A

The treatment of sparse matrices

This appendix is intended to readers interested in the numerical implementation of the methods discussed in the book. It aims to review how matrices originating from the discretization of a differential problem by the finite element method (or finite volumes/differences) can be handled by a computer program. Indeed, in real life computations the discretization of PDEs by any of the cited methods leads to the solution of large systems of linear equations whose matrix is sparse. And the efficient storage of sparse matrices requires to adopt special techniques.

We will also recall some practical strategies for dealing with Dirichlet-type conditions in finite element codes.

As far as numerical techniques for solving linear systems are concerned, the reader may refer to the vast literature on the matter, for instance [GV96, QSS00, Saa03, Sha08].
A matrix is *sparse* if it "contains a large number of zeros". Better said, if the number of non-null entries (also called non-zeros) is $O(n)$. This means that the average number of non-zero entries in each row is bounded independently from n. Indeed, what is important is that the location of the zero elements is known a-priori, so we can avoid reserving storage for them, as explained in Appendix. A non-sparse matrix is also said *full*: obviously here the number of non-zero elements is $O(n^2)$.

A.1 Storing techniques for sparse matrices

The distribution of non-zero elements of a sparse matrix may be described by the *sparsity pattern*, defined as the set $\{(i,j) : \mathrm{A}_{ij} \neq 0\}$. Alternatively, one may consider the matrix graph, where nodes i and j are connected by an edge if and only if $\mathrm{A}_{ij} \neq 0$.

A representation of the pattern can be obtained through the MATLAB command `spy` (see Fig. A.1 for an example). The sparsity of a finite element matrix is a direct consequence of the small-support property of the finite element basis functions. Thus, the sparsity pattern depends on the topology of the adopted computational grid, on the kind of finite element chosen and on the indexing of the nodes. It is completely known before the actual construction of the matrix. Therefore, the matrix can be stored efficiently by excluding the terms that are a-priori zero.

The use of adequate storage techniques for sparse matrices is fundamental, especially when dealing with large-scale problems typical of industrial applications. Let us make an example. Suppose we want to solve the Navier-Stokes equations on a two-dimensional grid formed by 10.000 vertexes with finite elements P^2-P^1(and this is a rather small problem!). By using the results of Exercise 2.2.4 and the relations of (2.10) we deduce that the number of degrees of freedom is around 10^5 for the pressure and 4×10^5 for each component of the velocity. The associated matrix will then be 90000×90000. If we had to store all 8.1×10^9 coefficients, using the usual double precision (8 bytes to represent each floating point number), around 60 *Gigabytes* would be necessary! This is too much even for a very large computer. Modern operative systems are able to employ areas larger than the available RAM by using the technique of virtual memory (also known as "*paging*"), whcih saves part of the data on a mass storage device (typically the hard disc). However, this does not solve our problem because paging is extremely inefficient. In case of a three-dimensional problem the situation becomes even worse, since the number of degrees of freedom grows very rapidly as the grid gets finer, and nowadays it is customary to deal with millions of degrees of freedom.

Therefore to store sparse matrices efficiently we need data formats that are more compact than the classical table (*array*). The adoption of sparse formats, though, may affect the speed of certain operations. Indeed with these formats we cannot access or search for a particular element (or group of elements) directly, as happens with `array`, where the choice of two indexes i and j allows to determine directly where in the memory the wanted coefficient A_{ij} is located[1].

On the other hand, even if the operation of accessing an entry of a matrix in sparse format (like a matrix-vector multiplication) turns out to be less efficient, by adopting a sparse format we will nevertheless access only non-zero elements, thus eschewing futile operations. That is why, in general, the sparse format is preferable in terms of computing time as well, as long as the matrix is sufficiently sparse (and this is usually the case for finite element, finite volume and finite difference descretizations).

We can distinguish different kinds of operations on a matrix, the most important ones being:

1. *accessing a generic element:* this is sometimes called "random access";

[1] The efficiency in accessing and browsing an *array* actually depends on the way the matrix is organized in the computer memory and on the operating system's ability to use the processor's *cache* memory proficiently. To go into details is beyond the scope of the present book, but the interested reader may refer to [HVZ01], for example.

