is reached, or if the criterion for stopping by relative residual is met (see section C.2). ***Default values*** for parameter α in (C.1) are:
  - 1e-6 for single precision (`utype==USP_FLOAT` or `utype==USP_COMPLEX_FLOAT`);;
  - 1e-14 for double precision (`utype==USP_DOUBLE` or `utype==USP_COMPLEX_DOUBLE`).

  To change the value of the parameter α, use the function call

  ```
  uspars_param_set(&t, USP_ITER_STOP_CRIT, &value);
  ```

  where `t` is a pointer to the USP structure, `value` is a double number, the parameter value to be set.

- `options[USP_BOOSTING]` – enabling diagonal boosting (see section C.3.1) when calculating LDLT-, LDLH-, and LU factorizations to avoid the occurrence of a Zero Pivot error (see section C.3).
  - 0: Boosting is not using;
  - 1: Enable diagonal boosting for zero elements on the diagonal. (default value);
  - 2: Enable enhanced diagonal boosting.

With diagonal boosting enabled, some of the diagonal elements are modified during the factorization process. In order to determine which elements are being modified, the 'smallness level' δ_{boost} (by default 10^{-14} for options[USP_BOOSTING]=1 and 10^{-8} for options[USP_BOOSTING]=2). To set the value of δ_{boost}, use the AV function call.

```
uspars_param_set(&t, USP_BOOSTING_VALUE, &value),
```

where t is a pointer to the corresponding USP structure, value is a double precision number, and the set value is δ_{boost}. If diagonal boosting is enabled and iterative refinement is not enabled, then it is automatically enabled for 2 iterations for options[USP_BOOSTING]=2.

Note 2.2.1. *If the value of any element of the* options *array is equal to -1 (or does not correspond to the options described above), then during the calculation process this value is replaced by the default value.*

Return Codes

Code	What does it mean
0	Successful completion
1	*tt==NULL. Check that uspars_alloc() was called before uspars_init() and its return code is 0
2	Conditions for the input data are not met (see the list below)
3	Memory allocation error. Check the size of arrays.
9	Attempt to use unavailable functionality
999	Unclassifiable internal error

Conditions on the input data for the uspars_init() are presented in the list:

- n > 0;
- Pointers ia, ja, a are correctly defined;
- The diagonal elements of the complex Hermitian matrix are real;
- The elements of the ja array satisfy the conditions (1.1), (1.2) ((1.4), (1.5) in the symmetric case).

2.3 uspars_fact(): Triangular factorization of sparse matrices

Syntax

```
int uspars_fact(void **tt)
```

Include

```
uspars.h
```

Description

The function is designed to calculate triangular factorizations (see B.1), the type of which was determined at the previous stage (see 2.2).

The order of the matrix n and the array pointers (ia, ja, a) used to represent the matrix *A* in the CSR format are stored in the fields of the structure variable specified by *tt, which were pre-filled by function uspars_init(). The USP structure **tt is also the output parameter of the uspars_fact() function - factorization results are also passed through the **tt.

If options[USP_CHECK_LU] was enabled at the uspars_init() stage, additional diagnostics of factorization quality is performed after the factors are computed (see 2.2).

Return Codes

Code	What does it mean
0	Successful completion
1	`*tt==NULL`. Check the correctness of the code.
3	Memory allocation error. Check the size of arrays
4	'Zero pivot' (see sections C.3.1, C.3.2)
6	The sequence (2.1) of calling functions is broken
999	Unclassifiable internal error

2.3.1. Multiple factorization

После предварительной подготовки (см раздел 2.2) матрицы A, имеется возможность неограниченное число раз запускать `uspars_fact`. Это позволяет экономить время на решение систем с одинаковым портретом (одинаковым расположением ненулевых элементов). Единственным ограничением на переиспользование данных, полученных в ходе выполнение функции `uspars_init`, является отсутствие возможности использования `USP_MATCHING` (см раздел 2.1).

Для того, чтобы задать новые значения для ненулевых элементов исходной матрицы A, необходимо перед повторным запуском факторизации добавить их в структуру с помощью функции `uspars_param_set` (см раздел 2.6). Например, следующим образом:

```
uspars_param_set(&tt, USP_A_CSR, &a_updated);
uspars_fact(&tt);
```

After preliminary preparation (see Section 2.2) of matrix A, it is possible to run `uspars_fact` an unlimited number of times. This allows to reuse result of `uspars_init` when solving systems with the same pattern (the same positions of nonzero elements). The one limitation on reusing data obtained during the execution of the `uspars_init` function is the restriction to use `USP_MATCHING` (see Section 2.1).

To assign new values to nonzero elements of the matrix A, they must be added to the structure using the `uspars_param_set` function (see Section 2.6) before rerunning the factorization. For example, as follows:

```
uspars_param_set(&tt, USP_A_CSR, &a_updated);
uspars_fact(&tt);
```

2.4 `uspars_solve()`: Solving systems of linear equations with a factorized matrix of coefficients

Syntax

```
int uspars_solve(void **t, int m, void *b, usp_scalar_type utype)
```

Include

```
uspars.h
```

Description

The function is designed to solve a system of linear equations

$$AX = B \tag{2.8}$$

with a sparse $n \times n$ matrix of coefficients A and m vectors of the right − hand sides, represented in (2.8) by a rectangular $n \times m$ matrix B. The coefficient matrix is pre-factorized by calling the `uspars_fact()` function.

Table 2.4.1: Arguments of the `uspars_solve()` function

type	name	Assignment
input	`tt`	Pointer -to-pointer to USP structure
input	`m`	The number of right-hand sides
in/out	`B`	An array[1] (of length $n * m$) of floating-point numbers. At the input, the array contains components of column vectors of the right-hand sides, located in memory in columns (the j-th component of the k-th vector has the address $b + k * n + j$). At the output, in the same way, the components of the corresponding vectors of unknowns (columns of the matrix X) are located in the array b
input	`utype`	Information about the actual type of matrix B elements. The value of the parameter must match the value of the `utype`, which determined the type of elements of the coefficient matrix when calling the `uspars_init()` function. In upcoming releases of USPARS, support will be added for solving systems that have been factored with a different scalar type.

Return Codes

Code	What does it mean
0	Successful completion
1	`*tt==NULL`. Check the correctness of the code.
2	$m < 0$, or `b==NULL` or `utype` does not match the type of matrix elements
3	Memory allocation error. Check the size of arrays
6	The sequence (2.1) of calling functions is broken
7	Iterative refinement of the solution (see section C.2) did not converge
999	Unclassifiable internal error

2.4.1. Multiple solving

After factorization (see Section 2.3) of the matrix A, one can run `uspars_solve` an unlimited number of times for the same matrix with different right-hand sides. There are no restrictions on reusing the solver.

2.5 `uspars_param_get()`: access to some `tt` fields

Syntax

```
int uspars_param_get(void **tt, usp_param_data id, void *a)
```

Include

`uspars.h`

Description

[1] The pointer is converted to the `void*` type to universalize the interface; information about the real type of array elements b is passed through the `utype` parameter.

The function is used to extract data stored in the fields of a structural variable specified by the pointer `*tt`. Not all of this data is available to the user. The extracted data is specified by the value of the input parameter `id` (see Table 2.5.2).

Table 2.5.1: Arguments of the `uspars_param_get()` function

type	name	Assignment
input	`tt`	Pointer -to-pointer to USP structure
input	`id`	Parameter for specification of extracted data
output	`a`	A pointer to the array where to place the extracted information. Memory for the array elements must be allocated before calling the `uspars_param_get()` function

Table 2.5.2: Possible values of the id argument of the `uspars_param_get()` function

id value	Data Specification	Type[size] of array a
`USP_ERR`	Error code	int[1]
`USP_NNZCL`	Estimation the number of nonzero elements in L-factor	double[1]
`USP_FLPCL`	Estimation the number of FLOPS (multiplications) for factorization L-factor (without panels separation)	int[n]
`USP_BOOSTING_VALUE`	Current double precision number – tolerance level for 'support' (see section C.3.1)	double[1]
`USP_D_VECTOR`	Diagonal vector D (length n) from LDL^T and LDL^H factorization. Elements are copied.	utype[n]

Return Codes

Code	What does it mean
0	Successful completion
1	`*tt==NULL`. Check the correctness of the code
2	Incorrect argument values

2.6 `uspars_param_set()`: editing `tt` fields

Syntax

```
int uspars_param_set(void **tt,  usp_param_data id, void *a)
```

Include

```
uspars.h
```

Description

The function is used to edit information stored in the fields of a structure variable indicated by the `*tt` pointer. The type of information is determined by the value of the input parameter `id` (see Table 2.6.2)

Table 2.6.1: Arguments of the `uspars_param_set()` function

type	name	Assignment
input	`tt`	Pointer -to-pointer to USP structure

input	`id`	What to edit
output	`a`	Pointer to an array with data that should replace the existing ones

Table 2.6.2: Possible values of the id argument of the `uspars_param_get()` function

id value	**Data Specification**	**Type[size] of array a**
`USP_BOOSTING_VALUE`	Double precision number – tolerance level for 'support' (see section C.3.1)	double[1]
`USP_ITER_STOP_CRIT`	Double precision number – criterion for stopping iterative refinement (see section C.2)	double[1]
`USP_GP_EPS`	Double precision number – a smallness parameter used in the global permutation algorithm (see section C.3.2)	double[1]
`USP_PERM_VECTOR`	Permutation vector used to reorder the elements of the original matrix before factorization	int[n] (long[n])
`USP_A_CSR`	Array of updated values of matrix *A*. Can be set after `uspars_init` stage. Allows to reuse all data obtained at `uspars_init` stage for matrices with the same sparse portrait	utype[nnz]

Return Codes

Code	**What does it mean**
0	Successful completion
1	`*tt==NULL`. Check the correctness of the code
2	Incorrect argument values

2.7 `uspars_destroy()`: destruction of USP structure

Syntax

```
int uspars_destroy(void **tt)
```

Include

```
uspars.h
```

Description

The function is used to release the memory allocated at the initialization step. At the output of this function, the value of the `*tt` pointer is `NULL`.

Return Codes

Code	**What does it mean**
0	Successful completion
1	`*tt==NULL`. Check the correctness of the code.

3. Error codes

3.1 Generating errors

Return values of USPARS functions are given in the function descriptions. Return values are formed based on the codes of errors that may occur during the calculation process and which contain more detailed information about the error that occurred. This more detailed information may be useful in investigating a problem that has arisen while using USPARS functions. The `uspars_param_get()` function is used to extract such information (see section 2.5). In this section, we provide tables of error codes.

