Details of the implementation of the linear solver for the new HiGHS interior point method

Filippo Zanetti

November 21, 2024

1 BLAS functions

We make extensive use of BLAS functions and their notation.

Each BLAS function name consists of five letters:

- The first letter specifies the precision, `s` for single, `d` for double and so on. We leave the first letter empty `_`, as the behaviour of the function is independent of it.
- The second and third letters indicate the type of matrix that are used. Some of the most common examples that we use are:
  - `ge`, generic matrix
  - `sy`, symmetric matrix
  - `tp`, triangular matrix in packed format
  - `tr`, triangular matrix in full format.
- The fourth and fifth letters indicate the operations performed. Common examples are:
  - `mm`, matrix-matrix product
  - `mv`, matrix-vector product
  - `rk`, rank-k update
  - `sm`, solve with multiple right-hand sides
  - `sv`, solve with single right-hand side.

For example, the function `_trsm` performs the forward or backward solve with multiple right-hand sides, where the triangular factor is stored in full format. Notice that not all combinations of letters are available.

Some common exceptions to the previous rules are:

- `_axpy` performs $y \leftarrow y + \alpha x$
- `_copy` copies a vector
- `_dot` performs the dot product of two vectors.

- `_scal` scales a vector.
- `_swap` swaps two vectors.

Notice also that the BLAS functions are grouped in three categories: level 1 subroutines perform vector-vector operations; level 2 subroutines perform matrix-vector operations; level 3 subroutines perform matrix-matrix operations.

1.1 Parameters

BLAS functions take a number of parameters, that indicated the type of operations to be performed, the size of the data, its location in memory and how to access it. More in details, the following are possible parameters of the subroutines.

- Inputs of type `char`. They specify better the type of operation to be performed. Typical examples are `uplo` to indicate whether the upper or lower triangle is used, `trans` to indicate to perform the operations with the transpose of the given matrix, `diag` to indicate that the matrix has unit diagonal...
- Size of the data. They specify the length of vectors and number of rows and columns of the matrices involved.
- Coefficients. They specify the multiplicative coefficients used to add the newly computed result. Usually, `alpha` is the coefficient that multiplies the new result and `beta` is the coefficient that multiplies the pre-existing vector or matrix. E.g., the operation of matrix-vector product is generalized to compute $y \leftarrow \beta y + \alpha Ax$.
- Pointers to the data. These should point to the first entry of the vector or matrix.
- Increments for vectors. They specify the number of entries to skip to access the next element of the vector. For example, if a matrix of size $n \times n$ is stored by columns in full format, any given row can be accessed by skipping n entries each time.
- Leading dimensions for matrices. They specify the leading dimension of the array that stores the matrix. If a matrix is stored by columns, the leading dimension is the number of entries to skip to access the element in the same row and in the next column. This may be larger than the size of the matrix, because the given matrix may be a submatrix of a larger matrix, stored in a larger array.

When showing the BLAS functions to call, we also show the inputs of type `char`, in order to fully characterize the operation. They are indicated with a single capital letter. E.g., `_syrk(L, N, ...)` means that the function `_syrk` should be called with parameter `uplo` set to lower and `trans` set to non-transpose.

To see more details about the meaning of these parameters and to see the complete BLAS specification, see [2] and [1] and the links therein.

2 Frontal matrices

We assume that the reader is familiar with the details of a multifrontal solver for systems of linear equations; see [4, 5] for a detailed description.

We assume that the matrix has been permuted by some ordering heuristic and that the supernodal elimination tree has been found and postordered.

For each supernode, we associate a corresponding *frontal matrix*. This matrix receives contributions from the original matrix and the children supernodes in the elimination tree; the frontal matrix then undergoes partial factorization and passes the remaining Schur complement as a contribution to its parent in the tree.

For each supernode, the fundamental steps of the sparse factorization are the following:

- Assemble the contributions of the original matrix and of the children into the frontal matrix.

- Perform dense partial factorization of the frontal matrix.

The first step involves *sparse* operations, because the original matrix and the Schur complements of the children need to be scattered into the frontal matrix of the current supernode. This step can be quite slow because it involves irregular memory accesses.

The second step involves dense operations and can be performed exploiting BLAS level 3 subroutines. The memory accesses are much more regular, but the size of the matrices to factorize can be very large.

The frontal matrix is stored as shown in Figure 1: here, k is the number of columns in the supernode, f is the size of the front and s is the size of the Schur complement that remains after the partial factorization.

The red part undergoes factorization and stores k columns of the factorization. Notice that we perform $\mathcal{LDL}^T$ factorization and not Cholesky. The ones on the diagonal of $\mathcal{L}$ are not explicitly stored; instead, the diagonal of $\mathcal{L}$ stores the elements of $\mathcal{D}$.