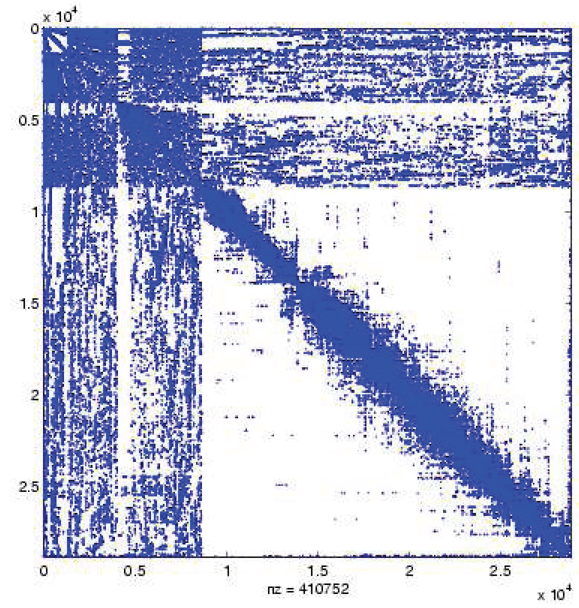

Fig. A.1. Sparsity pattern for the matrix of a finite-element discretization of a vectorial problem in three dimensions. The number of elements is around 9×10^8, of which only about 4.2×10^4 non-zero. The pattern was obtained with MATLAB's `spy` command

2. *accessing the elements of a whole row:* important when multiplying a matrix by a vector;
3. *accessing the elements of a whole column*, or equivalently, of a row in the transpose matrix. This is relevant for operations such as symmetrizing the matrix after imposing Dirichlet conditions, as we will see in Section A.2.
4. *adding a new element to the matrix pattern:* this is no major issue if one builds the pattern at the beginning, and does not change it during the computations. It becomes critical if the pattern is not known beforehand or it can change throughout the computations. This happens for instance with grid adaptation techniques.

It is important to characterize formats for sparse matrices by the computational cost of these operations and by how the latter depends on the matrix size. That different formats exist for sparse matrices is due, historical reasons aside, precisely to the fact that there is no format that is simultaneously optimal for all above operations, and be at the same time efficient in terms of storage capacity.

In the sequel we will review the most common formats, including those used by MATLAB, FreeFem and some important linear algebra libraries, like

Sparsekit [Saa90], PetSC [BBG$^+$01], UMFPACK [DD97, Dav04] or `AztecOO` [HBH$^+$05, Her04]. For completeness, we also mention a document describing the Harwell-Boeing format [DGL92]. This is not so much a format for storage on a computer, but rather one meant for writing and reading sparse matrices on files. The reader interested in software and tools for operating on large matrices and related examples and bibliography may refer to the *Matrix Market* web site (`http://math.nist.gov/MatrixMarket`).

We have to remark that matrices generated by finite-element codes have certain fixed features "by construction",

1. even if the matrix is not symmetric, its sparsity pattern is. This is because an element A_{ij} is in the pattern if the intersection of the support of the basis functions associated to nodes i and j has a non-zero measure. And this is obviously a symmetric property;
2. diagonal elements are in most of the cases non-zero, so we can assume they belong to the pattern.

In fact, in a finite element matrix (i, j) belongs to the pattern if nodes i and j share a common element. Note that by using this definition it is possible that we allocate storage for elements which may eventually be zero: we exclude only elements which are *a-priori* zero. We also have to point out that not always the

