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

Filippo Zanetti

October 25, 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.

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 $\mathcal{D}$.

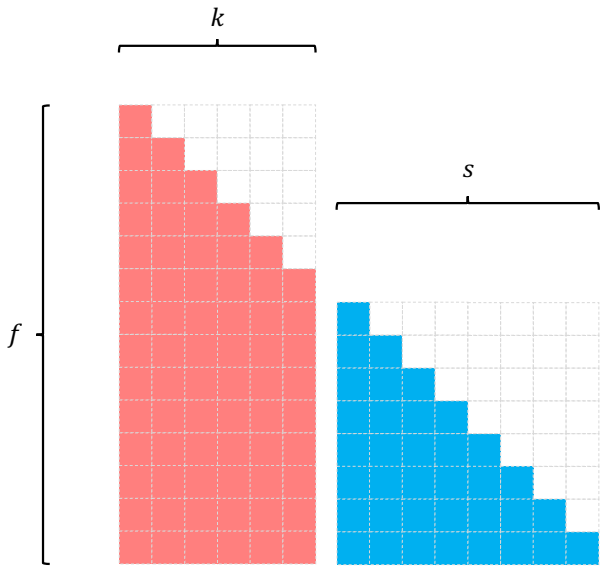
The blue part instead does not undergo factorization and stores the Schur complement that is passed to the parent in the tree. Given the two different purposes of these two parts of the matrix, given their different lifespan, and given that we assume to use completely static data structures (no pivoting 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.

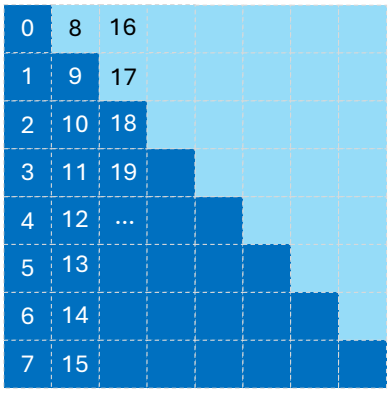
The *lower-blocked hybrid format with full diagonal blocks* (FH) is similar to format H, but the upper triangular entries of each block are stored explicitly, as shown in Figure 3e.

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

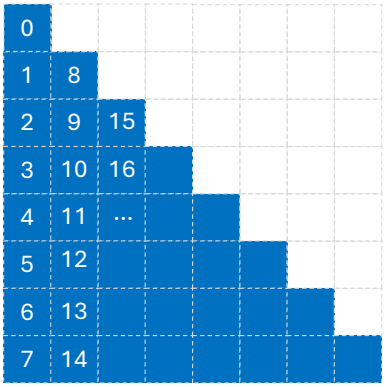
3.1 Access to the data

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

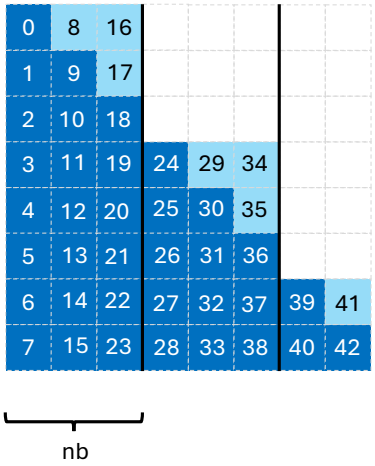
Figure 2: Different storage formats: white squares are not stored, light blue squares are stored but not used, dark blue square are stored and used. The numbers indicate the address of the entry within the array.