3.2 Input data errors (return code 2)

Numeric value	**Interpretation**
201	Parameter `N` is not positive
202	Pointer `ia` of the CSR matrix is `NULL`
203	Pointer `ja` of the CSR matrix is `NULL`
204	Pointer `a` of the CSR matrix is `NULL`
205	The immaterial diagonal of a matrix declared to be Hermitian
206	Mismatch between the types of elements of the coefficient matrix and the vector of the right-hand side
207	Parameter `NRHS` is not positive
208	Pointer to an array with elements of the right-hand side of the SLAE is `NULL`
210	Unordered elements in the `ja` array of the CSR matrix
212	Incorrect value of the `id` parameter in the `uspars_param_set()` function
213	Incorrect value of the `id` parameter in the `uspars_param_get()` function

3.3 Memory allocation errors (return code 3)

Numeric value	**Interpretation**
301	Memory allocation errors

3.4 Factorization errors (return code 4)

Numeric value	**Interpretation**
401	Non-positive-definite input matrix when using the Cholesky decomposition
402	Zero leading element when performing factorization (Zero pivot)

3.5 Errors when reversing factors (return code 5, reserved for future use)

3.6 Incorrect order of calling functions (return code 6)

Numeric value	**Interpretation**
601	Incorrect order of function calls: `uspars_fact()` called before `uspars_init()`
602	Incorrect order of function calls: `uspars_solve()` called before `uspars_init()`

603	Incorrect order of function calls: `uspars_solve()` called before `uspars_fact()`

3.7 Solution phase errors (Return code 7)

Numeric value	**Interpretation**
701	Iterative refinement did not converge within given number of steps

3.8 Errors in Metis usage (Return code 8)

Numeric value	**Interpretation**
801	Dynamic library Metis not found
802	It is not possible to call Metis.

3.9 Errors in the USPARS version used (return code 9)

Numeric value	**Interpretation**
901	An attempt to use functionality that is not included in the freely distributed version of USPARS

3.10 Unclassifiable errors (return code 999)

These errors have numbers[1] from 99900 to 99913.

[1] These numbers mean nothing to users, but are important to developers

A. Examples

The `examples` directory contains source codes in the C language for examples of using the functions the USPARS package to solve systems of linear equations with sparse coefficient matrices. The scenarios of the examples are of the same type:

- Initialize the matrix in CSR format.
- Initialize the right-hand side vector corresponding to the x_{exact} solution.
- In accordance with the solution algorithm, call functions sequentially:
    - uspars_init(),
    - uspars_fact(),
    - uspars_solve().

  Each subsequent function is called if the previous one has completed successfully; the return code is analyzed for verification.
- Print the vectors of the exact and numerical solution for comparison.

The difference between the exact and numerical solutions is due to the influence of rounding errors. The magnitude of this difference is influenced by both the conditionality number of the coefficient matrix and the operating precision used. For convenience, we present the values of the relative errors of the calculated solutions along with the values of the conditionality numbers of the coefficient matrix.

A.1. Permutations and number of nonzero elements in factors

This example shows comparative results for the number of nonzero elements in matrices L and U after factorization using different reorderings. The value `nnz(A)` in the output represents the number of nonzero elements in the original matrix, and `nnz(L+U)` is the sum of the numbers in L and U.

An asymmetric matrix is used as a matrix of coefficients https://sparse.tamu.edu/Mathworks/Sieber with a conditionality number of 4.613510e+08. The exact solution vector consists of units

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{2290} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

The corresponding right-hand side of the system of linear equations is obtained by multiplying the matrix by the exact solution vector.
Below is a table of results for the case of a solution without any reordering, using METIS and Atlant:

```
    w/o reorder results:
name              |    n    |nnz(A)   | nnz(L+U) |      error C     |  error L2    |
./mtx/Sieber.mtx |  2290 |   14873 |   3125286 |   7.622005e-09 | 1.748052e-10|

METIS results:
name              |    n    |nnz(A)   | nnz(L+U) |      error C     |  error L2    |
./mtx/Sieber.mtx |  2290 |   14873 |     41194 |   2.409236e-09 | 1.068810e-10|

Atlant reorder results:
name              |    n    |nnz(A)   | nnz(L+U) |      error C     |  error L2    |
./mtx/Sieber.mtx |  2290 |   14873 |     41174 |   1.782625e-09 | 2.290986e-10|
```

Here `nnz(A)` denotes the number of non–zero elements in the matrix A, and `nnz(L+U)` is the total number of non–zero elements in the matrices L and U, `error C and error L2` are the relative errors of the solution

$$\frac{\|x - x_{calc}\|}{\|x\|}$$

in the norms of the 'maximum modulus' and Euclidean, respectively. For the source code of the example, see the `example_compare_reorder.c.`

A.2. Measurement of phases operation time

This example illustrates how to get information about the execution time of the phases of the USPARS package. An asymmetric matrix is used as a matrix of coefficients https://sparse.tamu.edu/Mathworks/Sieber with a conditionality number of 4.613510e+08. The exact solution vector consists of units

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{2290} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

The corresponding right-hand side of the system of linear equations is obtained by multiplying the matrix by the exact solution vector. The output has the following form

```
USPARS Init  time: 0.028851 seconds.
USPARS Fact  time: 0.035071 seconds.
USPARS Solve time: 0.001889 seconds.
Results:
name              | n    | nnz(A) |    error C     |  error L2     |
./mtx/Sieber.mtx | 2290 |  14873 | 1.782625e-09 | 2.290986e-10 |
```

Here `nnz(A)` denotes the number of non–zero elements in the matrix A, and `nnz(L+U)` is the total number of non–zero elements in the matrices L and U, error C and error L2 are the relative errors of the solution

$$\frac{\|x - x_{calc}\|}{\|x\|}$$

in the norms of the 'maximum modulus' and Euclidean, respectively. Depending on the computer used, the time measurement results can vary greatly. For the source code of the example, see the `example_timer.c` file.

A.3. Real symmetric positive-definite coefficient matrix with single precision

The system of equations

$$\begin{pmatrix} 4 & 0 & -2 & 0 & 0 & 2 \\ 0 & 5 & 0 & -3 & -2 & 0 \\ -2 & 0 & 4 & 0 & 0 & 3 \\ 0 & -3 & 0 & 5 & 0 & 2 \\ 0 & -2 & 0 & 0 & 3.96 & -2 \\ 2 & 0 & 3 & 2 & -2 & 8.16 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \\ 5 \\ 4 \\ -0.04 \\ 13.16 \end{bmatrix}$$

See the source code of the example in the `example_LLH_s.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution     Numerical solution
 1.0000000e+00      9.9545372e-01
 1.0000000e+00      9.9966192e-01
 1.0000000e+00      9.9480432e-01
 1.0000000e+00      9.9823838e-01
 1.0000000e+00      1.0017973e+00
 1.0000000e+00      1.0038968e+00
```

The conditionality number of the coefficient matrix is 5.0787e+05, which justifies the relative deviation of 0.0034 of the approximate solution from the exact one.

A.4. Real symmetric positive-definite coefficient matrix with double precision

The system of equations

$$\begin{pmatrix} 4 & 0 & -2 & 0 & 0 & 2 \\ 0 & 5 & 0 & -3 & -2 & 0 \\ -2 & 0 & 4 & 0 & 0 & 3 \\ 0 & -3 & 0 & 5 & 0 & 2 \\ 0 & -2 & 0 & 0 & 3.96 & -2 \\ 2 & 0 & 3 & 2 & -2 & 8.16 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \\ 5 \\ 4 \\ -0.04 \\ 13.16 \end{bmatrix}$$

See the source code of the example in the `example_LLH_d.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution              Numerical solution
 1.0000000000000000e+00      1.0000000000038496e+00
 1.0000000000000000e+00      1.0000000000002858e+00
 1.0000000000000000e+00      1.0000000000043996e+00
 1.0000000000000000e+00      1.0000000000014913e+00
 1.0000000000000000e+00      9.9999999999847777e-01
 1.0000000000000000e+00      9.9999999999670042e-01
```

The conditionality number of the coefficient matrix is 5.0787e+05, the relative deviation of 2.88e-12 of the approximate solution from the exact one is quite justified.

A.5. Complex Hermitian positive-definite coefficient matrix with single precision

The system of equations

$$\begin{pmatrix} 5.01 & 0 & -2+i & 0 & 0 & 2.09 \\ 0 & 4.99 & 0 & -2.98 & -2+2i & 0 \\ -2-i & 0 & 4 & 0 & 0 & 3.19 \\ 0 & -2.98 & 0 & 5.03 & 0 & 0 \\ 0 & -2-2i & 0 & 0 & 3.99 & -2 \\ 2.09 & 0 & 3.19 & 0 & -2 & 8.98 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 2.1+12.55i \\ -9.99-9.94i \\ 6.19+28.14i \\ 2.05+14.16i \\ -6.01+15.95i \\ 24.26+63.54i \end{bmatrix}$$

See the source code of the example in the `example_LLH_c.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution                         Numerical solution
 1.0000000e+00 +  1.0000000e+00*i       1.0010201e+00 +  9.9902976e-01*i
 1.0000000e+00 +  2.0000000e+00*i       1.0015880e+00 +  2.0003891e+00*i
 1.0000000e+00 +  3.0000000e+00*i       1.0017089e+00 +  2.9991865e+00*i
 1.0000000e+00 +  4.0000000e+00*i       1.0009408e+00 +  4.0002303e+00*i
 1.0000000e+00 +  5.0000000e+00*i       1.0009670e+00 +  5.0015926e+00*i
 1.0000000e+00 +  6.0000000e+00*i       9.9880081e-01 +  6.0007300e+00*i
```

The conditionality number of the coefficient matrix is 88.23. The relative error of the obtained solution is 4.58-04, with the help of one iteration of refinement, it can be improved to 1.85e-06.

A.6. Complex Hermitian positive-definite coefficient matrix with double precision

The system of equations

$$\begin{pmatrix} 2 & 0 & 0.51i & 0 & 0 & 0 & 0.74 \\ 0 & 1 & 0 & 0 & 0.8 & 0 & 0 \\ -0.51i & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & -0.3i \\ 0 & 0.8 & 0 & 0 & 1 & 0.6 & 0 \\ 0 & 0 & 0 & 0 & 0.6 & 2 & 1+0.8i \\ 0.74 & 0 & 0 & 0.3i & 0 & 1-0.8i & 2 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 5.65+8.71i \\ 6+2i \\ 3.51+2.49i \\ 10.1-10.1i \\ 10.2-0.2i \\ 16.4+3.6i \\ 17.14+5.14i \end{bmatrix}$$

See the source code of the example in the `example_LLH_z.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution                                        Numerical solution
 1.0000000000000000e+00 +  1.0000000000000000e+00*i    9.9999999999754519e-01 +  1.0000000000010525e+00*i
 2.0000000000000000e+00 + -2.0000000000000000e+00*i    1.9999999999896669e+00 + -2.0000000000028564e+00*i
 3.0000000000000000e+00 +  3.0000000000000000e+00*i    2.9999999999994627e+00 +  2.9999999999987477e+00*i
 4.0000000000000000e+00 + -4.0000000000000000e+00*i    4.0000000000003704e+00 + -3.9999999999991331e+00*i
 5.0000000000000000e+00 +  5.0000000000000000e+00*i    5.0000000000129159e+00 +  5.0000000000035705e+00*i
 6.0000000000000000e+00 + -6.0000000000000000e+00*i    5.9999999999922489e+00 + -6.0000000000021423e+00*i
 7.0000000000000000e+00 +  7.0000000000000000e+00*i    7.0000000000057723e+00 +  6.9999999999975264e+00*i
```

The relative deviation of solutions 1.21e-12 is quite consistent with the value of the conditionality number 1.38e+05 of the coefficient matrix

A.7. Real symmetric coefficient matrix with single precision

The system of equations

$$\begin{pmatrix} 4 & 0 & -2 & 0 & 0 & 2 \\ 0 & 3.26 & 0 & -3 & -2 & 0 \\ -2 & 0 & 1.001 & 0 & 0 & 1 \\ 0 & -3 & 0 & 4 & 0 & 2 \\ 0 & -2 & 0 & 0 & 4 & -2 \\ 2 & 0 & 1 & 2 & -2 & 8 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 0 \\ -2.26 \\ 1.999 \\ -3 \\ 8 \\ -9 \end{bmatrix}$$

See the source code of the example in the `example_LDLH_s.c` file. Below is the output of this example:

The relative deviation of solutions 5.63e-05 is quite consistent with the value of the conditionality number 1.88e+03 of the coefficient matrix.