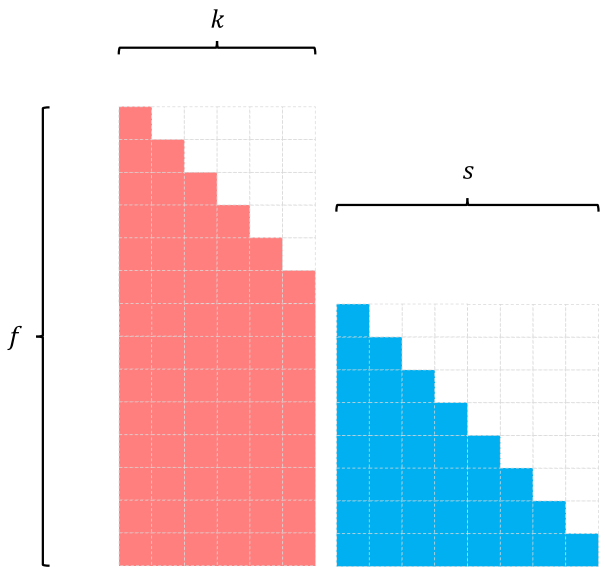
The blue part instead does not undergo factorization and stores the Schur complement that is passed to the parent in the tree. Given the two different purposes of these two parts of the matrix, given their different lifespan, and given that we assume to use completely static data structures (no pivoting involved), these two matrices are stored in two separate arrays, called respectively `frontal` and `clique`. The entries in the upper triangular part, represented as white squares in the figure, may or may not be stored, depending on the storage format used.

3 Storage formats

We consider the following formats to store dense matrices.

The *full format* (F) stores a matrix of size $m \times n$ in an array of size $m \times n$, stacking its columns and storing the upper triangular part as well, as shown in Figure 3a. Here, the numbers indicate the location within the array where a given entry is stored, starting from zero. The upper triangular elements are stored, but never accessed; they may store zeros, but this is not guaranteed.

Figure 1: Frontal matrix



They are used only to align the rows of the matrix, to provide a fixed pattern to access memory. This format requires a large amount of memory, since it stores almost twice as many entries as the minimum required.

The *lower packed format* (P) stores only the lower triangular part of a matrix, simply by skipping the upper triangular entries, as shown in Figure 3b. This format uses the minimum amount of memory required to store the matrix; however, accessing rows of the matrix is problematic, because the data is not properly aligned in memory.

The *lower packed format with full diagonal blocks* (FP) stores the matrix in blocks, where each block is stored by columns, as shown in Figure 3c. The size of the blocks, i.e., the number of columns in each block, is denoted as `nb`. A typical size of the blocks is `nb` $= 128$.

The *lower-blocked hybrid format* (H) stores the matrix in packed format (i.e., without the upper triangular entries) by blocks of columns, with each block stored by rows, as shown in Figure 3d. This format was introduced in [3] and is particularly suited to perform dense factorization.

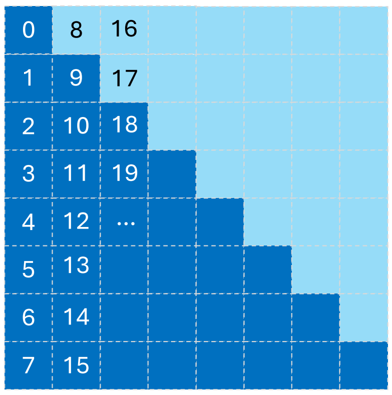
The *lower-blocked hybrid format with full diagonal blocks* (FH) is similar to format H, but the upper triangular entries of each block are stored explicitly, as shown in Figure 3e.

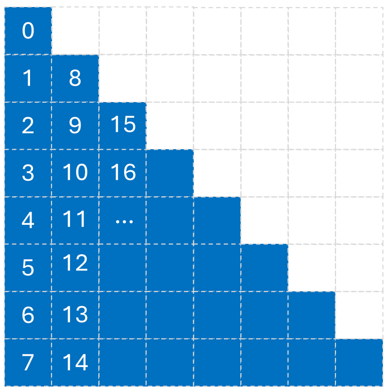
Notice that formats F, FP and H are used to perform dense factorizations. Formats P and FH are only used to store matrices.