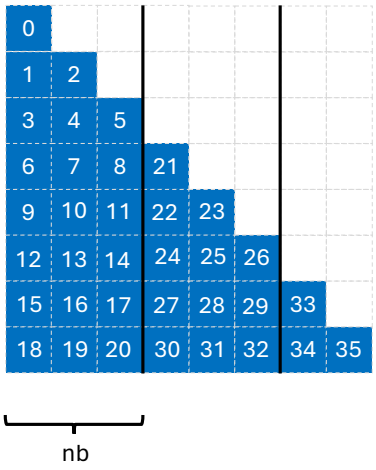
(a) Format F



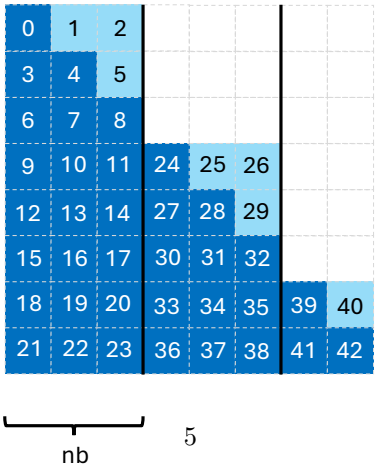
(b) Format P



(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.

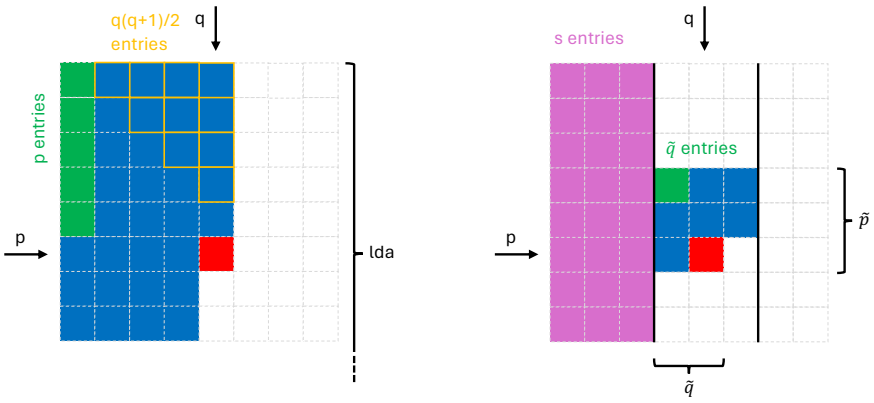
Notice that the location within the array of a certain entry corresponds to the number of entries that are stored *before* it, without counting the entry itself (numbering starts from zero). In Figure 3, we show entry (p, q) in red; the entries stored before it are coloured.

- 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`.

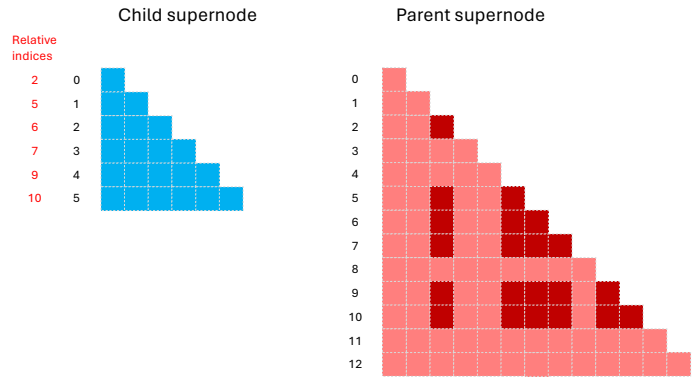
4 Assembly of the contributions

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.

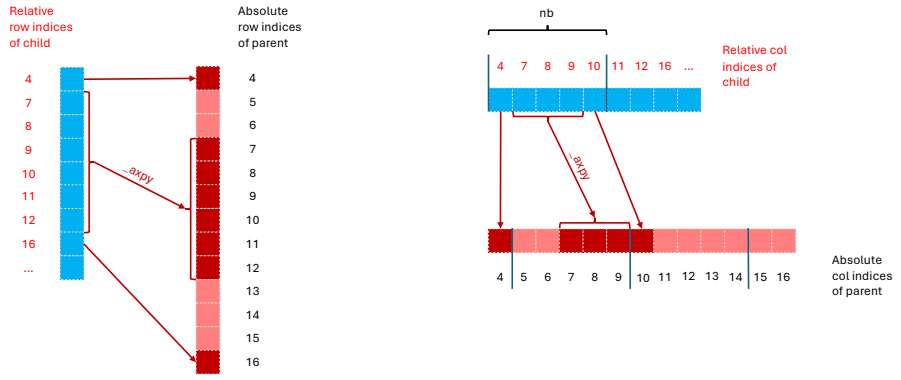
4.2 Assembly by rows

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).