A.8. Real symmetric coefficient matrix with double precision

The system of equations

$$\begin{pmatrix} 4 & 0 & -2 & 0 & 0 & 2 \\ 0 & 3.26 & 0 & -3 & -2 & 0 \\ -2 & 0 & 1.00001 & 0 & 0 & 1 \\ 0 & -3 & 0 & 4 & 0 & 2 \\ 0 & -2 & 0 & 0 & 4 & -2 \\ 2 & 0 & 1 & 2 & -2 & 8 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 0 \\ -2.26 \\ -1.99999 \\ -3 \\ 8 \\ -9 \end{bmatrix}$$

See the source code of the example in the `example_LDLH_d.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution              Numerical solution
 1.0000000000000000e+00      1.0000000000291038e+00
-1.0000000000000000e+00     -9.9999999999998579e-01
 1.0000000000000000e+00      1.0000000000582077e+00
-1.0000000000000000e+00     -9.9999999999998923e-01
 1.0000000000000000e+00      1.0000000000000071e+00
-1.0000000000000000e+00     -1.0000000000000002e+00
```

The conditionality number of the coefficient matrix is 1.90+03. The relative deviation of the approximate solution is 2.66-11, with a single iteration of refinement it can be improved to 3.39e-15.

A.9. An attempt to use the Cholesky decomposition for a non-positive-definite matrix

In this example, the coefficients of the system of equations form the matrix bfwb62, all components of the exact solution vector are equal to one:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{62} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

The vector of the right-hand side is obtained by multiplying the coefficient matrix by the exact solution. An attempt to solve this system of equations using the Cholesky decomposition (`USP_fact=LLH`) causes error message 4 at the factorization stage. On the contrary, the value `USP_fact=LDLH` leads to success:

```
---- Trying to solve system with LLH
USPARS factorization failed.
Nonzero return code ier=4 was obtained from uspars_fact()
---- Solving system with LDLH
Results:
               name | n    | nnz(A) |    error C     |    error L2    |
./mtx/bfwb62.mtx |   62 |       202| 6.770540e-16 | 3.614540e-16 |
```

See the source code of the example in the `example_pos_def.c` file.

A.10. Complex symmetric coefficient matrix with single precision

The system of equations

$$\begin{pmatrix} 1 & 0 & 5i & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 8 & 0 & 0 \\ 5i & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.001 & 0 & 0 & -3i \\ 0 & 8 & 0 & 0 & 2 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0.1 & 1 & 1+i \\ 0 & 0 & 0 & -3i & 0 & 1+i & -1 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 1+15i \\ 44 \\ 9+5i \\ 0.004-21i \\ 26.6 \\ 13.5+7i \\ -1-6i \end{bmatrix}$$

See the source code of the example in the `example_LDLT_c.c` file. For better accuracy, one additional refinement iteration is used. Below is the output of this example:

```
The system was successfully solved.
       Exact solution                              Numerical solution
 1.0000000e+00 +  0.0000000e+00*i 1.0000000e+00 +  1.7029897e-07*i
 2.0000000e+00 +  0.0000000e+00*i 2.0000002e+00 +  7.6784645e-10*i
 3.0000000e+00 +  0.0000000e+00*i 3.0000000e+00 + -1.7029897e-08*i
 4.0000000e+00 +  0.0000000e+00*i 4.0000005e+00 + -2.1909364e-07*i
 5.0000000e+00 +  0.0000000e+00*i 5.0000000e+00 + -1.9196161e-10*i
 6.0000000e+00 +  0.0000000e+00*i 6.0000005e+00 + -5.7588494e-08*i
 7.0000000e+00 +  0.0000000e+00*i 7.0000000e+00 +  1.5073585e-11*i
```

The conditionality number of the coefficient matrix is 12.58. The relative deviation of 6.65e-8 even exceeds expectations.

A.11. Complex symmetric coefficient matrix with double precision

The system of equations

$$\begin{pmatrix} 1 & 0 & 5i & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 8 & 0 & 0 \\ 5i & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.001 & 0 & 0 & -3i \\ 0 & 8 & 0 & 0 & 2 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0.1 & 1 & 1+i \\ 0 & 0 & 0 & -3i & 0 & 1+i & -1 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 1+15i \\ 44 \\ 9+5i \\ 0.004-21i \\ 26.6 \\ 13.5+7i \\ -1-6i \end{bmatrix}$$

See the source code of the example in the `example_LDLT_z.c` file. Below is the output of this example:

```
The system was successfully solved.
```

```
Exact solution                                      Numerical solution
1.0000000000000000e+00 +  0.0000000000000000e+00*i  1.0000000000000000e+00 +  1.9032394707859828e-16*i
2.0000000000000000e+00 +  0.0000000000000000e+00*i  2.0000000000000000e+00 +  2.8931319859074138e-18*i
3.0000000000000000e+00 +  0.0000000000000000e+00*i  3.0000000000000004e+00 +  2.2204460492503131e-16*i
4.0000000000000000e+00 +  0.0000000000000000e+00*i  4.0000000000000000e+00 +  0.0000000000000000e+00*i
5.0000000000000000e+00 +  0.0000000000000000e+00*i  5.0000000000000000e+00 + -7.2328299647685346e-19*i
6.0000000000000000e+00 +  0.0000000000000000e+00*i  6.0000000000000000e+00 + -2.1698489894305601e-16*i
7.0000000000000000e+00 +  0.0000000000000000e+00*i  7.0000000000000000e+00 +  0.0000000000000000e+00*i
```

The conditionality number of the coefficient matrix is 12.58. The relative deviation of 4.85e-17 of the approximate solution from the exact one exceeds expectations.

A.12. Complex Hermitian coefficient matrix with single precision

The system of equations

$$\begin{pmatrix} 1 & 0 & 5i & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 8 & 0 & 0 \\ -5i & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.001 & 0 & 0 & -3i \\ 0 & 8 & 0 & 0 & 2 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0.1 & 1 & 1+i \\ 0 & 0 & 0 & 3i & 0 & 1-i & -1 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 1+15i \\ 44 \\ 9+5i \\ 0.004-21i \\ 26.6 \\ 13.5+7i \\ -1-6i \end{bmatrix}$$

See the source code of the example in the `example_LDLH_c.c` file. For better accuracy, one additional refinement iteration is used. Below is the output of this example:

```
The system was successfully solved.
       Exact solution              Numerical solution
 1.0000000e+00 +  0.0000000e+00*i 1.0000000e+00 + -2.1674416e-07*i
 2.0000000e+00 +  0.0000000e+00*i 2.0000002e+00 + -2.3225937e-09*i
 3.0000000e+00 +  0.0000000e+00*i 3.0000000e+00 +  5.4186060e-09*i
 4.0000000e+00 +  0.0000000e+00*i 4.0000000e+00 +  1.7601997e-07*i
 5.0000000e+00 +  0.0000000e+00*i 5.0000000e+00 +  5.8064842e-10*i
 6.0000000e+00 +  0.0000000e+00*i 6.0000000e+00 +  1.7419441e-07*i
 7.0000000e+00 +  0.0000000e+00*i 7.0000000e+00 + -7.0595190e-13*i
```

The conditionality number of the coefficient matrix is 12.63. The relative deviation of 3.26e-08 of the approximate solution from the exact one is within reasonable limits.

A.13. Complex Hermitian coefficient matrix with double precision

The system of equations

$$\begin{pmatrix} 1 & 0 & 5i & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 8 & 0 & 0 \\ -5i & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.001 & 0 & 0 & -3i \\ 0 & 8 & 0 & 0 & 2 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0.1 & 1 & 1+i \\ 0 & 0 & 0 & 3i & 0 & 1-i & -1 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 1+15i \\ 44 \\ 9+5i \\ 0.004-21i \\ 26.6 \\ 13.5+7i \\ -1-6i \end{bmatrix}$$

See the source code of the example in the `example_LDLH_z.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution                                    Numerical solution
 1.0000000000000000e+00+0.0000000000000000e+00*i 1.0000000000000000e+00 -2.4223047810003419e-16*i
 2.0000000000000000e+00+0.0000000000000000e+00*i 2.0000000000000000e+00 +1.1835803775002936e-17*i
 3.0000000000000000e+00+0.0000000000000000e+00*i 3.0000000000000004e+00 -2.2204460492503131e-16*i
 4.0000000000000000e+00+0.0000000000000000e+00*i 4.0000000000000000e+00 +0.0000000000000000e+00*i
 5.0000000000000000e+00+0.0000000000000000e+00*i 5.0000000000000000e+00 -2.9589509437507341e-18*i
 6.0000000000000000e+00+0.0000000000000000e+00*i 6.0000000000000000e+00 -8.8768528312522019e-16*i
 7.0000000000000000e+00+0.0000000000000000e+00*i 7.0000000000000000e+00 -1.0842021724855044e-19*i
```

The conditionality number of the coefficient matrix is 12.63. The relative deviation of 3.75e-17 of the approximate solution from the exact one is within reasonable limits.

A.14. Real coefficient matrix with single precision

The system of equations

$$\begin{pmatrix} 1 & 0 & 5 & 0 & 0 & 0 & 10 \\ 0 & 2 & 0 & 8 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -3 \\ 0 & 1.01 & 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & -1 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 86 \\ 40 \\ 10 \\ -17 \\ 16 \\ 12 \\ 3 \end{bmatrix}.$$

See the source code of the example in the `example_LU_s.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution     Numerical solution
 1.0000000e+00      1.0000591e+00
 0.0000000e+00      1.7764522e-05
 3.0000000e+00      2.9999802e+00
 4.0000000e+00      4.0000134e+00
 5.0000000e+00      4.9999957e+00
 6.0000000e+00      5.9999909e+00
 7.0000000e+00      7.0000043e+00
```

The conditionality number of the coefficient matrix is 73.89. The relative deviation of 5.75e-06 of the approximate solution from the exact one is within reasonable limits.

A.15. Real coefficient matrix with double precision

The system of equations

$$\begin{pmatrix} 0 & 0 & 105 & 0 & 0 & 0 & 107 \\ 0 & 1 & 0 & 0 & 18 & 0 & 0 \\ 1 & 0 & 1 & 0 & 111 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & -103 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & -1 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 212 \\ 17 \\ 113 \\ -105 \\ 0 \\ 1 \\ -3 \end{bmatrix}.$$

See the source code of the example in the `example_LU_d.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution             Numerical solution
 1.0000000000000000e+00     1.0000000000000637e+00
-1.0000000000000000e+00    -9.9999999999999045e-01
 1.0000000000000000e+00     1.0000000000000000e+00
-1.0000000000000000e+00    -1.0000000000000004e+00
 1.0000000000000000e+00     9.9999999999999944e-01
-1.0000000000000000e+00    -9.9999999999999956e-01
 1.0000000000000000e+00     1.0000000000000000e+00
```

The conditionality number of the coefficient matrix is 3.06e+04. The relative deviation of 2.44e-14 of the approximate solution from the exact one is within reasonable limits.