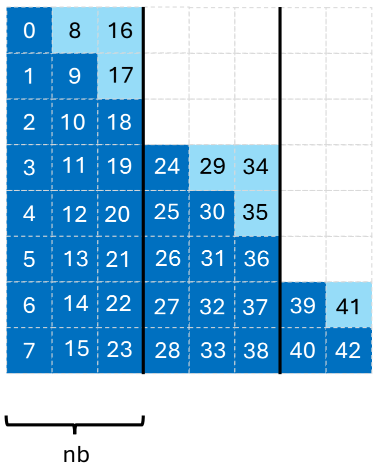
3.1 Access to the data

When assembling the frontal matrices of a supernode, it is required to access some specific locations of the matrix. If we want to access entry (p, q) of a certain

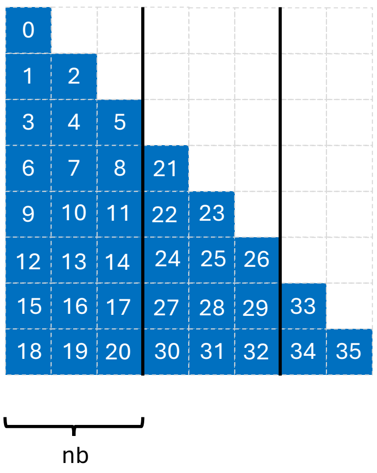
Figure 2: Different storage formats: white squares are not stored, light blue squares are stored but not used, dark blue square are stored and used. The numbers indicate the address of the entry within the array.


(a) Format F

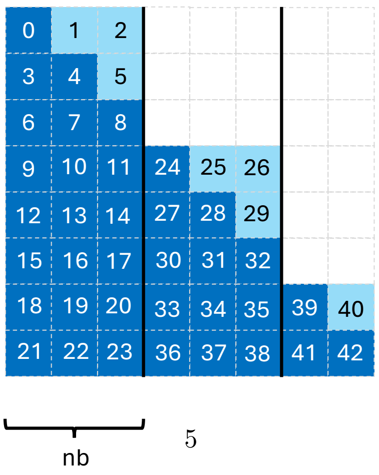

(b) Format P



(c) Format FP



(d) Format H



(e) Format FH

matrix, this is stored in different locations of the array, depending on the format used. We indicate the location within the array with square brackets.

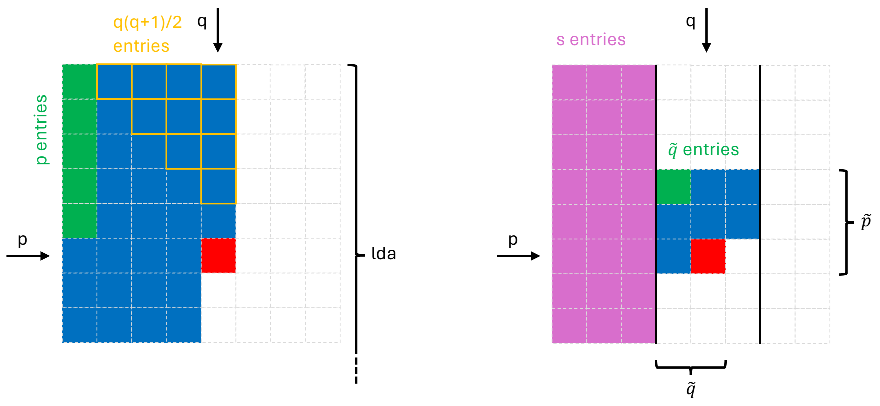
Notice that the location within the array of a certain entry corresponds to the number of entries that are stored *before* it, without counting the entry itself (numbering starts from zero). In Figure 3, we show entry (p, q) in red; the entries stored before it are coloured.

- Format F: looking at Figure 3, on the left, we need an offset of p entries to align with the proper row (in green) and then we need to skip q columns, i.e., $\texttt{lda} \cdot q$ entries (in blue). Therefore, entry (p, q) is found at location $[p + \texttt{lda} \cdot q]$.
- Format P: we use the same reasoning as for format F; however, the upper triangular entries (shown with yellow border) are not stored. Therefore, we subtract $q(q + 1)/2$ entries, so that entry (p, q) is found at location $[p + \texttt{lda} \cdot q - q(q + 1)/2]$.
- Format FH: define $(\tilde{p}, \tilde{q})$ as the row and column index within the current block of columns (they can be found from (p, q) by subtracting the number of columns in the previous blocks), and s as the starting location of the current block. Looking at Figure 3, on the right, we need an offset of s entries (in pink), to align with the proper block. We then need an offset of $\tilde{q}$ entries to align with the proper column (in green) and then we need to skip $\tilde{p}$ rows, i.e., $\texttt{nb} \cdot \tilde{p}$ entries (in blue). Therefore, entry (p, q) is found at location $[s + \tilde{q} + \texttt{nb} \cdot \tilde{p}]$.
- Format FP: a similar reasoning as for format FH gives that entry (p, q) is stored at location $[s + \tilde{p} + \texttt{nb} \cdot \tilde{q}]$.
- Format H: a similar reasoning as for format FH can be used, skipping the entries in the upper triangular part.