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.

5 Operations with format F

We present the operations needed to perform the factorization with format F. This format is not very practical, because it uses much more memory than necessary. However, it is more intuitive to understand the logic of the factorization. This makes it easier to understand the other formats.

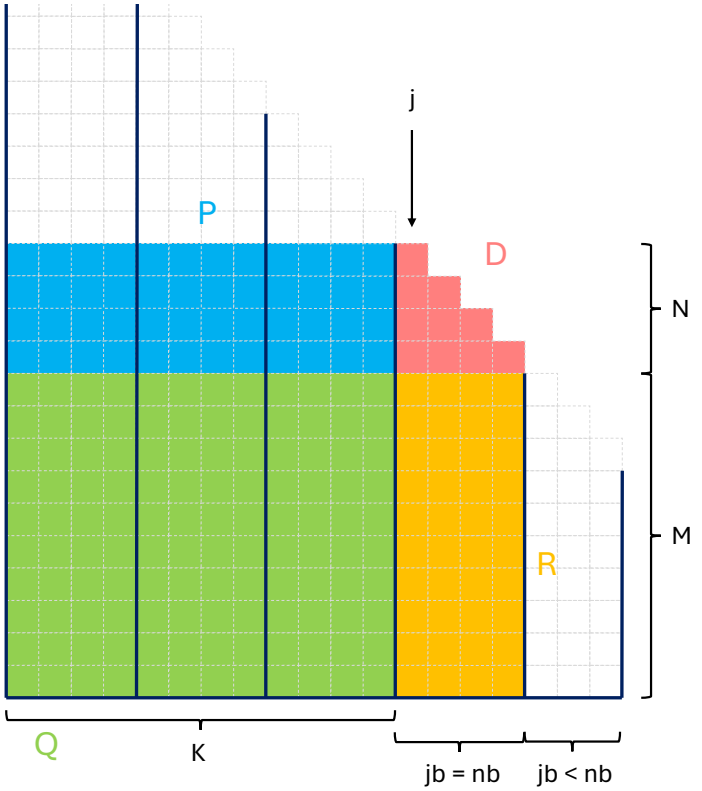
5.1 Accessing data with format F

A matrix A of size $n \times n$ can be stored in format F in an array of size at least n^2. We assume that the matrix is stored in an array with leading dimension `lda`.

To perform the dense partial factorization, we consider blocks of columns. We consider in the blocks only the columns that undergo factorization (and not the ones that are part of the Schur complement). The number of columns in a given block is denoted by `jb`; for all blocks except the last one, `jb` is equal to a fixed quantity `nb`; the last block may contain fewer than `nb` columns, as shown in Figure 6.

We now focus on a specific block of columns, shown in red and yellow in the figure. We assume that the first column of the block is j. We define the following quantities

Figure 6: Blocks of matrix in format F.



- Size of the block, $N =$ `jb`.
- Number of rows below the block, $M = n - j -$ `jb`.
- Number of columns already factorized, $K = j$.

Notice that, since we use format F, the entry (p, q) of matrix A can be found at location $[p + \texttt{lda} \cdot q]$ of the array.

We can identify four parts that are relevant when doing the factorization of this block. We call these parts "blocks", however notice that this has a different meaning than the block of columns that we are considering.

- Block D, shown in red, of size $N \times N$; its starting location is $[j + \texttt{lda} \cdot j]$.
- Block P, shown in blue, of size $N \times K$; its starting location is $[j]$.
- Block Q, shown in green, of size $M \times K$; its starting location is $[j + N]$.
- Block R, shown in yellow, of size $M \times N$; its starting location is $[j + N + \texttt{lda} \cdot j]$.

All these blocks are accessed with leading dimension `lda`.

Notice that blocks D and R are modified, while blocks P and Q are only read.

5.2 Factorization with format F

We now show the operations to be computed for the current block of columns, in order to obtain the partial factorization of the matrix A, and the BLAS functions needed to achieve them.