A.16. Complex coefficient matrix with single precision

The system of equations

$$\begin{pmatrix} 0 & 0 & 105 & 0 & 0 & 0 & 300-4i \\ 0 & 1+i & 0 & 0 & 1+8i & 0 & 0 \\ 1 & 0 & 1 & 0 & 111-20i & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & -103 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & i & 0 & 1 & -1+10i \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 300+101i \\ 7i \\ 113-20i \\ -105 \\ 0 \\ 1 \\ -2+9i \end{bmatrix}.$$

See the source code of the example in the `example_LU_c.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution                              Numerical solution
 1.0000000e+00 +  0.0000000e+00*i     9.9996775e-01 +  5.3781223e-05*i
-1.0000000e+00 +  0.0000000e+00*i    -1.0000030e+00 +  7.5085632e-07*i
 1.0000000e+00 +  0.0000000e+00*i     1.0000001e+00 +  0.0000000e+00*i
-1.0000000e+00 +  0.0000000e+00*i    -1.0000001e+00 + -4.4926441e-07*i
 1.0000000e+00 +  0.0000000e+00*i     1.0000004e+00 + -4.1516284e-07*i
-1.0000000e+00 +  0.0000000e+00*i    -1.0000005e+00 +  4.1516284e-07*i
 1.0000000e+00 +  0.0000000e+00*i     1.0000000e+00 +  0.0000000e+00*i
```

The conditionality number of the coefficient matrix is 9.45e+05. The relative deviation of 2.37e-05 of the approximate solution from the exact one is within reasonable limits.

A.17. Complex coefficient matrix with double precision

The system of equations

$$\begin{pmatrix} 104 & 0 & 105i & 0 & 0 & 0 & 30-4i \\ 0 & 1+i & 0 & 0 & 1+8i & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 97 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0.1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0.96i & 0 & 111-20i & 0 & 0 \\ 0 & 0 & 0 & i & 0 & 1 & -3i \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 134+101i \\ 2+7i \\ 97-2i \\ 1.1-i \\ 1-i \\ 112-19.04i \\ 1-4i \end{bmatrix}.$$

See the source code of the example in the `example_LU_z.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution                                             Numerical solution
 1.0000000000000000e+00 +  0.0000000000000000e+00*i    9.9999999999737788e-01 + -1.3273233099706723e-12*i
-0.0000000000000000e+00 + -1.0000000000000000e+00*i   -7.2840132591628681e-16 + -1.0000000000000058e+00*i
 1.0000000000000000e+00 +  0.0000000000000000e+00*i    1.0000000000013194e+00 + -2.6025693228422387e-12*i
-0.0000000000000000e+00 + -1.0000000000000000e+00*i    7.2980095736518887e-16 + -1.0000000000000009e+00*i
 1.0000000000000000e+00 +  0.0000000000000000e+00*i    1.0000000000000009e+00 +  7.2984424493577328e-16*i
-0.0000000000000000e+00 + -1.0000000000000000e+00*i   -8.8817841970012523e-16 + -1.0000000000000007e+00*i
 1.0000000000000000e+00 +  0.0000000000000000e+00*i    1.0000000000000000e+00 +  1.8503717077085926e-17*i
```

The conditionality number of the coefficient matrix is 1.098e+08. The relative deviation of 1.57e-12 of the approximate solution from the exact one is within reasonable limits.

A.18. Real symmetric positive-definite coefficient matrix with single precision, two right-hand sides

The system of equations

$$\begin{pmatrix} 4 & 0 & -2 & 0 & 0 & 2 \\ 0 & 5 & 0 & -3 & -2 & 0 \\ -2 & 0 & 4 & 0 & 0 & 3 \\ 0 & -3 & 0 & 5 & 0 & 2 \\ 0 & -2 & 0 & 0 & 3.96 & -2 \\ 2 & 0 & 3 & 2 & -2 & 8.16 \end{pmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \\ x_{51} & x_{52} \\ x_{61} & x_{62} \end{bmatrix} = \begin{bmatrix} 4 & 0 \\ 0 & -4 \\ 5 & -1 \\ 4 & -4 \\ -0.04 & 7.96 \\ 13.16 & -7.16 \end{bmatrix}.$$

See the source code of the example in the `example2rhs_LLH_s.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution    Numerical solution

 rhs 0:
 1.0000000e+00      9.9828684e-01
 1.0000000e+00      9.9987251e-01
 1.0000000e+00      9.9804211e-01
 1.0000000e+00      9.9933606e-01
 1.0000000e+00      1.0006772e+00
 1.0000000e+00      1.0014684e+00

 rhs 1:
 1.0000000e+00      1.0017132e+00
-1.0000000e+00     -9.9987239e-01
 1.0000000e+00      1.0019579e+00
-1.0000000e+00     -9.9933606e-01
 1.0000000e+00      9.9932289e-01
-1.0000000e+00     -1.0014684e+00
```

The conditionality number of the coefficient matrix is 5.0787e+05. The relative deviation of 6.92e-04 of the approximate solution from the exact one is is quite justified.

A.19. Real coefficient matrix with single precision, two right-hand sides

The system of equations

$$\begin{pmatrix} 1 & 0 & 5 & 0 & 0 & 0 & 10 \\ 0 & 2 & 0 & 8 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -3 \\ 0 & 1.01 & 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & -1 \end{pmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \\ x_{51} & x_{52} \\ x_{61} & x_{62} \\ x_{71} & x_{72} \end{bmatrix} = \begin{bmatrix} 86 & 16 \\ 40 & 10 \\ 10 & 4 \\ -17 & -2 \\ 16 & 4.01 \\ 12 & 2 \\ 3 & 1 \end{bmatrix}.$$

See the source code of the example in the `example2rhs_LU_s.c file.` Below is the output of this example:

```
 The system was successfully solved.
 Exact solution    Numerical solution

 rhs 0:
 1.0000000e+00      1.0000591e+00
 0.0000000e+00      1.7764522e-05
 3.0000000e+00      2.9999802e+00
 4.0000000e+00      4.0000134e+00
 5.0000000e+00      4.9999957e+00
 6.0000000e+00      5.9999909e+00
 7.0000000e+00      7.0000043e+00

 rhs 1:
 1.0000000e+00      1.0000657e+00
 1.0000000e+00      1.0000175e+00
 1.0000000e+00      9.9997813e-01
 1.0000000e+00      1.0000131e+00
```

```
 1.0000000e+00      9.9999565e-01
 1.0000000e+00      9.9999130e-01
 1.0000000e+00      1.0000044e+00T
```

The conditionality number of the coefficient matrix is 73.89 The relative deviation of 2.81e-05 of the approximate solution from the exact one is is quite justified.

A.20. Iterative refinement

The system of equations is formed from the matrix rw496 with the vector of the exact solution consisting of units and the corresponding right-hand side:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{496} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

See the source code of the example in the `example_iter_ref.c` file. The matrix is not well conditioned (cond(A)=1.99e+10), which affects the accuracy of the solution. Enabling iterative refinement can improve the accuracy of the result, as demonstrated by the calculation results in this example:

```
  Results w/o iter ref:
             name |    n    |  nnz(A)   |     error C     |     error L2     |
 ./mtx/rw496.mtx |   496 |     1859 |   6.065465e+02 |   3.089256e+01 |

  Results with iter ref:
             name |    n    |  nnz(A)   |     error C     |     error L2     |
 ./mtx/rw496.mtx |   496 |     1859 |   6.858562e-07 |   3.441423e-08 |
```

These tables in columns `error C` and `error L2` contain relative errors

$\frac{\|x-x_{calc}\|_{\max}}{\|x\|_{\max}}$ *and* $\frac{\|x-x_{calc}\|_2}{\|x\|_2}$, respectively, in norms

$$\|x\|_{\max} = \max_{1 \le i \le 496} |x_i|, \quad \|x\|_2 = \sqrt{\sum_{i=1}^{496} |x_i|^2}\,.$$

A.21. Diagonal boosting to 'work around' Zero Pivot problem

This example demonstrates the effect of enabling diagonal boosting (see C.3.1) when error code 4 occurs at the factorization stage. The system of equations is formed (see also A.22) from the matrix impcol_c[1] with the vector of the exact solution consisting of units and the corresponding right-hand side.

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{137} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

and the corresponding right-hand side. See the source code of the example in the `example_boosting.c` file. Below is the output of this example:

```
---- Trying to solve system with boosting turned off.
USPARS factorization failed.
 Nonzero return code 4 was obtained from uspars_fact()

---- Solving system with boosting turned on.
  Results:
             name |    n    |  nnz(A)   |     error C     |     error L2     |
```

[1] Note that the impcol_c matrix is conditioned quite well. Its conditionality number is 1.77e+04

```
./mtx/impcol_c.mtx |    137 |       411 |    6.661338e-16 |    9.485275e-17 |
```

A.22. Global pivoting to 'Bypass' Zero Pivot Problem

This example demonstrates the effect of enabling global pivoting (see C.3.2) when error code 4 occurs at the factorization stage. The system of equations is formed (see also A.21) from the matrix impcol_c with the vector of the exact solution consisting of units and the corresponding right-hand side:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{137} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

See the source code of the example in the `example_gp.c` file. Below is the output of this example:

```
---- Trying to solve system with global pivot turned off.
USPARS factorization failed.
Nonzero return code ier=4 was obtained from uspars_fact()

---- Global pivot turned on.
Results:
              name | n   | nnz(A)   |   error C     |  error L2     |
./mtx/impcol_c.mtx | 137 |     411 | 4.440892e-16 | 7.705874e-17 |
```

A.23. Using Out-Of-Core mode

This example demonstrates the use of OOC (see C.4) in two cases. In the first case, due to the small matrix size and the appropriate parameter value selection, intermediate values are not offloaded to disk, and the system is solved in IC mode. In the second case, intermediate values are offloaded to disk during factorization and are read back later during the solution stage. The system of equations is formed (see also A.21) from the matrix impcol_c with the vector of the exact solution consisting of units and the corresponding right-hand side:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{137} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

See the source code of the example in the `example_OOC.c` file. Below is the output of this example:

```
---- USP_OOC_MEM = 10000
Results w/o force OOC:
              name |    n    |  nnz(A)   |      error C     |     error L2     |
./mtx/impcol_c.mtx |    137  |      411  |   4.440892e-16   |   6.773842e-17   |