If instead we need to access the diagonal or off-diagonal entries of a specific column, we proceed as follows:

- Format F: for each column, we skip a number of entries equal to the index of the column. The next entry is the diagonal one, followed by all the off-diagonal ones.
- Format P: for each column, the first entry if the diagonal one, followed by all the off-diagonal ones.
- Format FP: For each block, and for each column of the block, we skip a number of entries equal to the local column number $\tilde{q}$. The next entry is the diagonal one, followed by all the off-diagonal ones.
- Format H: For each block, we read the first `jb` rows; they contain the off-diagonal entries and then the diagonal one. After these rows, we read the remaining rows, which contain only off-diagonal entries.
- Format FH: similar to format H, we just need to ignore the zeros at the end of the first `jb` rows.

Figure 3: Access to entry (p, q) of the matrix in formats F and P (left), and lower-blocked hybrid format with full diagonal blocks (right)



3.2 Conversions

Converting from one format to another is usually done with the help of a temporary buffer.

For example, to convert from format P to format H, with a matrix of size n, we use a buffer of size $n \times$ `nb`. We copy each column from the lower-packed array into the buffer, leaving space in between columns to properly align the rows. We then copy from the buffer to the final location, row by row. Both these operations can be done using `_copy`.

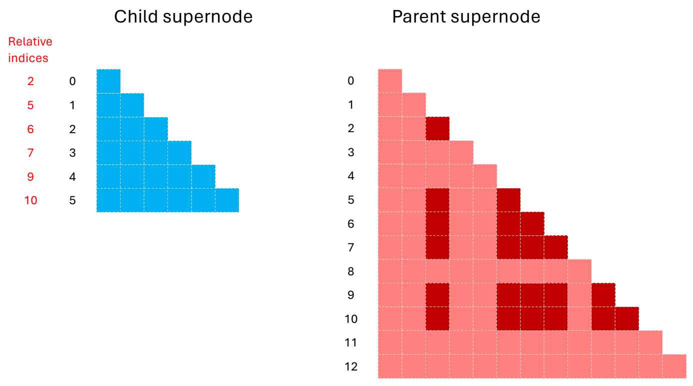
4 Assembly of the contributions

One of the most expensive operations during a multifrontal factorization is the so-called *extend-add* operations, in which the Schur complement from a child supernode is added to the frontal matrix of the parent supernode. To perform this operations efficiently, local indices are stored that indicate where the entries of the Schur complement of a given supernode will be summed in the frontal matrix of the parent supernode.

In Figure 4, the child supernode in blue has size 6; its parent supernode in red has size 13. The child supernode stores local relative indices (shown in red) that indicate the position where to sum its entries into the frontal matrix of the parent supernode (shown in dark red on the right).

It often happens that there are sequences of consecutive local indices; in Figure 4 for example, there is a sequence of three consecutive indices $5, 6, 7$. This means that the assembly of some of the entries of the child supernode into the parent can be done following a specific pattern. Identifying these patterns is fundamental to speed-up the assembly process: summing entries from the child to the parent one by one can be very slow; the same operation can instead be performed very efficiently using the BLAS level 1 subroutine `_axpy`, if the entries to be summed are evenly spaced. The most common occurrence of this is

Figure 4: Assembly of the contribution from a child supernode into the frontal matrix of its parent supernode



when the local indices are consecutive. Experiments show that this phenomenon occurs often in real matrices. The phenomenon of having a long sequence of local indices with equal spacing larger than one (e.g., the sequence $4, 7, 10, 13$ has equal spacing equal to 3) is much less likely to happen.

4.1 Assembly by columns

In the case of formats F, P and FP, the assembly can be done by columns: given a column of the child, the local indices indicate which column of the parent should receive its contribution. Then, every time there is a long sequence of consecutive relative row indices, `_axpy` can be used to sum a chunk of column from the child to the parent. This is depicted in Figure 5, on the left.

Notice that assembling in this way requires a single pass through the columns of the child and parent.