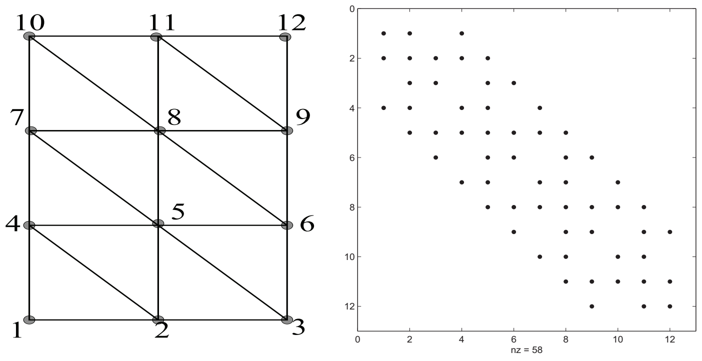

Fig. A.2. Example of grid with linear finite elements and pattern of the associated matrix. Note that the pattern depends on the numeration chosen for the notes

matrices of concern are square: think of the matrices D and D^T of Ch. 7. Some of the formats we will describe are suitable only for the square case, and cannot be employed in general.

As reference example we will consider the matrix that might have arisen using linear finite elements on the grid of Fig. A.2, left. The pattern of this matrix is shown on the right. In particular, the matrix A in "full" format (*array*) could be

$$
\mathrm{A} = \begin{bmatrix}
101. & 102. & 0. & 103. & 0. & 0. & 0. & 0. & 0 & 0. & 0. & 0. \\
104. & 105. & 106. & 107. & 108. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 109. & 110. & 0. & 111. & 112. & 0. & 0. & 0. & 0. & 0. & 0. \\
113. & 114. & 0. & 115. & 116. & 0. & 117. & 0. & 0. & 0. & 0. & 0. \\
0. & 118. & 119. & 120. & 121. & 122. & 123. & 124. & 0. & 0. & 0. & 0. \\
0. & 0. & 125. & 0. & 126. & 127. & 0. & 128. & 129. & 0. & 0. & 0. \\
0. & 0. & 0. & 130. & 131. & 0. & 132. & 133. & 0. & 134. & 0. & 0. \\
0. & 0. & 0. & 0. & 135. & 136. & 137. & 138. & 139. & 140. & 141. & 0. \\
0. & 0. & 0. & 0. & 0. & 142. & 0 & 143. & 144. & 0 & 145. & 146. \\
0. & 0. & 0. & 0. & 0. & 0. & 147. & 148. & 0. & 149. & 150. & 0. \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 151. & 152. & 153. & 154. & 155. \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 156. & 0 & 157. & 158.
\end{bmatrix}, \tag{A.1}
$$

where the values of the matrix elements are not relevant for this discussion, and indeed they have been just made up to allow to identify them easily.

In the sequel n will always be the matrix' size, nz the number of non-zero entries. Moreover, we will adopt the convention of indexing entries of matrices and vectors (arrays) starting[2] from 1. To estimate how much memory the matrix occupies we have assumed an integer occupies 4 bytes, and a real number (floating point representation) 8 bytes (double precision)[3]. Hence storing the matrix of Fig. A.1, which has $n = 12$ and $nz = 58$, would require $12 \times 12 \times 8 = 1152$ bytes if stored as an *array*. At last, A_{ij} will denote the entry of the matrix A on row i and column j.

A.1.1 The COO format

The format by *coordinates*, (*COO*rdinate format) is conceptually the simplest, even though it is poorly efficient in terms of both memory space and access to a generic element.

This format uses three arrays which we denote `I`, `J` and `A`. The first two describe the pattern: precisely, in the generic kth place of `I` and `J` we store the row and column indexes of the coefficient whose value is stored at the same position in `A`. Hence `I`, `J` and `A` all have as many elements as the number of non-zero elements nz.

In this way the space occupied is $(4 + 4 + 8) \times nz$ bytes. For the matrix A in (A.1), a possible coding in COO format reads

[2] Some programming languages (e.g., C and C++) number arrays from 0, so to use this convention it suffices to subtract 1 from our indexes.