---- USP_OOC_MEM = -10000
Results w force OOC:
              name |    n    |  nnz(A)   |      error C     |     error L2     |
./mtx/impcol_c.mtx |    137  |      411  |   4.440892e-16   |   6.773842e-17   |
```

A.24. Using long long int (64-bit) interfaces

This example demonstrates the use of the long long int (see 1.1.1, 2.2) interfaces. It presents three cases:

- `ia` and `ja` arrays of type `int`, `uspars_init32` function is called
- `ia` array is of type `int`, `ja` array is of type `long long int`, `uspars_init64` function is called
- `ia` and `ja` arrays of type `long long int`, `uspars_init64` function is called

The following system of equations is solved in all cases:

$$\begin{pmatrix} 0 & 0 & 105 & 0 & 0 & 0 & 107 \\ 0 & 1 & 0 & 0 & 18 & 0 & 0 \\ 1 & 0 & 1 & 0 & 111 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & -103 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & -1 \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 212 \\ 17 \\ 113 \\ -105 \\ 0 \\ 1 \\ -3 \end{bmatrix}.$$

See the source code of the example in the `example_3264.c` file. Below is the output of this example:

```
The system was successfully solved.
 Exact solution              Numerical solution (32)
 1.0000000000000000e+00    1.000000000000637e+00
-1.0000000000000000e+00   -9.999999999999045e-01
 1.0000000000000000e+00    1.000000000000000e+00
-1.0000000000000000e+00   -1.000000000000004e+00
 1.0000000000000000e+00    9.999999999999944e-01
-1.0000000000000000e+00   -9.999999999999956e-01
 1.0000000000000000e+00    1.000000000000000e+00
____________

The system was successfully solved.
 Exact solution              Numerical solution (3264)
 1.0000000000000000e+00    1.000000000000637e+00
-1.0000000000000000e+00   -9.999999999999045e-01
 1.0000000000000000e+00    1.000000000000000e+00
-1.0000000000000000e+00   -1.000000000000004e+00
 1.0000000000000000e+00    9.999999999999944e-01
-1.0000000000000000e+00   -9.999999999999956e-01
 1.0000000000000000e+00    1.000000000000000e+00
____________

The system was successfully solved.
 Exact solution              Numerical solution (64)
 1.0000000000000000e+00    1.000000000000637e+00
-1.0000000000000000e+00   -9.999999999999045e-01
 1.0000000000000000e+00    1.000000000000000e+00
-1.0000000000000000e+00   -1.000000000000004e+00
 1.0000000000000000e+00    9.999999999999944e-01
-1.0000000000000000e+00   -9.999999999999956e-01
 1.0000000000000000e+00    1.000000000000000e+00
____________
```

B. Linear algebraic and algorithmic foundations

B.1. Gaussian elimination method

The system of equations is written as

$$Ax = f, \qquad \text{(B.1)}$$

where vectors x and f each have N components, matrix A has dimensions $N \times N$. From an algorithmic point of view, it is advisable to identify several classes to which the coefficient matrix may belong.

B.1.1. Real symmetric or complex Hermitian positive definite matrices

In the real case, the class of matrices is distinguished by the condition

$$A = A^T > 0, \qquad (B.2)$$

in a complex condition looks like this:

$$A = A^H > 0. \qquad \text{(B.3)}$$

Here A^T denotes the transposition of the matrix A, $A^H = \overline{A}^T$ - Hermitian conjugation. For this class of matrices, the Cholesky decomposition can be used

$$A = \begin{cases} LL^T & \text{real symmetric case} \\ LL^H & \text{complex Hermitian case} \end{cases} \qquad (B.4)$$

Note that here L – is lower triangular, and L^T and L^H automatically turn out to be upper triangular. In the case of a sparse matrix A the lower triangular matrix L also turns out to be sparse, but usually with a significantly larger number of non-zero elements (*filling effect*).

The number of non-zero elements in L is often significantly greater than the number of non-zero elements in the original matrix. This number depends on the order of rows (and columns) selected for writing the original matrix. To reduce the number of non-zero elements in L, thereby reducing the memory load and reducing the required number of floating-point operations, special permutation algorithms are used, which, at relatively low additional costs, can achieve significant savings. This means that instead of the original matrix, the product PAP^T, where P is the permutation matrix (see 1.4).

Thus, the algorithm for solving a system of linear equations (B.1) with a coefficient matrix satisfying the conditions (B.2) or (B.3), consists of the following steps:

1. Symmetric permutation of rows and columns of a matrix

$$A_1 = PAP^T \qquad \text{(B.5)}$$

and the right-hand side vector component

$$g = Pf; \qquad \text{(B.6)}$$

2. Triangular matrix factorization

$$A = \begin{cases} LL^T & \text{real symmetric case} \\ LL^H & \text{complex Hermitian case} \end{cases} \qquad (B.7)$$

3. Solving systems of equations with triangular coefficient matrices

$$Ly = g; \quad \begin{matrix} L^T z = y & \text{real symmetric case} \\ L^H z = y & \text{complex Hermitian case} \end{matrix} \qquad \text{(B.8)}$$

4. Permutation of the components of the vector of unknowns

$$x = P^T z. \qquad \text{(B.9)}$$

It is assumed that the conditions of positive certainty (B.2), (B.3) are sufficient for the numerical stability of the algorithm.

B.1.2. Symmetric (real or complex) and complex Hermitian coefficient matrices}

The class of matrices is distinguished by the following conditions

$$A = \begin{cases} A^T & \text{symmetric case} \\ A^H & \text{complex Hermitian case} \end{cases} \qquad (\mathrm{B}.10)$$

This class completely contains the positive definite matrices highlighted in the previous subsection. Thus, everything described below in this subsection also applies to the previous one.

The corresponding decomposition is called *LDLT decomposition* (*LDLH decomposition* in the Hermitian case)

$$PAP^T = \begin{cases} LDL^T & \text{symmetric case} \\ LDL^H & \text{complex Hermitian case} \end{cases} \qquad (\mathrm{B}.11)$$

In these formulas, P denotes the permutation matrix used to increase the stability of the algorithm, L is a lower triangular matrix with units on the diagonal, D denotes a block-diagonal matrix with (symmetric or Hermitian) small blocks on the diagonal[1].

Everything said in the previous subsection regarding the structure of the triangular multiplier also applies to the case of the LDLT decomposition. Accordingly, the algorithm for solving the system of linear equations (B.1) with a coefficient matrix satisfying conditions (B.10) is divided into steps:

1. Symmetric permutation of rows and columns (B.5) of the matrix and vector components of the right-hand side (B.6).
2. Triangular matrix factorization (cf. (B.11))
$$A_1 = \begin{cases} LDL^T & \text{symmetric case} \\ LDL^H & \text{complex Hermitian case} \end{cases} \qquad (\mathrm{B}.12)$$
3. Solving systems of equations with coefficient matrices obtained as a result of factorization
$$\begin{aligned} Ly &= g; \\ Dw &= y; \\ L^T z &= w. \end{aligned} \qquad (\mathrm{B}.13)$$
In the Hermitian case, the last system of equations will take the form
$$\begin{aligned} Ly &= g; \\ Dw &= y; \\ L^H z &= w. \end{aligned} \qquad (\mathrm{B}.14)$$
4. Permutation of components of the vector of unknowns (B.9)

B.1.3. Coefficient matrices of general form

Unlike the previous subsections, no restrictions are imposed on the coefficient matrix A of the system of linear equations (B.1).

The triangular decomposition of the coefficient matrix has the form

$$A = LU \qquad (\mathrm{B}.15)$$

where L is a lower triangular matrix with units on the diagonal, U is an upper triangular matrix.

[1] The corresponding functions in LAPACK are named `?SYTRF, CHETRF, ZHETRF,` where the sign is `?` stands for one of the letters `S, D, C, Z.`

As in the previous subsection, permutations of rows and columns can be applied to the sparse coefficient matrix, designed to reduce the number of non-zero elements in triangular factors and increase the robustness of the calculations to rounding errors. Only the permutations on the left and on the right are, generally speaking, different. The algorithm for solving a system of linear equations turns out to be ideologically similar to those described in the previous subsections.

1. Permutation of rows and columns of the coefficient matrix and vector components of the right-hand side (B.6)

$$\begin{aligned} A_1 &= P_1 A P_2; \\ g &= P_1 f. \end{aligned} \tag{B.16}$$

2. Triangular matrix factorization (cf. (B.15))

$$A_1 = LU \tag{B.17}$$

3. Solving systems of equations with triangular coefficient matrices

$$\begin{aligned} Ly &= g; \\ Uz &= y. \end{aligned} \tag{B.18}$$

4. Permutation of components of the vector of unknowns

$$x = P_2 z. \tag{B.19}$$

B.1.4. Structure of triangular factors

Y- The triangular factors into which the original matrix is decomposed also have a sparse structure, but with one significant caveat. The number of non-zero elements in triangular factors usually exceeds the number of non-zero elements in the original matrix. This is manifested by the so-called *fill-in effect* (*fill-in*).. To illustrate, consider the symmetric positive definite matrix

$$A = \begin{pmatrix} 7 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Calculating its Cholesky decomposition $A = LL^T$, we obtain

$$L = \begin{pmatrix} 2.6458 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.3780 & 0.9258 & 0 & 0 & 0 & 0 & 0 \\ 0.3780 & -0.1543 & 0.9129 & 0 & 0 & 0 & 0 \\ 0.3780 & -0.1543 & -0.1826 & 0.8944 & 0 & 0 & 0 \\ 0.3780 & -0.1543 & -0.1826 & -0.2236 & 0.8660 & 0 & 0 \\ 0.3780 & -0.1543 & -0.1826 & -0.2236 & -0.2887 & 0.8165 & 0 \\ 0.3780 & -0.1543 & -0.1826 & -0.2236 & -0.2887 & -04082 & 0.7071 \end{pmatrix}$$

The Cholesky decomposition of the same matrix, but with symmetrically rearranged rows and columns, has the form

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

A comparison of the two decompositions illustrates the statement that the number of nonzero elements in the triangular factors of the Gauss method depends on the selected row and column order of the matrix.

The filling effect affects the amount of RAM required. It is easy to understand that the number of arithmetic operations required to perform triangular factorization also depends on the filling. The task of finding the optimal (in the sense of minimizing the filling effect) ordering of rows and columns of the NP matrix is difficult, and therefore has no practical significance. However, there are algorithms ([4], [5], [6]), as a result of which the filling effect is significantly reduced.

In the USPARS package, the Atlant component (see B.4) has been developed to find matrix permutations in order to reduce the occupancy of triangular factors.

B.2. Balancing equation coefficients

In algorithms for solving systems of linear equations, to improve the stability of calculations to rounding errors, the so-called *scaling of rows and columns of the matrix* can be used instead of a system of equations

$$Ax = f$$

the system is considered

$$A_1 y = D_l f$$

in which the coefficient matrix is obtained from the original matrix by multiplication

$$A_1 = D_l A D_r$$

on the left and right to specially selected diagonal scaling matrices D_l and D_r. The elements of the diagonals of the scaling matrices are chosen so that in the rows and columns of the matrix A_1 the maximum elements are of the order of one. This technique leads to a decrease in the conditionality number of the coefficient matrix, and, accordingly, reduces the error in the numerical solution of a system of linear equations.

B.3. Matching

In order to increase the reliability of the numerical factorization of the matrix in the system

$Ax = f$,
equations can be reordered to improve the values of the diagonal elements.

For this purpose a permutation P_m of the rows of the original matrix A is applied, as a result of which the elements of the matrix with larger absolute values occurs on the main diagonal; for the elements of the right-hand side f, the same permutation is applied:

$$A_1 = P_m A, \;\; f_1 = P_m f.$$

The USPARS package implements several algorithms:

1) An algorithm for maximizing the minimum diagonal element (see [8])
2) An algorithm for maximizing the product of diagonal elements with the ability to scale the matrix (see [8])

B.4. Atlant

The USPARS package includes the Atlant component for reordering sparse matrices in order to reduce the fill-in reducing reordering of factors formed during matrix decomposition. The component uses a combined algorithm based on the multi-level method of nested sections [9] and an effective author's method for reordering small- sized subgraphs based on the minimum degree algorithm [5] as the main reordering method.

B.5. Bunch-Kaufman factorization

To improve numerical stability when solving systems of linear equations with symmetric matrix, Bunch-Kaufman factorization can be used. This method is a modification of LDL^T factorization that allows the use of both (1×1)-block and (2×2)-block updates to compute the factors. Unlike standard LDL^T factorization, where each update step involves changing one row and column of the matrix, Bunch-Kaufman factorization more efficiently handles ill-conditioned matrices by adaptively choosing between (1×1)-block and (2×2)-block updates.

The factorization process splits the original matrix (A) into a product of three matrices: $(A = LDL^T)$, where (L) is a lower triangular matrix with 1s on the diagonal, and (D) is a diagonal or block-diagonal matrix (with blocks of size (1×1) or (2×2)), providing improved numerical stability. This approach allows us to minimize the impact of rounding and accumulation of numerical errors, which is critical for solving ill-conditioned systems of linear equations.

C. Practical recommendations for using the USPARS package

C.1. Using multithreading

The functions of the USPARS package are parallelized for multicore systems using OpenMP tools. Starting with USPARS 2.3.1, the preferred way to set the number of OpenMP threads used inside USPARS is `options[USP_OMP_THREADS]` (see 2.2).
For example, if a specific solver run should use 16 threads, set the option before calling `uspars_init()`:

```
int options[USP_OPTIONS];

for (int i = 0; i < USP_OPTIONS; i++) options[i] = -1;

options[USP_OMP_THREADS] = 16;
```

The default value of `options[USP_OMP_THREADS]` is omp_get_max_threads() at the moment of the uspars_init() call. The `OMP_NUM_THREADS` environment variable and the omp_set_num_threads() function remain standard OpenMP tools and affect the value returned by omp_get_max_threads(). However, `options[USP_OMP_THREADS]` makes it possible to limit the number of threads for a specific USPARS run without changing the global OpenMP setting of the process.

You can set the upper bound for the number of OpenMP threads by assigning the `OMP_NUM_THREADS` environment variable a value equal to the required number of threads. In a Linux environment, this is done with the command

```
export OMP_NUM_THREADS=16
```

Accordingly, in the Windows environment the command looks like this

```
set OMP_NUM_THREADS=16
```

If one application runs several independent USPARS instances at the same time, it is recommended to distribute available resources between them using `options[USP_OMP_THREADS]`.

C.2. Iterative refinement

The third on the list

- rearrangement,
- factorization,
- solving systems with triangular coefficient matrices

phase of the USPARS package algorithm used to solve a system of linear equations

$$Ax = f,$$

usually takes a relatively small fraction of computational time.

The use of this circumstance underlies the use of iterative refinement to improve the accuracy of solving a system of linear equations. An additional computational process is organized, in which the solution is obtained in several similar steps. At each step of the process, the triangular factors of the original matrix obtained as a result of factorization are used.

C.2.1. Simple iterations

In the method of simple iterations, the residual from the refined solution obtained at the previous step serves as the vector of the right-hand side. The initial matrix is involved in the residual calculations, and, if possible, the calculations are carried out in a high-precision format. Using the described technique in many cases allows you to get a solution with better accuracy than without using it.

Under certain conditions, the vector $x^{(k)}$ obtained at the k-th step of the iterative process provides a solution with the best residual

$$\|Ax^{(k)} - f\| < \|Ax^{(k-1)} - f\|.$$

The exit from the iterative refinement process occurs when the limit of the number of iterations is reached

$$k \leq k_{\max} \tag{C.1}$$

or achieving the criterion of smallness of the relative norm of the residual

$$\frac{\|Ax^{(k)} - f\|}{\|f\|} < \alpha. \tag{C.2}$$

Parameters α and $k_{\max}$α are set by default or set by the user (see section 2.2). If the residual in the solution process reaches the value inf, the iterative refinement process is declared to have failed and the `uspars_solve()` function (see section 2.4) returns code 7.

C.2.2. BiCGStab

Biconjugate Gradient Stabilized (BiCGStab) is designed for solving asymmetric systems of linear equations, but it can be applied to any other type. This method is particularly useful for solving large sparse systems where a direct solver may be ineffective.

1. Calculate the residual of the USPARS solution and determine the stopping criteria based on the relative norm of the residual.

2. Apply BiCGStab iterations to refine the solution:

• Calculate search directions;

• Solve auxiliary systems of linear equations using triangular factors of the original matrix;

• Update the solution approximation and the residual vector.

Similar to simple iterations, the process terminates either when the maximum number of iterations is reached or when the relative norm of the residual becomes sufficiently small.

This method can be used as a tool for achieving higher accuracy in ill-conditioned systems, where it demonstrates superior convergence compared to simple iterations.

C.3. Techniques for 'bypassing' the Zero Pivot

During triangular factorization of a matrix, the diagonal element may turn out to be zero. The process cannot be continued because the diagonal elements are used as denominators in the factorization process. At the same time, the appearance of a zero element is not necessarily associated with the degeneracy of the original matrix - it can be well-conditioned.

To overcome this difficulty, the so-called *selection of the leading element in the column (pivoting)* is used, a permutation of rows, as a result of which a non-zero element from the same column located below the diagonal takes the place of the diagonal element. After this, factorization can be continued until the next zero element appears on the diagonal. Then the string permutation is required again. As a result, the matrix turns out to be factorized, represented as a product of two triangular matrices and a permutation matrix.

If at some step it is not possible to find a non-zero leading element in the column below the zero diagonal element, then this will mean that the original matrix is degenerate with all the ensuing consequences for the corresponding system of linear equations.

If calculations are performed on a computer, arithmetic operations on numbers are subject to rounding, and the reasoning above must be modified. Firstly, even if zeros do not appear on the diagonal during the factorization process, division by small denominators can lead to large values up to `overflow`. Secondly, there is an accumulation of rounding errors, which can distort the result. Thus, the selection of the leading element is necessary, but it must be modified.

It is clear that as the leading element, you just need to take the maximum modulo element, and always do this. There are two strategies for choosing the maximum modulo element:

A. Selecting the maximum in the column (Partial pivoting). When considering the diagonal element $\tilde{A}_{ii}$, to find the leading element, you should find

$$\max_{i \le k \le N} |\tilde{A}_{ki}|.$$

The found maximum element must be rearranged into place $\tilde{A}_{ii}$. To do this, you need to rearrange two lines.

B. Selecting the maximum on the diagonal (Diagonal pivoting). This method allows you to select the leading element without breaking the symmetry of the matrix, if any. The problem to be solved

$$\max_{i \le k \le N} |\tilde{A}_{kk}|$$

Here, the maximum element is searched on the diagonal. The found element is rearranged into the place of $\tilde{A}_{ii}$ by rearranging rows and columns.

C. Selection across the entire matrix (Full pivoting). Here the solved task of searching for the leading element has the form

$$\max_{\substack{i \le k \le N \\ i \le j \le N}} |\tilde{A}_{kj}|$$

Here, in order to put the found maximum element in the place of $\tilde{A}_{ii}$, you will need to rearrange the rows and columns.

In these formulas, the tilde in the notation is used to indicate the current values of the elements of the array containing the matrix elements in order to distinguish them from the elements of the original array (all calculations are performed 'on the spot').

Theorems on the stability of the solution are proved under the condition that the choice of the leading element is made over the entire matrix. There are no theorems that guarantee stability when choosing by column. Moreover, there are known counterexamples showing that when using column leader selection, computational instability can occur. However, for quite obvious reasons, the use of the Partial pivoting strategy finds practical application. At the same time, the authors refer to the fact that the use of this strategy usually almost always leads to sustainable results.

All previous arguments do not take into account the structure of non-zero elements in the matrix and therefore refer to dense matrices. Before factorizing a sparse matrix, special permutations of rows and columns are used to minimize memory consumption. In this case, a certain structure of non-zero elements appear in the matrix, which should not be violated. As a result, permutations of rows and columns are allowed, but only within certain limits. After rearranging the rows and columns to reduce the filling effect, square blocks are allocated on the diagonal in the matrix. Permutations carried out in order to select the leading element should not violate this block structure. This can be illustrated by the following formula

$$\begin{pmatrix} A_{11} & \times & \cdots & \times & \times & \times & \cdots \\ \times & A_{22} & \cdots & \times & \times & \times & \cdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \times & \times & \cdots & A_{i-1,i-1} & \times & \times & \cdots \\ \times & \times & \cdots & \times & \boxed{A_{i,i}} & \times & \cdots \\ \times & \times & \cdots & \times & \times & A_{i+1,i+1} & \cdots \\ \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Square diagonal blocks $A_{i,i}$ can have different orders and are treated as dense. Generally speaking, rectangular blocks are marked with crosses, which can be purely zero or have non-zero elements.

Factorization of the entire matrix includes factorization of diagonal blocks and inversion of the resulting triangular factors. In other words, the diagonal block must be non-degenerate[1]. at the time of factorization. During the factorization of the diagonal block $A_{i,i}$, permutations of rows and columns intersecting with the block $A_{i,i}$ are allowed.

Thus, when performing LU factorization in the USPARS package, the main element is selected according to the Full pivoting scheme with respect to the diagonal blocks. We call this strategy Local Full Pivoting.

The process of factorizing a dense $m \times m$ diagonal block is illustrated by the formula.

$$\begin{pmatrix} \times & \times & \times & \cdots & \times \\ \times & a_{kk} & a_{k,k+1} & \cdots & a_{km} \\ \times & a_{k+1,k} & a_{k+1,k+1} & \cdots & a_{k+1,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \times & a_{mk} & a_{m,k+1} & \cdots & a_{mm} \end{pmatrix} \qquad (C.3)$$

Here, the crosses indicate the already processed rows and columns, the processing of the diagonal element a_{kk} is next, and the remaining non-factorized subblock of size $(m-k+1) \times (m-k+1)$ is highlighted. Due to the fact that the selection of the leading element is limited to the elements of the selected block, it is possible that all the elements of this block are small in absolute value and cannot be taken as the leading ones.

It is worth noting that this can actually signal the poor conditioning of the diagonal block being factorized, but does not necessarily mean that the original matrix is poorly conditioned. The `uspars_fact()` function in this case returns code 4 (*Zero pivot*). This code indicates that factorization was not carried out because the leading element could not be found in the local matrix.