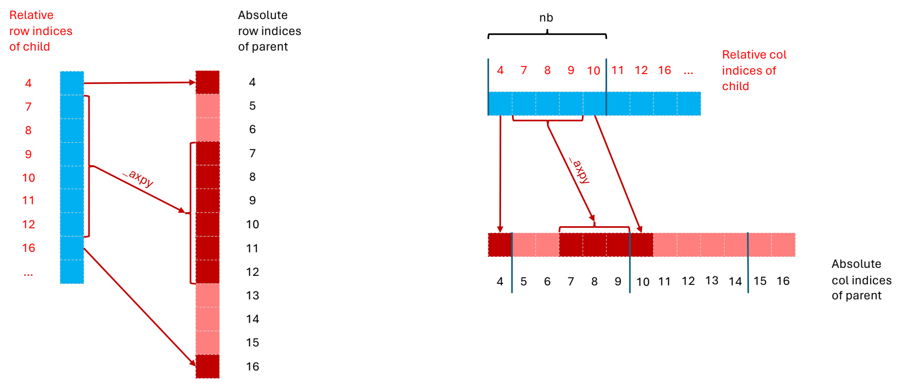
4.2 Assembly by rows

In the case of formats H and FH, the assembly should be done considering one block of columns of the child supernode at a time. Within the block, the assembly should be done by rows. In this way, only one pass through the Schur complement of the child is needed; however, the pattern of memory accesses for the parent supernode may be more irregular.

For example, in Figure 5, on the right, a sequence of 6 consecutive local indices is found $7, 8, 9, 10, 11, 12$. This can be fully exploited when assembling by columns; however, when using the lower-blocked hybrid format, we need to consider the edge of the block of columns of the child supernode. Indeed, the entries that correspond to local indices $7, 8, 9, 10$ are stored contiguously in memory, while the entries corresponding to local indices $11, 12$ are stored in another block of columns. This reduces the length of data that can be assembled using `_axpy` from 6 to 4 entries.

We also need to consider the edge of the block of columns in the parent supernode, where the data is going to be summed. Indeed, the entries that correspond to local indices $7, 8, 9$ are stored contiguously in the parent supernode,

Figure 5: Assembly of contribution by columns (left) and by rows (right).

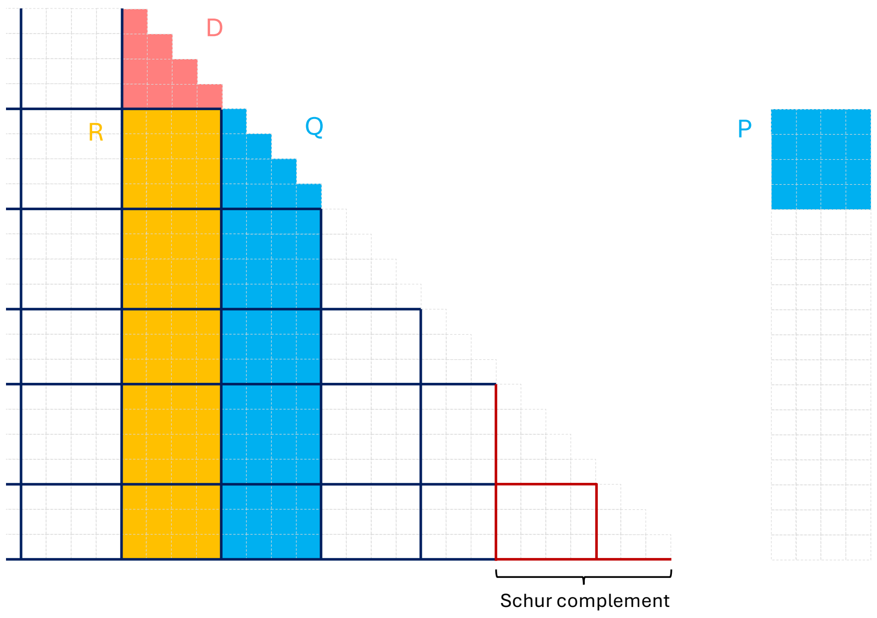

Relative row indices of child
4
7
8
9
10
11
12
16
...
_axpy
Absolute row indices of parent
4
5
6
7
8
9
10
11
12
13
14
15
16
nb
4 7 8 9 10 11 12 16 ...
Relative col indices of child
_axpy
4 5 6 7 8 9 10 11 12 13 14 15 16
Absolute col indices of parent

while the other entries belong to another block of columns of the parent. This reduces further the length of data that can be assembled using `_axpy` to 3 entries.

Notice also that to assemble the current row of the current block of columns of the child, three blocks of columns of the parent need to be loaded into memory. Some of these blocks are probably loaded and evicted from cache multiple times during the assembly. Therefore, it is not true any more that a single pass through the data structure of the parent is needed; multiples passes, in irregular order, may be needed.

Assembling by rows is thus slower than assembling by columns. However, this approach provides other savings when using the lower-blocked hybrid format to perform the dense partial factorization.

Figure 6: Factorization of block j.



5 Factorization with blocked formats

We present the operations that need to be performed to compute the factorization using the blocked formats FP, H and FH. Formats F is not a good choice to perform the dense factorization because it uses too much memory; format P is not a good choice either, because the rows of the matrix are not properly aligned.