- We start by making a copy of block P, with each column multiplied by the corresponding pivot, in a temporary buffer T. This is achieved using `_copy` and `_scal`.
- Update of diagonal block, $D \leftarrow D - PT^T$: `_gemm(N, T, ...)`.
- Factorization of diagonal block: this is achieved by a purposely developed factorization kernel, that calls BLAS level 2 functions.
- Update of columns, $R \leftarrow R - QT^T$: `_gemm(N, T, ...)`.
- Solve with diagonal block, $R \leftarrow RD^{-T}$: `_trsm(R, L, T, U, ...)`. Notice that in this operations the diagonal of D is assumed to be made of ones, hence the fourth parameter U for `_trsm`.
- Scale each column of R by the corresponding pivot in D, using `_scal`.

The Schur complement can be computed in different ways. In order to use `_syrk` instead of `_gemm`, we can make temporary copies of the columns that influence the Schur complement, multiplied by the square root of the absolute value of the corresponding pivot. We store these copies in two different arrays, depending on the sign of the pivot. We then use two calls to `_syrk`, one with $\alpha = -1$ and one with $\alpha = 1$, in order to compute the contributions of the columns that correspond to positive and negative pivots.

5.3 Solve with format F

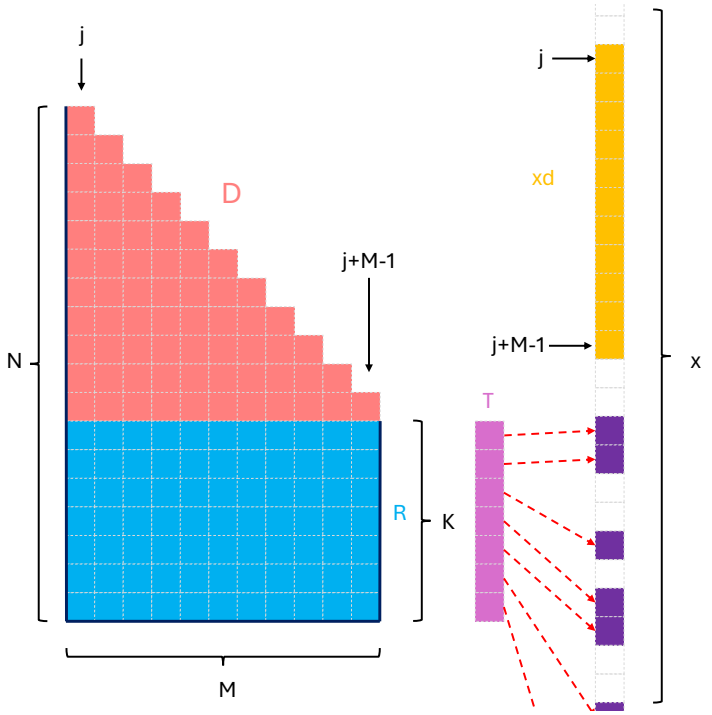
We now focus on how to perform efficiently the forward solve, when the columns of each supernode are stored in format F. We do not show the equivalent operations for the backward solve, because they are very similar but in a different order. We do not show the operations to perform the diagonal solve, as this is relatively simple to implement, following the guidelines on how to access the diagonal entries in Section 3.1.

We assume that we want to perform the forward solve, for a given supernode, with a vector x, as shown in Figure 7. Each supernode is stored as a dense matrix, together with a list of indices that indicate the corresponding rows in the sparse factor $\mathcal{L}$.

We define the following quantities:

- j, index of the starting column of the supernode.
- N, size of front.
- M, number of columns in the supernode.
- K, size of the Schur complement for this supernode, $K = N - M$.

Figure 7: Forward solve procedure in format F.



Notice that the column of the supernode correspond to the columns j to $j+M-1$ of $\mathcal{L}$. The corresponding entries of x, denoted as xd and shown in yellow, are directly affected by the diagonal block of the supernode (D, shown in red). The supernode also affects other entries of x, shown in violet and determined by the indices of the rows of the supernode.

