SKYRS: A SKYLINE SPARSE MATRIX SOLVER IN RUST

PHILIPPE HELLUY

1. Theory

1.1. **Notations.** We want to solve a linear system

$$Ax = b, \tag{1.1}$$

where A is a $n \times n$ real (for instance) matrix. We use the following notations:

- The range $i \dots j$ is the list of integers $[i, i+1, \dots, j-1]$. Please note that j is not included in the range (this is similar to the C, Python and Rust notations) and that if if $i \geq j$, then $i \dots j = []$ (the range is empty).
- The elements of the matrix A are noted a^i_j: the row index i is noted with a superscript and the column index j with a subscript. We use the convention that the row and column indices are in the range $0 \dots n$.
- $a^{i\dots k}_{j\dots l}$ is the sub-matrix of A constructed from the given rows and columns ranges. If one range is empty, the sub-matrix is empty . The scalar product of two empty vectors is assumed to be zero.

1.2. *LU* **decomposition.** It is generally possible to write A under the form

$$A = LU, \tag{1.2}$$

where L is a lower triangular matrix with a unit diagonal and U is an upper triangular matrix. The LU decomposition allows to reduce (1.1) to the resolution of two triangular linear systems.

Traditionally, in the computer algorithm, the decomposition is stored in the same place as the initial matrix A. The following algorithm (Doolittle algorithm) replaces A by $L - I + U$. For an easier reading, we distinguish between L, U and A, but in practice we can replace L and U by A in the Doolittle algorithm.

- Initialization: $L - I + U = A$.
- for k in $1 \dots n$ (diagonal loop on the pivots)
  - for j in $0 \dots k$

$$L^k_j \leftarrow \frac{1}{U^j_j} \left(L^k_j - \sum_{0 \leq p < j} L^k_p U^p_j \right)$$

    (update row of L left to the pivot).
  - for i in $0 \dots k+1$

$$U^i_k \leftarrow U^i_k - \sum_{0 \leq p < i} L^i_p U^p_k$$

    (update the column of U above the pivot).

This algorithm works if the pivot U^k_k never vanishes. It can be rewritten with scalar products of sub-rows of L with sub-columns of U:

- Initialization: $L - I + U = A$.
- for k in $1 \dots n$
  - for j in $0 \dots k$

$$L_j^k \leftarrow L_j^k - L_{0\dots j}^k \cdot U_j^{0\dots j}$$

$$L_j^k \leftarrow \frac{L_j^k}{U_j^j}$$

  - for i in $0 \dots k+1$

$$U_k^i \leftarrow U_k^i - L_{0\dots i}^i \cdot U_k^{0\dots i}$$

1.3. **Triangular solve.** Once the decomposition is computed, for solving

$$Ax = b,$$

we solve two triangular systems, first

$$Ly = b,$$

and then

$$Ux = y.$$

The solution of $Ly = b$ is given by the following algorithm

- Initialization: $y^0 = b^0$
- for i in $1 \dots n$

$$y^i = b^i - \sum_{0 \le j < i} L_j^i y^j$$

  or

$$y^i = b^i - L_{0\dots i}^i \cdot y^{0\dots i}$$

  The solution of $Ux = y$ is given by the following algorithm
- Initialization: $x^{n-1} = y^{n-1}/U_{n-1}^{n-1}$
- for i in $[n-2, n-3, \dots, 0]$

$$x^i = \frac{1}{U_i^i}\left(y^i - \sum_{i+1 \le j < n} U_j^i x^j\right)$$

  or

$$x^i = \frac{1}{U_i^i}\left(y^i - U_{i+1\dots n}^i \cdot x^{i+1\dots n}\right)$$

2. Sequential implementation for sparse matrices

2.1. **Sparse skyline matrix.** The matrix A is said to be sparse if it contains many zeros. For this kind of matrix it is interesting to find data structures that avoid the storage of the zeros and unnecessary operations with these zeros. A nice case is when the pattern of the zeros is the same for A and $L - I + U$. It is indeed possible to find such patterns. The skyline storage is an example of such a pattern. We need some definitions:

- An array $\text{prof}[0 \dots n]$ is called a profile of A if $A_j^i = 0$ when $j < \text{prof}[i]$. A couple of indices (i, j) is said to be in the profile of A if $j \ge \text{prof}[i]$.
- An array $\text{sky}[0 \dots n]$ is called a skyline of A if $A_j^i = 0$ when $i < \text{sky}[j]$. A couple of indices (i, j) is said to be in the skyline of A if $i \ge \text{sky}[j]$.