We present the operations assuming that the diagonal block explicitly stores the upper zeros, in order to align the data. In the case of format H, this requires making a temporary copy of the diagonal block with the data properly aligned.

We show the BLAS calls that should be used for format FP (which accesses the data by columns) and for formats H and FH (which access the data by rows); the BLAS calls are very similar, but the matrices have to be considered transposed for formats H and FH.

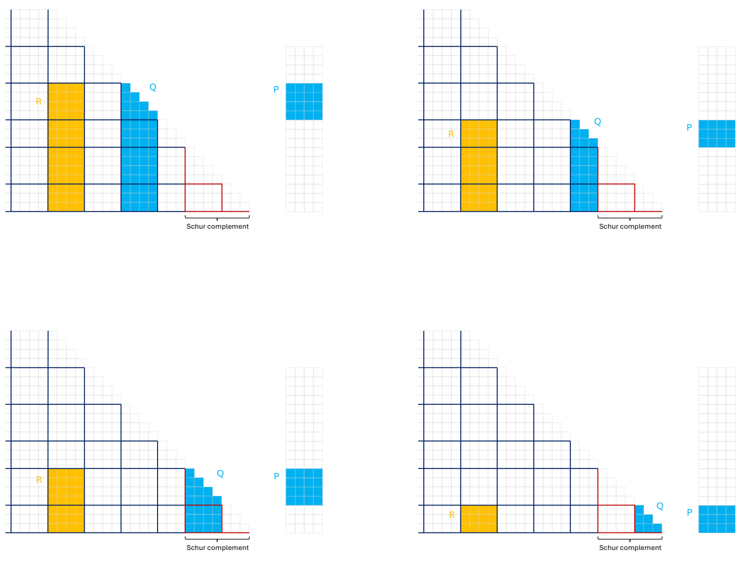
Information about how to access the correct part of the matrix using the various formats is provided later.

Looking at Figure 6, we are computing the block of columns j. We define blocks D and R as shown in the figure. All blocks of columns include `nb` columns, apart from the last one that may be smaller; we denote by `jb` the number of columns in block j. Notice that block D is the diagonal block of the current block of columns. It is not $\mathcal{D}$, the diagonal matrix in the $\mathcal{L}\mathcal{D}\mathcal{L}^T$ factorization.

We proceed as follows:

- Factorize block D.
  This is done using a factorization kernel that employs BLAS level 2 operations. The resulting block D stores the entries of $\mathcal{D}$ on the diagonal and

Figure 7



does not store the ones on the diagonal of $\mathcal{L}$.

- Solve $\boldsymbol{R} \leftarrow \boldsymbol{R}\boldsymbol{D}^{-\boldsymbol{T}}$.
  This is done using `_trsm(R, L, T, U, ...)` for format FP and `_trsm(L, U, T, U,...)` for formats H and FH.

- Make a copy of the block R into temporary storage, using `_copy`.

- Divide each column of R by the corresponding pivot in D. This is done using `_scal`.

- Update $\boldsymbol{Q} \leftarrow \boldsymbol{Q} - \boldsymbol{R}\boldsymbol{P}^{\boldsymbol{T}}$, for all block of columns on the right of the current one. Here P is taken from the temporary copy of R.
  This is done using `_gemm(N, T, ...)` for format FP and `_gemm(T, N, ...)` for formats H and FH.

Notice that the update involving blocks Q and P is done for all blocks of columns remaining, including for the Schur complement. The location of blocks P, Q and R changes depending on which block of columns is updated, as shown in Figure 7.

6 Access to the blocks

We now show the memory locations where the blocks D, R, P and Q are stored, for the formats FP and H. Format FH combines elements of both FP and H, and it is not shown.

We assume that there is an array `dstart` that contains the location in memory of the start of each diagonal block. Notice that it can be computed recursively as

$$\texttt{dstart}_j = \texttt{dstart}_{j-1} + \texttt{nb} \cdot (n - \texttt{nb} \cdot (j-1)), \quad \texttt{dstart}_0 = 0$$

for format FP and

$$\texttt{dstart}_j = \texttt{dstart}_{j-1} + \texttt{nb} \cdot (n - \texttt{nb} \cdot (j-1)) - \texttt{nb} \cdot (\texttt{nb}-1)/2, \quad \texttt{dstart}_0 = 0.$$

for format H.