A temporary buffer T of size K (shown in pink) is needed to perform the solve. This buffer needs to be scattered into the correct location at the end of the procedure.

The operations to perform for the forward solve are the following.

- Solve with diagonal block, $xd \leftarrow D^{-1}xd$: `_trsv(L, N, U, ...)`.
- Update other entries of x, $T \leftarrow Rxd$: `_gemv(N, ...)`.
- Scatter and subtract content of T from x, based on the row indices of the supernode.

To perform solve with multiple right-hand sides, BLAS level 3 functions can be used instead of BLAS level 2.

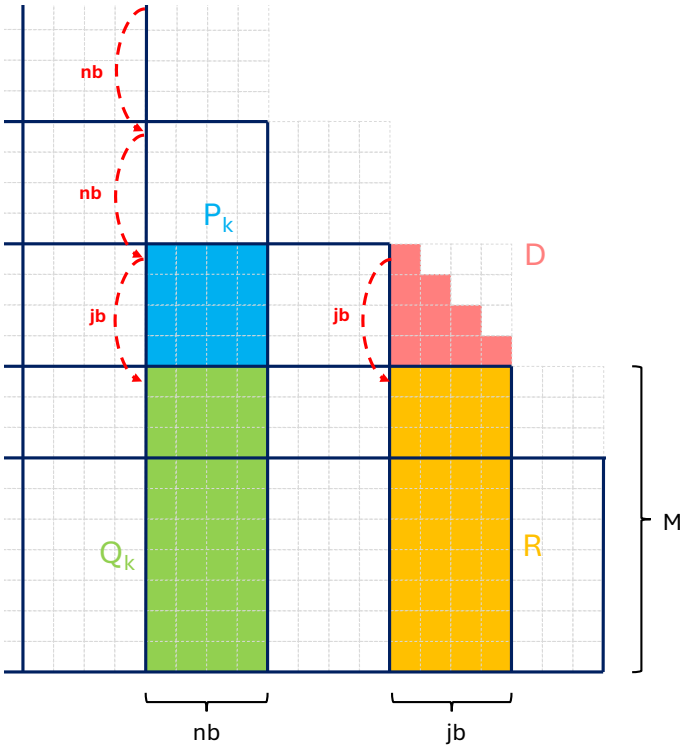
6 Operations in format FP

We consider the factorization of the frontal matrix in format FP.

6.1 Accessing data in format FP

We assume that the blocks used to store the matrix have size `nb`, apart from the last one, which may be smaller than `nb`.

Figure 8: Blocks of matrix in format FP.



With reference to Figure 8, we consider the block of columns shown in red and yellow; we assume that it is the $j-$th block of columns. We use the index $k < j$ to identify a previous block of columns. Define as M the number of rows below the current diagonal block. We can identify the following blocks:

- Block D, shown in red, of size `jb` $\times$ `jb`.
- Block R, shown in yellow, of size `jb` $\times M$.
- Block P_k, $\forall k < j$, shown in blue, of size `nb` $\times$ `jb`.
- Block Q_k, $\forall k < j$, shown in green, of size `nb` $\times M$.

Differently from format F, we now have to consider multiple P and Q blocks, for each block of columns that precedes the current block.

To describe the location in memory of these blocks, we define an array `dstart` that contains the location in memory of the start of each diagonal block. Then

- Block D is stored at location [$\texttt{dstart}_j$], with leading dimension $n - j \cdot$ `nb`.

- Block R is stored at location [$\texttt{dstart}_j$ + $\texttt{jb}$], with leading dimension $n - j \cdot \texttt{nb}$.
- Block P_k is stored at location [$\texttt{dstart}_k + \texttt{nb} \cdot (j-k)$], with leading dimension $n - k \cdot \texttt{nb}$.
- Block Q_k is stored at location [$\texttt{dstart}_k + \texttt{nb} \cdot (j-k) + \texttt{jb}$], with leading dimension $n - k \cdot \texttt{nb}$.

In Figure 8, we show the jumps needed to find the memory locations just described, starting from the locations of the diagonal blocks.

Notice that `dstart` can be computed recursively as

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

As before, blocks D and R are modified, while the others are only read.