[3] On a 64 bit architecture also the integers may use up 8 bytes.

$$\begin{aligned}
I = [&1,1,1,2,2,2,2,2,3,3,3,3,4,4,4,4,4,5,5,5,5,5,5,5,6,6,6,6,6,\\
&7,7,7,7,7,8,8,8,8,8,8,8,9,9,9,9,9,10,10,10,10,11,11,11,11,\\
&11,12,12,12\,]
\end{aligned}$$

$$\begin{aligned}
J = [&1,2,4,1,2,3,4,5,2,3,5,6,1,2,4,5,7,2,3,4,5,6,7,8,3,5,6,8,9,4,\\
&5,7,8,10,5,6,7,8,9,10,11,6,8,9,11,12,7,8,10,11,8,9,10,11,12,\\
&9,11,12]
\end{aligned} \tag{A.2}$$

$$\begin{aligned}
A = [&101.,102.,103.,104.,105.,106.,107.,108.,109.,110.,111.,112.,113.,\\
&114.,115.,116.,117.,118.,119.,120.,121.,122.,123.,124.,125.,126.,\\
&127.,128.,129.,130.,131.,132.,133.,134.,135.,136.,137.,138.,139.,\\
&140.,141.,142.,143.,144.,145.,146.,147.,148.,149.,150.,151.,152.\\
&,153.,154.,155.,156.,157.,158.]\,,
\end{aligned}$$

requiring 928 bytes. Clearly, the three arrays can contain the same elements in different order. This format does not guarantee rapid access to an element, nor to rows or columns. Finding the generic element of the matrix from the row and column indexes normally requires a number of operations proportional to nz. In fact, it is necessary to go through all elements of `I` and `J` until one hits those indexes, using expensive comparison operations. There is a way, though at a higher storing price, to use specific techniques to store the indexes in special search data structure, and reduce the cost to $\mathcal{O}(\log_2(nz))$.

The operation of multiplying a matrix and a vector can be done directly, by running through the elements of the three arrays. We show a possible code for the product $\mathbf{y} = \mathrm{A}\mathbf{x}$ using the MATLAB syntax[4]

```
y=zeros(nz,1);
for k=1:nz
 i=I(k); j=J(k);
 y(i)=y(i) + A(k)*x(j);
end
```

The additional cost of this operation, compared to the analogue for a full matrix, depends essentially on *indirect addressing*: accessing `y(i)` requires first of all to access `I(k)`. Furthermore, the access and update of the arrays `x` and `y` does not proceed by consecutive elements, a fact that would greatly reduce the possibility of optimizing the use of the processor's *cache*. Recall, however, that now we operate only on non-zero elements, and that, in general, $nz << n^2$.

An advantage of this format is that it is easy to add a new element to the matrix. In fact, it is enough to add a new entry to the arrays `I`, `J` and `A`. That is why COO is often used when the pattern is not known a priori. Obviously, to do so, it is necessary to handle memory allocation in a suitable dynamical way.

[4] We use MATLAB syntax for simplicity, yet normally these operations are coded in a compiled language, like C of FORTRAN, for efficiency reasons.

A generalization of the COO format uses an *associative array* or a *hash table* to construct the map $(i, j) \to \mathrm{A}_{ij}$. In the C++ language, for instance, one may adopt the `map` container of the standard library for this purpose [SS03]. In some linear algebra packages, like `Eigen` (`xxx.eigen.org`) or `Aztecoo` for instance, it is possible to build a sparse matrix dynamically, and in this case a "COO-type" format is used internally. When one knows that the pattern will not change anymore, the matrix can be "finalized" with a conversion to a more efficient, yet more static, format.

A.1.2 The *skyline* format

The format called *skyline* was among the first used to store matrices arising from the method of finite elements. The idea, schematically depicted in Fig. A.3, left, is to store the blue area formed, on each row, by the elements between the first and last non-zero coefficient. It is clear that this forces to store some null entries, in general. This extra cost will be small if the matrix has non-zero entries clustered around the diagonal. Indeed, algorithm have been developed, the most known one being probably the CuthillâĂŞMcKee algorithm, to cluster non zero elements by permuting the rows and columns of the matrix, see [Saa03] for details,

We will explain how the format this applies to symmetric matrices, and then generalize it.