To overcome this difficulty, the USPARS package has developed special techniques described in sections C.3.1, C.3.2.

C.3.1. Diagonal boosting

Suppose a `Zero pivot` situation occurred during the factorization process (the `uspars_fact()` function returned code 4). In other words, the process got into a situation described by the formula (C.3), where all the elements of the selected block are small

$$|a_{ij}| < \delta_{boost} \|A\|_1, \quad k \le i,j \le m.$$

Here $\|A\|_1$ denotes the norm of the original matrix that is being factorized, δ_{boost} is a small positive parameter. If the block

$$\begin{pmatrix} a_{kk} & a_{k,k+1} & \cdots & a_{km} \\ a_{k+1,k} & a_{k+1,k+1} & \cdots & a_{k+1,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{mk} & a_{m,k+1} & \cdots & a_{mm} \end{pmatrix}$$

replace it with a block

$$\delta_{boost} \|A\|_1 I_{m-k-1}, \qquad (C.4)$$

proportional to the unit matrix of order $m-k+1$, then the factorization can be continued. This technique is called *boosting* in English literature. To enable the option, see section 2.2.

By applying this technique several times, if necessary, factorization can be completed. However, it is not the original matrix that is factorized, but some perturbation of it. If the introduced perturbation at the

[1] more correctly, well-conditioned

support steps is not large, in other words, δ_{boost} is not large, then we can hope then we can hope that the result is a factorized matrix close to the original one.

Using the obtained triangular factors as a preconditioner, it is possible to solve a system of linear equations with the initial matrix of coefficients by the iterative method. The USPARS package uses the simple iteration method as an iterative method, in which the residuals are calculated from the original system, and triangular factors obtained using the support are used to calculate the next iteration. Assuming that the introduced disturbance was not too great, one can hope that the iterations will converge, which is confirmed by many examples. It should be borne in mind that enabling support requires additional time compared to a hypothetical situation where all calculations were completed without support.

The use of support is illustrated with `example_boosting.c` (see section A.19).

C.3.2. Global pivoting

In the case of returned code 4 by `uspars_fact()`, block

$$\hat{A}_{ii} = \begin{pmatrix} \times & \times & \times & \cdots & \times \\ \times & a_{kk} & a_{k,k+1} & \cdots & a_{km} \\ \times & a_{k+1,k} & a_{k+1,k+1} & \cdots & a_{k+1,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \times & a_{mk} & a_{m,k+1} & \cdots & a_{mm} \end{pmatrix}$$

in formula (C.3) is 'zero', i.e., satisfies the condition

$$\max_{k\le i,j\le m} |a_{ij}| \quad < \delta_{gp} \max_{1\le i,j\le N} |a_{ij}| \qquad (C.5)$$

Here, N is the order of the original matrix being factorized. The maximum on the right-hand side of inequality (C.5) is calculated over the elements of the original matrix, while the maximum on the left-hand side is calculated over the elements of the current block $\hat{A}_{ii}$.

The technique described in this section, when applied to the situation with the 'zero' subblock of block A_{ii}, can be illustrated by the following formulas.

$$\begin{pmatrix} A_{11} & \cdots & \times & \times & \times & \cdots & \times & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \times & \cdots & A_{i-1,i-1} & \times & \times & \cdots & \times & 0 \\ \times & \cdots & \times & A_{i,i} & \times & \cdots & \times & 0 \\ \times & \cdots & \times & \times & A_{i+1,i+1} & \cdots & \times & 0 \\ \vdots & \cdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \times & \cdots & \times & \times & \times & \cdots & A_{nn} & 0 \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & I \end{pmatrix} \qquad (C.6)$$

Here, the highlighted block row and the highlighted block column are based on the 'zero' subblock of the diagonal block A_{ii}. To reflect the fact that this is part of the entire block, the borders are drawn intersecting the symbol A_{ii}. The matrix is bordered on the right by a block column and on the bottom by a block row, the widths of which match the order of the 'zero' block. All border elements are zero, except for the lower-right block, which contains the identity matrix of the corresponding size.

Adding columns to a matrix in terms of the system of linear equations being solved can be interpreted as adding unknowns to the system. We pad the right-hand side vector with zeros from below so that the number of components in the vector equals the number of rows in the bordered matrix. Thus, additional rows represent added equations.

A symmetrical permutation of rows and columns is applied to the augmented matrix (C.6) so that the added unit block from the border falls into the place of the 'zero' block. The matrix structure takes the form

$$
\left(\begin{array}{ccc|c|ccc|c}
A_{11} & \cdots & \times & \times & \times & \cdots & \times & 0 \\
\vdots & \ddots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\
\times & \cdots & A_{i-1,i-1} & \times & \times & \cdots & \times & 0 \\
\hline
\times & \cdots & \times & \widetilde{A_{i,i}} & \times & \cdots & \times & 0 \\
\hline
\times & \cdots & \times & \times & A_{i+1,i+1} & \cdots & \times & 0 \\
\vdots & \cdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\times & \cdots & \times & \times & \times & \cdots & A_{nn} & 0 \\
\hline
0 & \cdots & 0 & 0 & 0 & \cdots & 0 & \hat{A}_{ii}
\end{array}\right) \tag{C.7}
$$

The identity matrix in the lower right corner is now replaced by a 'zero' block, labeled $\hat{A}_{ii}$. To indicate that in block A_{ii}, the 'zero' block has been replaced by the identity matrix, we use the tilde sign. The corresponding permutations must also be made in the right-hand side vector.

After this transformation, the situation causing the zero pivot disappears, and the factorization process can continue. If a zero pivot occurs again, the transformations described above are repeated. Clearly, this leads to block A_{nn}, after which a certain number of added rows and columns remain. To eliminate them, we must use the calculated factorization of the modified matrix, which ultimately leads to the problem of factorizing the added block located in the lower right corner. By factoring it, we obtain a triangular matrix decomposition of the augmented matrix. By solving the augmented system, we determine a vector of unknowns containing the added components. These components should be excluded from the answer.

If the original system matrix was well-conditioned and zero pivot errors resulted from poorly chosen row and column ordering, the last bottom block will not be too ill-conditioned[1], and its factorization will be possible. Experience with this technique confirms this. See Section 2.2 for how to enable this option.

An obvious drawback of this technique is increased memory consumption. Consequently, factorization time increases compared to the hypothetical time it would take if zero pivot errors had not occurred and factorization had been possible for the original matrix.

C.4. In-Core (IC) and Out-Of-Core (OOC) modes

The ability of the USPARS package to solve a specific SLAE depends on many factors, not the least of which are the characteristics of the shared-memory computing system (hardware characteristics). As a result of factorization of the coefficient matrix, the number of nonzero elements in the triangular factors can be tens or even hundreds of times greater than the number of nonzero elements in the original coefficient matrix. During the SLAE solution process, the triangular factors are intermediate results and must be saved before the solution stage. Insufficient available RAM can be a critical factor when solving large SLAEs.

Using disk memory for temporary storing intermediate results to reduce RAM requirements is an effective solution to the problem described in the previous paragraph. This approach is called out-of-core (OOC) and it is used in many SLAE solving packages. The USPARS package implements two computation modes:

- In-Core (IC) – a mode in which all intermediate results are stored in RAM;
- Out-Of-Core (OOC) – a mode in which disk is used to store some of the intermediate results.

Due to increasing amount of data transfers, the total computation time in OOC mode will be longer than on the same computing system equipped with additional RAM. Thus, OOC mode allows solving SLAEs for which RAM is insufficient when using IC mode.

To select a particular mode, use the `options[USP_OOC_MEM]` value. See Section 2.2 for a detailed description of how to enable each mode. The directory containing the files containing intermediate data obtained during factorization is determined by the `USP_OOC_PATH` environment variable. If this variable is not set, the working directory is used. The prefix for the files containing intermediate data is specified by

[1] This cautious formulation was chosen because the corresponding theorems justifying this assertion are unknown. It's possible that they simply don't exist.

the USP_OOC_NAME environment variable. If this variable is not set, the name ooc_lu is used. Upon completion of USPARS, the created files are always deleted by calling the uspars_destroy() function.

D. Migration from MKL PARDISO to USPARS

This section is intended to simplify migration from MKL PARDISO to USPARS for users. It describes the matrix definition specifics, correlations between parameters of both solvers, and provides some usage tips. Since USPARS does not have FORTRAN interfaces, this section describes examples in C. The MKL PARDISO interfaces correspond to the version from the Intel® oneAPI Math Kernel Library for C 2023.1

D.1. Interfaces and solution phases

In USPARS, like in PARDISO, the computation process consists of three phases: initialization, factorization, and solving. The main differences are the default parameters and the lack of separation of the solving phases into three steps: forward, diagonal, and backward step.

D.1.1. Memory allocation for the internal structure and default parameters setting

In PARDISO, the `pardisoinit(…)` function allocates memory for the internal `pt` structure, specifies the matrix type, and sets the `iparm` parameter array with default values. In USPARS, the `uspars_alloc(…)` function is used for this purpose, and parameters values in the `options` array are set equal to `-1` (see 2.1).

PARDISO:

```
void *pt[64];
MKL_INT iparm[64];
MKL_INT mtype;
pardisoinit(pt, &mtype, iparm);
```

USPARS:

```
void *tt = NULL;
int options[USP_OPTIONS];
for (int i = 0; i < USP_OPTIONS; i++)
  options[i] = -1;
uspars_alloc(&tt);
```

D.1.2. Running the solution phases

PARDISO has a common interface for running all solution phases – `pardiso(…)`. The phase argument is used to select the phase. In USPARS, these phases have different names – `uspars_init(…)`, `uspars_fact(…)`, `usparse_solve(…)` (see 2.2, 2.3, 2.4).

The following is a matching between the arguments of the `pardiso(…)` function and the USPARS parameters:

```
void pardiso(_MKL_DSS_HANDLE_t pt, const MKL_INT *maxfct, const MKL_INT
*mnum, const MKL_INT *mtype, const MKL_INT *phase, const MKL_INT *n,
const void *a, const MKL_INT *ia, const MKL_INT *ja, MKL_INT *perm,
const MKL_INT *nrhs, MKL_INT *iparm, const MKL_INT *msglvl, void *b,
void *x, MKL_INT *error);
```

- `pt` – internal structure. It’s set similarly and used as argument to all functions in USPARS.
- `maxfct, mnum` – there are no corresponding parameters in USPARS. It is assumed that `maxfct=1, mnum=1`.
- `mtype` – matrix type, this parameter explicitly defines factorization type. In USPARS, the analogue is `usp_factorize_type` (for more details, see 2.2). It’s an argument of `uspars_init(…)` function.
- `phase` – the phase of the solution. In USPARS this argument is absent, since the phase of the solution is determined by the function name.

phase	*USPARS function*
11	`uspars_init(…)`
22	`uspars_fact(…)`

33	uspars_solve(…)

- n – the number of unknowns. In USPARS it is specified similarly. It is an argument to uspars_init(…) (see 2.2).
- a, ia, ja – matrix in CSR format. In USPARS it is specified similarly (see 1.1). It is an argument to uspars_init(…) (see 2.2).
- perm – permutation vector that can be used to set a user-defined reordering or to get a calculated reordering. In USPARS, the uspars_param_get(…) function (see 2.5) is responsible for getting information obtained during the calculation, and the uspars_param_set(…) function (see 2.6) is responsible for setting parameters other than integer ones. More details on setting and getting a permutation vector are described in Section D.3.
- nrhs – the number of right-hand sides. In USPARS it is specified similarly and is an argument to uspars_solve(…) (see 2.4).
- iparm – solver parameters. In USPARS they are set similarly, it is an argument of uspars_init(…) (see 2.2)
- msglvl – logging level. In USPARS, this parameter is not defined as a separate argument, but is part of the options array – options[USP_MSGLVL]. In addition to enabling and disabling logging, there is an intermediate option (see 2.2).
- b, x – the right-hand side vector and the solution vector. In PARDISO, the right-hand side vector can be overwritten by the solution vector or stored separately. In USPARS, the right-hand side is an argument to uspars_solve(…). It is always overwritten by solution vector during calculations (see 2.4).
- error – an error flag. In USPARS, this is the return code for all functions. More information about possible values is provided in the section with the corresponding function, and their interpretation is in Section 3.

A special note about using 32/64-bit integer interfaces. In PARDISO, pardiso_64(…) function is used for 64-bit integer and pardiso(…) function for 32-bit integer. In USPARS, the difference between 32-bit and 64-bit integer interfaces only in the initialization stage, 64-bit functionality is initialized by the uspars_init64(…) function (see 2.2), all other functions are the same for both 32/64 bit.

D.2. Matrix format and factorization type

USPARS uses the same input matrix format as MKL PARDISO – Sparse CSR. General matrices are specified in the standard manner, while symmetric and Hermitian matrices are specified in the upper-triangular form (see 1.1.2). In other words, the arrays a, ia, ja are passed to the pardiso(…) function can be passed in the same form to the uspars_init(…) function (see 2.2).

There are some differences in the specification of factorization types. In PARDISO the input matrix type is specified via the mtype parameter, and the solver selects the factorization based on this information. In USPARS the factorization type is specified directly (see Table 2.2.2). Below is the correspondence between the matrix type, its mtype in PARDISO, and the parameters in USPARS, taking into account the scalar matrix type (see Table 1.3.1).

	PARDISO		*USPARS*	
Matrix type	**mtype**	**iparm[27]**	**usp_factorize_type**	**usp_scalar_type**
Real, structurally symmetric	1	0	USP_LU	USP_DOUBLE
		1		USP_FLOAT
Real, symmetric, positive definite	2	0	USP_LLT	USP_DOUBLE
		1		USP_FLOAT
Real, symmetric	-2	0	USP_LDLT	USP_DOUBLE
		1		USP_FLOAT
Complex, structurally symmetric	3	0	USP_LU	USP_COMPLEX_DOUBLE
		1		USP_COMPLEX_FLOAT
Complex, Hermitian, positive definite	4	0	USP_LLH	USP_COMPLEX_DOUBLE
		1		USP_COMPLEX_FLOAT
Complex, Hermitian	-4	0	USP_LDLH	USP_COMPLEX_DOUBLE
		1		USP_COMPLEX_FLOAT
Complex, symmetric	6	0	USP_LDLT	USP_COMPLEX_DOUBLE
		1		USP_COMPLEX_FLOAT
Real, general	11	0	USP_LU	USP_DOUBLE
		1		USP_FLOAT
Complex, general	13	0	USP_LU	USP_COMPLEX_DOUBLE
		1		USP_COMPLEX_FLOAT

D.3. Solver parameters

In PARDISO solver parameters are controlled via the `iparm` array. In USPARS this array is called `options` (see 2.2). Additionally, real parameters can be controlled using `uspars_param_get(…)` (see 2.5) and `uspars_param_set(…)` (see 2.6). Parameters are mapped as follows:

- `iparm[0]`. Default parameter values.
  To use all default parameters in USPARS, assign the value -1 to all `options` parameters. In this case, any optional `options` parameter can be used by default if you assign it the value -1.
- `iparm[1]`. Reordering type.
  In USPARS type of reordering is defined by the `options[USP_RTYPE]`. Values are related as follows:

Value of `iparm[1]`	Reordering type	Value of `options[USP_RTYPE]`
0	Minimum degree algorithm	-
2	METIS	2
3	Custom parallel algorithm (see B.4)	1
-	Do not apply any permutation (reordering)	0
see `iparm[4]`	Apply user-defined permutation (reordering)	3

- `iparm[3]`. Preconditioner for CGS/CG.
  USPARS doesn't support this functionality and doesn't have corresponding parameter.
- `iparm[4]`. Usage scenario of `perm` array in `pardiso(…)` function.

Value of `iparm[4]`	Description	Usage in USPARS	Call agreement
0	-	-	-
1	Apply	`options[USP_RTYPE]`	`uspars_param_set`

	user-defined permutation specified by `perm` array	=3, `uspars_param_set(&tt, USP_PERM_VECTOR, perm)`, `perm` – user-defined permutation array	(…) must be called between `uspars_alloc(…)` и `uspars_init(…)`
2	Getting permutation vector in `perm` array	`uspars_param_get(&tt, USP_PERM_VECTOR, perm)`	`uspars_param_get(…)` must be called after `uspars_init(…)`

- `iparm[5]`. Writing mode of solution vector (out-of-place/in-place).
  In USPARS, solution vector is written in-place in right-hand side vector, similar to `iparm[5]=1` case.
- `iparm[6]`. Actual number of iterative refinement steps.
  In USPARS, information about computation process can be obtained in advanced logging mode, which is enabled by `options[USP_MSGLVL]=2`.
- `iparm[7]`. Maximum number of iterative refinement steps.
  In USPARS, this parameter can be set by `options[USP_ITER_REF]` (see 2.2, C.2). Zero value of parameter means that iterative refinement is disabled. Moreover, if iterative process doesn't converge, the best solution in terms of residual from some iteration can be obtained by setting `options[USP_BRX]=1`.
- `iparm[8]`. Threshold value for convergence criteria of iterative refinement.
  Threshold value for convergence criteria in terms of relative residual is set by `uspars_param_set(&tt, USP_ITER_STOP_CRIT, value)`. Iterative refinement process stops when relative residual becomes smaller than value.
- `iparm[9]`. Pivot perturbation (boosting).
- Perturbation of small pivots can improve numerical stability during factorization. In USPARS this option is enabled by `options[USP_BOOSTING]`. This option also affects the number of iterative refinement steps (see 2.2, C.3.1). Threshold value for pivot perturbation can be set directly by `uspars_param_set(&tt, USP_BOOSTING_VALUE, value)`. Also USPARS has another functionality to avoid small pivots (see C.3.2).
- `iparm[10]`, `iparm[12]`. Matching and scaling.
  In PARDISO, maximum weighted matching is enabled by `iparm[12]` and `iparm[10]` enables matching-based scaling. In USPARS these two options are enabled by single parameter `options[USP_MATCHING]` (see 2.2, B.3).

Value of `iparm[10]`	Value of `iparm[12]`	Decription	Value of `options[USP_MATCHING]`
0	0	Matching is disabled	0
0	1	Matching is enabled	2
1	1	Matching and matching-based scaling are enabled	3

  Also USPARS has another type of scaling (`options[USP_SCALING]=1`, see B.2) and another type of matching (`options[USP_MATCHING]=1`).
- `iparm[11]`. Solution of conjugate system.
  In current version, this functionality isn't available.
- `iparm[13]`, `iparm[14]`, `iparm[15]`, `iparm[16]`, `iparm[17]`, `iparm[18]`, `iparm[19]`. Output parameters.
  In USPARS, information about computation process can be obtained in advanced logging mode, which is enabled by `options[USP_MSGLVL]=2`.
- `iparm[20]`. Baunch-Kaufman 2x2 pivoting for symmetrical matrices.

In USPARS, 2x2 pivoting is enabled by parameter options[USP_BK]. In opposite to PARDISO, this parameter doesn't affect iterative refinement.

Value of iparm[20]	Description	Value of options[USP_BK]
0	1x1 pivoting + iterative refinement	0
1	1x1 and 2x2 pivoting + iterative refinement	1
2	1x1 pivoting	0
3	1x1 и 2x2 pivoting	1

- iparm[21], iparm[22]. Output parameters. Number of positive/negative eigenvalues. It doesn't supported in USPARS.
- iparm[23], iparm[24]. Parallel execution.
  USPARS does not expose switches for selecting internal parallel algorithms. The number of OpenMP threads used by USPARS is set by options[USP_OMP_THREADS] (see C.1). If this parameter is left equal to -1, omp_get_max_threads() at the moment of the uspars_init() call is used.
- iparm[26]. Matrix check before solution.
  USPARS performs matrix check by default (see 3.2), control of this doesn't supported.
- iparm[27]. Single/double precision.
  It's defined by argument uspars_init(…) function. More details in E.2.
- iparm[29]. Output parameter. Number of negative pivots when returned error code -4.
  USPARS doesn't provide this information when returned similar error.
- iparm[30]. Partial solution.
  USPARS always computes full solution and this functionality doesn't supported.
- iparm[33]. CNR mode.
  It doesn't supported in USPARS.
- iparm[34]. Zero-based/one-based indexing of arrays.
  USPARS has zero-based indexing. The same as iparm[34]=1.
- iparm[35]. Schur's complement.
  USPARS doesn't provide Schur's complement values to users. If you need this functionality, please contact with developers.
- iparm[36]. Input matrix format.
  USPARS support only CSR format. This equals to setting iparm[36]=0.
- iparm[38]. Low-Rank Update.
  This functionality doesn't supported in USPARS.
- iparm[42]. Diagonal of inverse matrix.
- In USPARS, this functionality is available through uspars_param_get(…) with USP_D_VECTOR argument (см. 2.5)
- iparm[55]. Diagonal elements control.
  This functionality doesn't supported in USPARS. If you require this functionality, please contact developers.
- iparm[59], iparm[62]. Out-of-core (OOC)/in-core (IC) mode
  In PARDISO, limitation of RAM amount usage in out-of-core (OOC) mode defined by MKL_PARDISO_OOC_MAX_CORE_SIZE environment variable. In USPARS, this value is specified directly in the options. Control between automatic and forced OOC is performed using a sign.

Value of iparm[59]	MKL_PARDISO_OOC_MAX_CORE_SIZE	Description	Value of options[USP_MEM_OOC]
0	-	IC mode	0
1	value	Automatic	value

		IC/OOC	
2	value	OOC mode	-value

References

[1] Pieter Ghysels, Xiaoye S Li, Francois-Henry Rouet, Samuel Williams, and Artem Napov. An efficient multicore implementation of a novel hss-structured multifrontal solver using randomized sampling. SIAM Journal on Scientific Computing, 38(5):S358–S384, 2016.

[2] STRUMPACK (STRUctured Matrix PACKage), https://portal.nersc.gov/project/sparse/strumpack/index.html

[3] Intel® Math Kernel Library (Intel® MKL), https://software.intel.com/en-us/intel-mkl, 2017.

[4] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing, 20(1):359–392, 1998.

[5] Joseph W.H. Liu. Modification of the minimum-degree algorithm by multiple elimination. ACM Transactions on Mathematical Software (TOMS), 11(2):141–153, 1985.

[6] Pinar Heggernes, SC Eisestat, Gary Kumfert, and Alex Pothen. The computational complexity of the minimum degree algorithm. Technical report, Institute for computer applications in science and engineering, Hampton, VA, 2001.

[7] George Karypis and Vipin Kumar. Metis, a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices version 4.0. 1998.

[8] Iain S Duff and Jacko Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. SIAM Journal on Matrix Analysis and Applications, 20(4):889–901, 1999.

[9] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. Journal of parallel and distributed computing, 48(1):71–95, 1998.