6.2 Factorization with format FP

The sequence of operations is the following.

- Make a copy of block P_k, with each columns scaled by the corresponding pivot, in a temporary buffer T. This is done using `_copy` and `_scal`.
- Update of diagonal block
  $\forall k < j,\ D \leftarrow D - P_k T^T$: `_gemm(N, T, ...)`.
- Update of columns
  $\forall k < j,\ R \leftarrow R - Q_k T^T$: `_gemm(N, T, ...)`.
- Factorization of diagonal block using kernel.
- Solve with diagonal block
  $R \leftarrow RD^{-T}$: `_trsm(R, L, T, U, ...)`.
- Scale each column of R with the corresponding pivot, using `_scal`.

6.3 Schur complement with format FP

The computation of the Schur complement is very similar to the case for format H, with slightly different BLAS arguments and different position of the blocks. See Section 7.3.

6.4 Solve with format FP

The solve procedure is very similar to the one for format F. However, the procedure should be applied to each block of columns of each supernode.

7 Operations with format H

We consider the factorization of the frontal matrix in format H only (with packed diagonal blocks). The variation with full diagonal blocks is used only to store the Schur complement at the end of the factorization, in one of its versions.

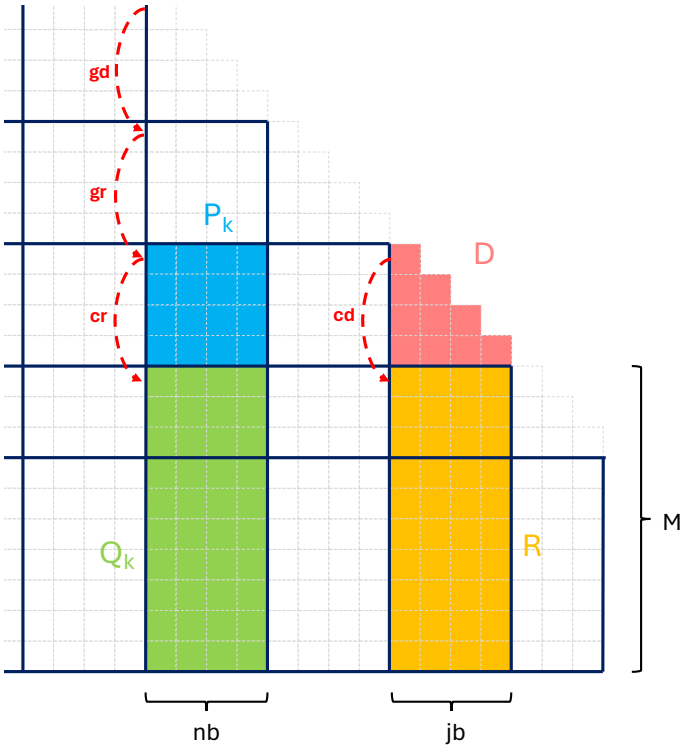
7.1 Accessing data with format H

We assume that the blocks used to store the matrix have size `nb`, apart from the last one, which may be smaller than `nb`.

We can identify the following types of blocks:

- A generic diagonal block has $\texttt{gd} = \texttt{nb}(\texttt{nb}+1)/2$ entries.
- A generic rectangular block (usually below a generic diagonal one) has $\texttt{gr} = \texttt{nb}^2$ entries.
- The current diagonal block has $\texttt{cd} = \texttt{jb}(\texttt{jb}+1)/2$ entries.
- The current rectangular block (to the left of the current diagonal one) has $\texttt{cr} = \texttt{nb} \cdot \texttt{jb}$ entries.

Figure 9: Blocks of matrix in format H.



With reference to Figure 9, we consider the block of columns shown in red and yellow; we assume that it is the $j-$th block of columns. We use the index $k < j$ to identify a previous block of columns. Define as M the number of rows below the current diagonal block. We can identify the following blocks:

- Block D, shown in red, of size $\texttt{jb} \times \texttt{jb}$.
- Block R, shown in yellow, of size $\texttt{jb} \times M$.