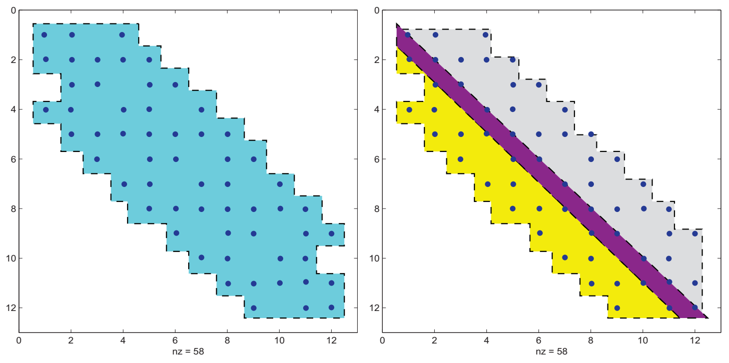

Fig. A.3. Skyline of a matrix (right). On the left, the decomposition in lower triangular, diagonal and upper triangular parts

Skyline for symmetric matrices. If a matrix is symmetric we can store only its lower triangular part (diagonal included). Or we can store the diagonal on an auxiliary array and treat the off-diagonal entries separately. The latter choice has the advantage of allowing the direct access to the diagonal elements. If we opt for this solution, the skyline format is given by three arrays, `D`, `I` and `AL`. In `D` we store diagonal entries, in `AL` all skyline elements in succession and row-wise (except the diagonal), i.e. the light-coloured area. This can clearly include null coefficients. The kth component of the array `I` tells (technically, "*points to*") where the $(k+1)$th row of `AL` begins: all elements of `AL` from position `I(k)` to `I(k+1)-1` are the off-diagonal elements belonging to row $k+1$, in increasing column order. In this way the first row is not stored, since it only has the diagonal element, `I(k)` points to the first non-zero element on the $(k+1)$th row, `I(k+1)-1` points to the element $A_{k+1,k}$, and the difference `I(k+1)- I(k)` tells how many off-diagonal elements on row $k+1$ belong to the skyline. A quick computation allows to verify that the first non-zero element on row $k>1$ is the one on column `k-I(k) - I(k-1)`.

Supposing, for example, we wish to store the symmetric matrix constructed from the lower triangular part in A as of (A.1), corresponding to the `Matlab` instructions `tril(A)+tril(A,-1)'`. Then

$$D = [101., 105., 110., 115., 121., 127., 132., 138., 144., 149., 154., 158.]$$

$$I = [1, 2, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]$$

$$AL = [104., 109., 113., 114., 0., 118., 119., 120., 125., 0., 126., 130., 131., 0., \\ 135., 136., 137., 142., 0., 143., 147., 148., 0., 151., 152., 153., 156., 0., 157.]\,.$$

Note that in the nth place of the array `I` we have left a pointer at the beginning of an hypothetical second row. In this way `I(n)` -1 is the total number of elements in the skyline. Moreover, we can compute the number of skyline elements using `I(n)` $-$ `I(n-1)`, for the last row as well. The product $\mathbf{y} = \mathrm{A}\mathbf{x}$ is computed as follows (MATLAB syntax),

```
y=D.*x;
for k=2:n
 nex = I(k)-I(k-1);
 ik  = I(k-1):I(k)-1;
 jcol= k-nex:k-1;
 y(k)   = y(k)+dot(AL(ik),x(jcol));
 y(jcol)= y(jcol)+AL(ik)*x(k);
end
```

Observe the need to operate symmetrically on rows and columns to exploit the fact that only the lower triangular part was stored in `AL`.

As said, the memory needed to store the matrix in this format, depends on how effectively the skyline reproduces the actual pattern. In the case under scrutiny the array `AL` contains 29 real numbers, to which we add the fixed length n of the arrays `D` and `I`, in our case 12. The first has real numbers, the second integers, so storing our matrix requires 376 bytes. A direct comparison with the *COO* format is not possible as in the previous section's example the matrix was non-symmetric.

One can exploit the possible symmetry also with *COO* by storing only the lower triangular part (the multiplication algorithm between matrix and vector changes accordingly). In this case, with *COO* we would store 35 coefficients and use 560 bytes. So *skyline* apparently looks more convenient: but if the coefficients are not well clustered around the diagonal the memory space used by *skyline* would increase quickly as n increases.