Remark: if (i, j) correspond to a non-zero value $a^i_j \neq 0$ then (i, j) is both in the profile and in the skyline of A. We can have zeros in the profile and in the skyline, they are simply treated as non-zero terms. The limit cases correspond to $\text{prof}[i] = 0$, $\text{sky}[j] = 0$ (full matrix storage) and $\text{prof}[i] = i$, $\text{sky}[j] = j$ (diagonal matrix).

It is possible to prove the following theorem.

Theorem 1. *If prof is a profile of A and sky a skyline of A then prof remains a profile and sky a skyline during all the steps of the Gauss algorithm and in the end prof is a profile of $L - I + U$ and sky a skyline of $L - I + U$.*

The most efficient method for storing a sparse matrix in term of memory is the coordinate format (coo). Assume that the matrix has nnz non-zero entries. We store it in an array $\text{coo}[0 \dots nnz]$ of triplets (i, j, v) such that

$$\text{coo}[k] = (i, j, v) \Rightarrow a^i_j = v.$$

We will store the lines of L (without the diagonal terms) in the array ltab, the columns of U (with the diagonal terms) in utab. ltab and utab are two vectors of vectors of 64 bits double precision numbers. Because we cannot know a priori the number of non-zero terms in the lines of L and in the columns of U we need first to construct the skyline and the profile. This is done by the following algorithm:

- Initialization: $\text{prof}[i] = i$, $\text{sky}[j] = j$.
- for k in $0 \dots nnz$
  - $(i, j, v) = \text{coo}[k]$
  - if $i > j$ and $v \neq 0$ then $\text{prof}[i] = \min(\text{prof}[i], j)$
  - if $i < j$ and $v \neq 0$ then $\text{sky}[j] = \min(\text{sky}[j], i)$

We can now construct ltab[][] and utab[][] with the following algorithm

- Initialization: $\text{ltab}[i] = [0; i - \text{prof}[i]]$, $\text{utab}[j] = [0; j - \text{sky}[j] + 1]$
- for k in $0 \dots nnz$
  - $(i, j, v) = \text{coo}[k]$
  - if $i > j$ then $\text{ltab}[i][j - \text{prof}[i]] \mathrel{+}= v$
  - if $i \leq j$ then $\text{utab}[j][i - \text{sky}[j]] \mathrel{+}= v$

2.2. **Optimisation of the Doolittle algorithm.** The Doolittle elimination algorithm given above can yet be optimized for the skyline representation. This gives:

- for k in $1 \dots n$
  - for j in $\text{prof}[k] \dots k$

$$L^k_j \leftarrow \frac{1}{U^j_j} \left(L^k_j - \sum_p L^k_p U^p_j \right), \quad p \in \max(\text{prof}[k], \text{sky}[j]) \dots j$$

  - for i in $\text{sky}[k] \dots k + 1$

$$U^i_k \leftarrow U^i_k - \sum_p L^i_p \cdot U^p_k, \quad p \in \max(\text{prof}[i], \text{sky}[k]) \dots i$$

Additional optimization: the scalar products of rows of L with columns of U are optimized by unsafe calls to the BLAS subroutine "ddot".