- Block P_k, $\forall k < j$, shown in blue, of size $\mathtt{nb} \times \mathtt{jb}$.
- Block Q_k, $\forall k < j$, shown in green, of size $\mathtt{nb} \times M$.

As for format FP, we now have to consider multiple P and Q blocks, for each block of columns that precedes the current block.

Notice that each of the blocks described above is stored contiguously in memory; this was not true in the case of formats F and FP. To describe the location in memory of these blocks, we define an array `dstart` that contains the location in memory of the start of each diagonal block. Then

- Block D is stored at location $[\mathtt{dstart}_j]$, with leading dimension `jb`.
- Block R is stored at location $[\mathtt{dstart}_j + \mathtt{cd}]$, with leading dimension `jb`.
- Block P_k is stored at location $[\mathtt{dstart}_k + \mathtt{gd} + \mathtt{gr} \cdot (j-k-1)]$, with leading dimension `nb`.
- Block Q_k is stored at location $[\mathtt{dstart}_k + \mathtt{gd} + \mathtt{gr} \cdot (j-k-1) + \mathtt{cr}]$, with leading dimension `nb`.

In Figure 9, we show the jumps needed to find the memory locations just described, starting from the locations of the diagonal blocks.

Notice that `dstart` can be computed recursively as

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

As before, blocks D and R are modified, while the others are only read.

7.2 Factorization with format H

To exploit the efficiency of the BLAS functions, before starting the operations for the current block of columns, a copy of block D into full format by rows is made. We assume that this copy is stored in an array $\tilde{D}$.

Notice that, since the blocks are stored by rows instead of columns, we need to perform the BLAS functions calls as if the matrices were transposed and the upper triangles were used. For example, the update of the diagonal block would be $D \leftarrow D - P_k T^T$, called as `_gemm(N, T, ...)` in normal circumstances; with the hybrid format instead we perform $\tilde{D} \leftarrow \tilde{D} - (T^T)^T (P_k^T)$, called as `_gemm(T, N, ...)`.

The sequence of operations is the following.

- Copy D into $\tilde{D}$ in full format by rows.
- Make a copy of block P_k, with each columns scaled by the corresponding pivot, in a temporary buffer T. This is done using `_copy` and `_scal`.
- Update of diagonal block
  $\forall k < j,\ \tilde{D} \leftarrow \tilde{D} - P_k T^T$, performed as $\tilde{D} \leftarrow \tilde{D} - (T^T)^T (P_k^T)$:
  `_gemm(T, N, ...)`.
- Update of columns
  $\forall k < j,\ R \leftarrow R - Q_k T^T$, performed as $R^T \leftarrow R^T - (T^T)^T (Q_k^T)$:
  `_gemm(T, N, ...)`.

- Factorization of diagonal block using kernel.
- Solve with diagonal block
  $R \leftarrow RD^{-T}$, performed as $R^T \leftarrow \tilde{D}^{-T}R^T$: `_trsm(L, U, T, U, ...)`.
- Scale each column of R with the corresponding pivot, using `_scal`.
- Copy $\tilde{D}$ into D in packed format by rows.

7.3 Schur complement with format H

We compute the Schur complement one block of columns at a time; for each block of columns, we compute the effect of all the columns already factorized on the current block of columns of the Schur complement.

For a given block of columns s of the Schur complement, we consider any block of already factorized columns j that affects block s. Looking at Figure 10, we define the following:

- `ns`, size of the Schur complement.
- N, number of columns in block s.
- `jb`, number of columns in block j.
- M, number of rows in block s.
- Block P_j, shown in blue, of size $N \times$ `jb`. Its location can be computed as shown before; notice however that there is a rectangular block that could have a smaller size than `gr` (shown in violet).
- Block Q_j, shown in green, of size $(M - N) \times$ `jb`. Its location is computed starting from the location of P_j.

The operations to compute the block of columns s of the Schur complement are performed on a temporary buffer T of size `ns` $\times$ `nb` in full format. The buffer contains blocks D, of size $N \times N$, shown in red, and R, of size $(M - N) \times N$, shown in yellow.