Skyline for general matrices. As with non-symmetric matrices in the general case, a reasonable way to proceed is to split A into the diagonal D, the strictly lower triangular part E and strictly upper triangular part F . Using the `Matlab` syntax, these matrices would be defined as `D=diag(diag(A)); E=tril(A,-1); F=triu(A,1)`. As the pattern of A is symmetric, the *skyline* of E coincides with that of F^T, hence we will store E and F^T (and D) with the previous technique. In this way there is no need to duplicate the array `I`, this being the same for both triangular parts. Therefore we can use two arrays of length n, still denoted `D` and `I`, and two real-valued arrays of length equal to the skyline dimension, called `E` and `FT` (containing E and F^T respectively). In the example, this would necessitate of 608 bytes.

$$D = [101., 105., 110., 115., 121., 127., 132., 138., 144., 149., 154., 158.]$$

$$I = [1, 2, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]$$

$$E = [104., 109., 113., 114., 0., 118., 119., 120., 125., 0., 126., 130., 131., 0., \\ 135., 136., 137., 142., 0., 143., 147., 148., 0., 151., 152., 153., 156., 0., 157.]\,,$$

$$FT = [102., 106., 0., 107., 103., 116., 111., 108., 122., 0., 112., 0., 123., 117., \\ 133., 128., 124., 139., 0., 129., 0., 140., 134., 150., 145., 141., 155., 0., 146.]\,.$$

The product matrix-vector $\mathbf{y} = \mathrm{A}\mathbf{x}$ now reads

```
y=D.*x;
for k=2:n
 nex  = I(k)-I(k-1);
 ik   = I(k-1):I(k)-1;
 jcol = k-nex:k-1;
 y(k)   = y(k)+dot(E(ik),x(jcol));
 y(jcol)= y(jcol)+FT(ik)*x(k);
end
```

We should observe that in this format the access to diagonal entries is direct, and the cost of extracting a row is independent of the matrix' size. Indeed, the fact that the data relative to a row are stored consecutively in the memory allows the system to optimize the processor's *cache* memory when multiplying a matrix by a vector. In the example above `icol` and `ik` contain all indexes corresponding to the columns of row `k`, so the scalar product `dot(E(ik),x(jcol))` and the multiplication vector-constant `FT(ik)*x(k)` can be optimized[5].

The extraction of column is, vice versa, an expensive operation that requires many comparisons, and whose cost grows linearly in n.

Being able to access diagonal entries directly has certain advantages. For instance we will see that methods to impose essential boundary condition based on penalization (Section A.2.2) only need the access to diagonal elements.

A.1.3 The CSR format

The problem with the skyline format is that the memory used depends on the numeration of elements and is in general impossible to avoid the unnecessary storage of zero elements. Renumeration algorithm may be very inefficients for large scale problems.

For these reasons other formats have been developed that render memory space independent of the numeration of degrees of freedom. The format `CSR` (*Compressed Sparse Row*) is one of them, and can be seen as a compressed version of `COO` that renders it more efficient, but also as an improved *skyline*, that stores non-zero elements only. The format uses three arrays:

1. The real-valued array `A` of length nz, containing the non-zero entries of the matrix, ordered row-wise: in the example it coincides with the array `A` written in (A.2).[6]
2. The integer-valued array `J` of length nz, whose entry $J(k)$ indicates the column of the element `A(k)`. In our case it coincides with the `J` in (A.2).
3. The array `I` of length n containing "pointers" to the rows. Essentially, `I(k)` gives the position where the kth row in `A` and `J` begins, as shown in Fig. A.4.