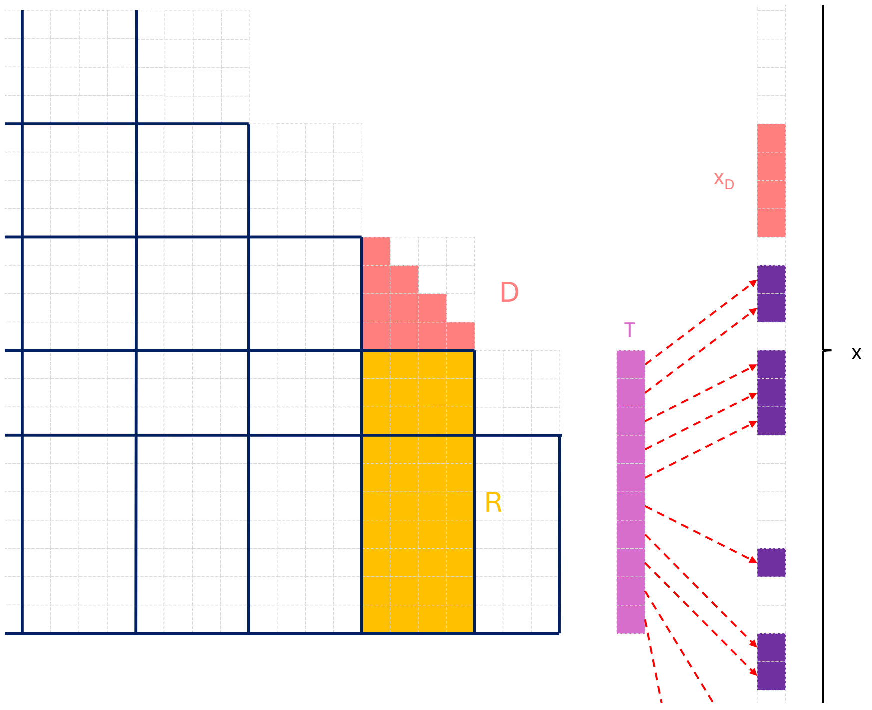
Then, block D is stored at location $\texttt{dstart}_j$. The location of block R is at an offset of `jb` entries from D, for format FP, and $\texttt{jb}(\texttt{jb}+1)/2$ entries from D, for format H.

Blocks Q can be accessed using the entry in `dstart` corresponding to the block of columns being updated. Block P and the correct portion of block R can be accessed with an offset that can be easily calculated for each format.

7 Solve with blocked formats

We describe the procedure to perform the forward solve. We want to compute $\mathcal{L}^{-1}x$, for a given vector x. We assume to store the result in place in x.

Figure 8: Data for forward solve.



We assume to iterate through the supernodes in order; for each supernode, we iterate through the blocks of columns in order. Looking at Figure 8, for a given block of columns the operations to be performed are the following:

- Solve $x_D \leftarrow D^{-1}x_D$.
  This is done using `_trsv(L, N, U, ...)` for format FP and `_tpsv(U, T, U, ...)` for format H.
- Compute $T = Rx_D$, in a temporary buffer T.
  This is done using `_gemv(N, ...)` for format FP and `_gemv(T, ...)` for format H.
- Subtract the entries of T from the correct location in x, depending on the row indices of the supernode.

We do not describe the procedure for the diagonal solve $\mathcal{D}^{-1}x$.

We describe the procedure to perform the backward solve $\mathcal{L}^{-T}x$. We assume to iterate through the supernodes in reveres order; for each supernode we iterate through the blocks of columns in reverse order. For a given block of columns the operations to be performed are the following:

- Collect the entries from the correct location in x into T, depending on the row indices of the supernode.
- Update $x_D \leftarrow x_D - R^T T$.
  This is done using `_gemv(T, ...)` for format FP and `_gemv(N, ...)` for format H.
- Solve $x_D \leftarrow D^{-T} x_D$.
  This is done using `_trsv(L, T, U, ...)` for format FP and `_tpsv(U, N, U, ...)` for format H.

8 Pivoting strategies

In order to obtain a robust factorization, some form of pivoting is required. To avoid changing the data structures and to avoid additional fill-in, we limit the pivoting within a supernode. If a good candidate pivot cannot be found, regularization is used to lift the pivot. We can further limit the pivoting to be within a block of columns of a supernode, to simplify things.

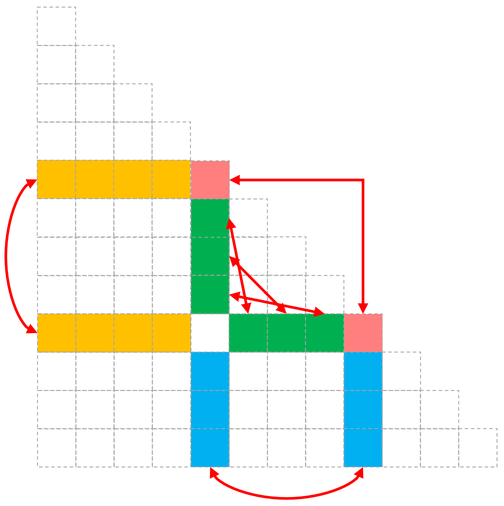
8.1 Swapping columns

In order to perform pivoting, we need a strategy to swap columns. During the factorization of a diagonal block, some strategy is used to decide that two columns should be swapped, in order to select a better pivot.