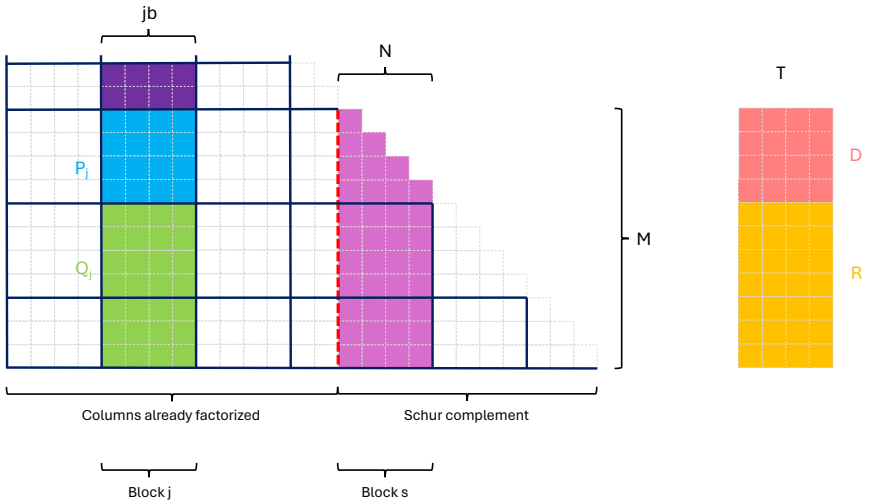
The content of the buffer is then copied in the corresponding position of the Schur complement array (shown in pink).

The operations to be performed are:

- Make a copy of block P_j, with each columns scaled by the corresponding pivot, in a temporary buffer T. This is done using `_copy` and `_scal`.
- Update of diagonal block
  $\forall j,\ D \leftarrow D - P_j T^T$, performed as $D \leftarrow D - (T^T)^T(P_j^T)$:
  `_gemm(T, N, ...)`
- Update of columns
  $\forall j,\ R \leftarrow R - Q_j T^T$, performed as $R^T \leftarrow R^T - (T^T)^T(Q_j^T)$:
  `_gemm(T, N, ...)`.

The result, in the buffer, is the block of columns of the Schur complement stored in format FH. We have two options now:

Figure 10: Blocks of the Schur complement in hybrid format.



- We copy the result from the buffer to the Schur complement array, transforming it into format FP. This is a more friendly format to be assembled into the frontal matrix of the parent; however, this copy and transformation are expensive.
- We leave the result in format FH (in this case we do not need the buffer, as the result can be computed in place). This is less friendly for the assembly process, because the data is stored by rows within the blocks, but does not require any copy.

7.4 Solve with format H

If the columns of the $\mathcal{L}$ factor are stored in lower-blocked hybrid format, the forward solve procedure is very similar to the one described for the full format. However, instead of applying it to each supernode, we apply it to each block of columns of each supernode. Moreover, since the diagonal blocks are not stored in full format, `_tpsv(U, T, U, ...)` should be used instead of `_trsv(L, N, U, ...)`.

In the case of multiple right-hand sides, the function `_tpsm` does not exist, and so the diagonal block needs to be copied into full format, in order to use `_trsm`. Further efficiency can be obtained by storing the right-hand sides in a special format, as described in [3].

7.5 Hybrid vs Full format

There are four options for how to compute the factorization

- Compute both the frontal matrix and the clique in format F.
  The BLAS functions need to access data not contiguous in memory, but the assembly process is done by columns and is very fast. No copies are needed.

- Compute both the frontal matrix and the clique in format FP.
  The BLAS function still access data not contiguous in memory and the assembly is fast. The memory requirement is considerably smaller than for format F.
- Compute the frontal matrix in format H and the clique in format FP.
  The BLAS functions are more efficient, because they access data contiguously. Transforming the clique to format P is expensive, but the assembly process is still done by columns and is very fast.
- Computing the frontal matrix in format H and the clique in format FH.
  The BLAS functions are as efficient as the previous option, but the clique does not need to be copied into a different format. The assembly process needs to be done by rows and is slower.

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.