2.3. **Optimisation of the triangular solvers.** The optimisation of the L solver is easy

- for i in $0 \dots n$
$$y^i = b^i - L^i_{\text{prof}[i]\dots i} \cdot y^{0\dots i}$$

The optimisation of the U solver is less simple (because the rows of U are less natural to access than the columns in the skyline storage). The naive algorithm would be to write:

- for i in $[n-1, n-2, \dots, 0]$
$$x^i = \frac{1}{U^i_i}\left(y^i - \sum_{i<j<n} U^i_j x^j\right), \quad \max(i, \text{sky}[j]) < j < n$$

But this is not the good method, because one would have to loop on all j and test if $\max(i, \text{sky}[j]) < j$. A better way is to loop on the columns of U starting from the last one and add the corresponding contribution to the x vector. When the contributions of column j are treated we can divide $x^{j-1}$by $U^{j-1}_{j-1}$and continue. We obtain the following algorithm

- Initialization $x^{n-1} = y^{n-1}/U^{n-1}_{n-1}$
- for j in $[n-2, n-3, \dots, 0]$
  - for i in $0..j+1$
$$x^i \leftarrow x^i - U^i_{j+1} x^{j+1}$$
  - $x^j \leftarrow x^j/U^j_j$

In this form, we can use the skyline structure, we obtain

- Initialization $x^{n-1} = y^{n-1}/U^{n-1}_{n-1}$
- for j in $[n-2, n-3, \dots, 0]$
  - for i in $\text{sky}[j+1]..j+1$
$$x^i \leftarrow x^i - U^i_{j+1} x^{j+1}$$
  - $x^j \leftarrow x^j/U^j_j$

Usual trick: if the vector b is not needed anymore, the computations can be done in place with $x = y = b$. At the end b is replaced by the solution x.

3. Parallelization with the bisection method

3.1. **Bisection renumbering.** We consider the usual undirected graph $G = (V, E)$ associated to the sparse matrix A. The set of vertices $V = \{0, 1, \dots n-1\}$ and the set of edges $E \subset V \times V$. The couple (i, j) is an edge of G if $A^i_j \neq 0$ or $A^j_i \neq 0$. We denote by $|C|$ the number of elements in the set C. We consider a partition of the vertices of G:
$$V = V_0 \cup W, \quad V_0 \cap W = \emptyset.$$
A good partition is a partition where $|V_0| \simeq |W|$ and the number of edges between V_0 and W is minimal. It is difficult to construct an optimal bisection. We plan to consider in the following two different heuristics for splitting V: by a breadth-first-search (BFS) algorithm and by a spectral bisection method. For the moment, only the BFS renumbering is used. Assume that the splitting is done. We then denote by V_1 the set of vertices of W that are not connected to a vertex in V_0. And we

denote by V_2 the remaining vertices: they are in W and connected to a vertex in V_0. Let us define

$$n_0 = |V_0|, \quad n_1 = |V_1|, \quad n_2 = |V_2|.$$

We then construct a permutation of the vertices of V by renumbering first the vertices of V_0 then the vertices in V_1 and finally the vertices in V_2. More precisely, let us denote by $\sigma : V \to V$ this permutation, it has to satisfy

$$\begin{aligned} &\sigma(k) \in V_0, \quad 0 \leq k < n_0, \\ &\sigma(k) \in V_1, \quad 0 \leq k - n_0 < n_1, \\ &\sigma(k) \in V_2, \quad 0 \leq k - n_0 - n_1 < n_2. \end{aligned}$$

We now consider the permuted matrix B constructed from A by the formula

$$B^i_j = A^{\sigma(i)}_{\sigma(j)}.$$

The matrix B has the following structure

$$B = \begin{pmatrix} C^0_0 & 0 & C^0_2 \\ 0 & C^1_1 & C^1_2 \\ C^2_0 & C^2_1 & C^2_2 \end{pmatrix}, \tag{3.1}$$

where the block matrix C^i_j is of size $n_i \times n_j$. The interest of this structure is that the Doolittle algorithm can now be performed independently in parallel on the C^0_0 and C^1_1 blocks. Then the remaining blocks can be computed. If $n_2 \ll n_0$ and $n_2 \ll n_1$ this can be much more efficient. In addition, the bisection can be reiterated on the diagonal blocks recursively.