To achieve this, we need to swap rows and columns in the diagonal block. This requires swapping rows in the already factorized portion of the diagonal block. It would be possible to only swap the columns within the portion of the block still to factorize, but keeping the whole block consistent allows us to use BLAS with it.

Figure 9 shows the operations required to swap the rows and columns of the diagonal block. The yellow, blue and green parts need to be swapped with each other; this is achieved with three calls to `_swap`. The two pivots can be exchanged manually.

Figure 9: Operations to swap rows and columns within a block.



We store the order of swaps in a vector `swaps`. If pivot i is exchanged with pivot j, then we set `swaps[i] = j`. Notice that we only ever swap the current pivot with a later one, so that $i < j$. We need to use `swaps` to swap the columns in the blocks of columns below the diagonal block, once its factorization is complete. We also need `swaps` to perform the solve operation. Indeed, we need to permute the portion of right-hand side corresponding to the diagonal block

(x_D in pink in Figure 8) before performing the forward, backward or diagonal solve with that block, and unpermute it afterwards.

Given the vector `swaps`, we can apply the corresponding swaps by going through it from left to right. We can undo the swaps by going through it from right to left. Given `swaps = [1 3 2 3 4]` and the vector `v = [a b c d e]`, we go through the five entries of `swaps` starting from the left:

- Entry `swaps[0]` is not equal to 0, so a swap happened. We swap entry 0 and entry `swaps[0]`, to obtain `[b a c d e]`.
- Entry `swaps[1]` is not equal to 1, so a swap happened. We swap entry 1 and entry `swaps[1]`, to obtain `[b d c a e]`.
- Entry `swaps[2]` is equal to 2, so no swap happened. The same is true for entries 3 and 4. The swaps are completed.

To undo the permutation, we go through swaps from right to left.

- Entry `swaps[4]` is equal to 4, so no swap happened. The same is true for entries 3 and 2.
- Entry `swaps[1]` is not equal to 1, so a swap happened. We swap entry 1 and entry `swaps[1]`, to obtain `[b a c d e]`.
- Entry `swaps[0]` is not equal to 0, so a swap happened. We swap entry 0 and entry `swaps[0]`, to obtain `[a b c d e]`.

8.2 2×2 pivots

When using a 2×2 pivot, we store the two diagonal entries on the diagonal of block D, as for 1×1 pivots. We store an explicit zero as the entry that links the two columns, so that we can use `_trsm` to perform the solve with $\mathcal{L}$. We store the $(1, 2)$ entry of the pivot in another location.

Define d_1 and d_2 as the two diagonal entries of the pivot and e as the off-diagonal entry. Then the inverse of the 2×2 pivot is given by

$$\begin{bmatrix} d_1 & e \\ e & d_2 \end{bmatrix}^{-1} = \begin{bmatrix} d_2 & -e \\ -e & d_1 \end{bmatrix} \cdot \frac{1}{d_1 d_2 - e^2}$$

Given two columns $[c_1, \quad c_2]$ that need to be solved with pivots 1 and 2, if the pivots are 1×1, then the columns become $[c_1/d_1, \quad c_2/d_2]$. If instead the pivot is 2×2, then the two columns are combined together and become

$$\left[\frac{d_2}{d_1 d_2 - e^2} c_1 - \frac{e}{d_1 d_2 - e^2} c_2, \qquad \frac{d_1}{d_1 d_2 - e^2} c_2 - \frac{e}{d_1 d_2 - e^2} c_1 \right].$$

Similarly, entries of the right-hand side are combined together when doing the diagonal solve with $\mathcal{D}$.

References

[1] *Netlib BLAS documentation.* https://netlib.org/lapack/explore-html/de/d6a/group___blas___top.html.

[2] *Netlib BLAS overview.* https://www.netlib.org/blas/.

[3] B. S. Andersen, J. A. Gunnels, F. G. Gustavson, J. K. Reid, and J. Wasniewski, *A fully portable high performance minimal storage hybrid format cholesky algorithm*, ACM Transactions on Mathematical Software, 31 (2005), pp. 201–227.

[4] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Transactions on Mathematical Software, 9 (1983), pp. 302–325.

[5] J. W. H. Liu, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Review, 34 (1992), pp. 82–109.