3.2. **Bibliographical note.** It is not possible to cite all the research works done on sparse matrix solvers and nested bisection algorithm. We refer to the short bibliography given at the end of this report. The reader is alerted on the fact that the `skyrs` library is relatively fast, but that it cannot compete with the most up-to-date sparse LU solvers, such as SuperLU [7], UMFPACK [1] or PaStiX [6]. But `skyrs` is not wrapper, it is written in Rust !

3.3. **Notes on the implementation.**

- I have programmed the `skyrs` library in Rust. It can be found at `https://github.com/phelluy/skyrs`.
- The main struct is the `Sky` struct. A skyline matrix is constructed from a matrix given in the coordinate format (see the example given in the online documentation).
- The library has two main functions:
  - the `vec_mult` function performs a matrix vector product (that is parallelized thanks to `Rayon`);
  - the `solve` function solves the linear system $Ax = b$. At the first invocation, the bisection renumbering and the LU decomposition are performed. On the next invocations, the LU decomposition is reused and the resolution is faster.
- `skyrs` uses BLAS (Basic Linear Algebra Subroutines) for accelerating scalar products computations. A working installation of a BLAS library is thus mandatory. It depends on the OS.

- The nested bisection is programmed with a recursive function. The node permutation is updated recursively and the bisection data are stored in a binary tree (in practice I used a Rust HashMap for storing the tree).
- The parallelism is implemented thanks to the Rust `Rayon` library. I used two features of the `Rayon` library:
  - the `join` function for recursively compute the LU factors of C_0^0 and C_1^1 in (3.1);
  - the `par_iter` iterator when it is possible for accelerating computation loops. This is particularly important for accelerating the computations of the in-place LU factors in the extra-diagonal terms of (3.1).
- An extended example is given in `example/lap2d.rs`. It solves the Laplace equation on a rectangular finite difference grid. This example requires a working installation of Python3 and Matplotlib for viewing the plots.
- TODO list:
  - modify the library so that it can be used with other type. For the moment, only the f64 type is possible.
  - do not rely on Python for matrix plotting.
  - improve the bisection algorithm with a spectral bisection method.

3.4. **Small benchmark.** We solve the Laplace equation on a finite difference grid of size 1200×300. The initial numbering generates a skyline matrix with a bandwidth of 1200. We obtain the results presented in Table (1). The Python code for the comparison with `numpy` is given in the `examples` directory.

algorithm	time (s)	fill-in (nnz)
skyrs: no renumber. (sequential)	146	865,804,501
skyrs: bfs renumber. (sequential)	155	864,365,701
skyrs: bisection (sequential, no BLAS)	58	309,334,093
skyrs: bisection (sequential)	17.2	309,334,093
skyrs: bisection (parallel)	11.8	309,334,093
numpy: SuperLU (parallel)	6.3	?

TABLE 1. Comparison of several LU implementation for solving the Laplace on 1200×300 finite difference grid. The computations have been done on an Apple M1 silicon CPU with eight cores. The BLAS library is Accelerate (Apple). The coo matrix contains 1,804,501 non-zero values.

REFERENCES

[1] Timothy A Davis. Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, 30(2):196–199, 2004.

[2] Gouri Dhatt, Emmanuel Lefrançois, and Gilbert Touzot. *Finite element method*. John Wiley & Sons, 2012.

[3] Iain S Duff, Albert M Erisman, and John K Reid. On george's nested dissection method. *Siam journal on numerical analysis*, 13(5):686–695, 1976.

[4] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[5] Laura Grigori, Erik G Boman, Simplice Donfack, and Timothy A Davis. Hypergraph-based unsymmetric nested dissection ordering for sparse lu factorization. *SIAM Journal on Scientific Computing*, 32(6):3426–3446, 2010.

[6] Pascal Hénon, Pierre Ramet, and Jean Roman. Pastix: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Computing*, 28(2):301–321, 2002.
[7] Xiaoye S Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):302–325, 2005.
[8] François Pellegrini and Jean Roman. Sparse matrix ordering with scotch. In *International Conference on High-Performance Computing and Networking*, pages 370–378. Springer, 1997.