In many practical applications, the array `I` is of length $n+1$ so that the number of non-zero entries on row k is always `I(k+1)-1-I(k)`. To make this hold the last element `I(n+1)` will contain $nz+1$ and in this way we also have that nz=`I(n+1)-I(1)`. The format CSR stores the matrix using $4\times(nz+n+1)+8\times nz$ bytes. For the example we have

$$I = [1, 4, 9, 13, 18, 25, 30, 35, 42, 47, 51, 56, 59] \tag{A.3}$$

while `J` and `A` are the same as in (A.2). Notice again that this particular fact is not general. With COO the order of the indexes in `I`, `J` can be arbitrary.
The memory space required for the example is of 748 bytes. However, the gain in storage requirement with this format becomes more evident with large sparse matrices.

[5] These operations are contained in the library BLAS (Basic Linear Algebra Subroutines), which furnishes highly optimized functions for some elementary matrix operations.

[6] With the format COO it is not necessary this array to be ordered row-wise

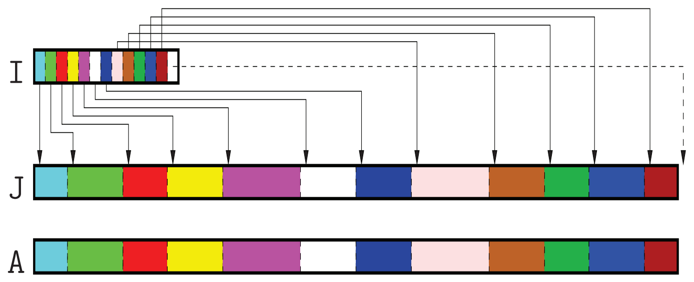

Fig. A.4. The format CSR. The picture refers to the numerical example discussed in the text. In CSR the elements of `I` point to `J` and `A`, respectively telling where the column indexes and the values of a given row begin

This format suits square and rectangular matrices alike, and allows a quick extraction the i-th row: it is sufficient to consider the elements of `A` lying between `I(i)` and `I(i+1)-1`. Less immediate is column extraction, which requires localizing on each row the values of `J` corresponding to the wanted column. If we adopt no particular ordering, the operation has a cost proportionate to nz. If, instead, column indexes of each row in `J` are ordered, e.g. in increasing order as in our example, with a binary-search algorithm the extraction cost for a column lowers; more precisely, it becomes proportional to $n \log_2(m)$, where m is the mean number of elements on each row. Analogously, the access to a generic element has normally a cost proportional to m, yet if we order columns it reduces to $\log_2 m$.

A further variant is to store in the first element of the part of `J` corresponding to a given row the index of the diagonal element. In this way `A(I(k))` provides the coefficient A_{kk} directly.

The matrix-vector product $\mathbf{y} = \mathrm{A}\mathbf{x}$ is given by

```
y=zeros(n,1);
% y=A(I(1:n)).*x if the  diagonal is stored first
for k=1:n
 ik=I(k):I(k+1)-1;
 % ik=I(k)+1:I(k+1)-1; if the  diagonal is stored
 %                     first
 jcol =J(ik); y(k)=y(k)+dot(A(ik),x(jcol));
end
```

A.1.4 The CSC format

Evidently, there is a corresponding format *CSC* (Compressed Sparse Column) that stores matrices by ordering them column-wise, so it is easy to extract a column as opposed to rows. Here the roles of vectors `I` and `J` is exchanged compared with the CSR format. It is the format preferred, for instance, by the `UMFPACK` library.

When performing matrix-times-vector operations with a sparse matrix ordered by columns it is preferable to compute the result as a linear combination of the columns of the matrix. Indeed, if $\mathbf{c}_i$ indicates the i-th column of matrix A we have that $\mathrm{A}\mathbf{x} = \sum_i x_i \mathbf{c}_i$. Therefore, the matrix-vector product $\mathbf{y} = \mathrm{A}\mathbf{x}$ on a CSC matrix may be computed as

```
y=zeros(n,1);
for k=1:n
  xcoeff=x(k);
  jk=I(k):I(k+1)-1;
  ik=J(jk);
  y(ik)=y(ik) + xcoeff * A(jk)';
end
```

A.1.5 The MSR format

The format MSR (*Modified Sparse Row*) is a special version of *CSR* for square matrices whose diagonal elements are always contained in the pattern (as it happens in general for matrices generated by finite elements). Diagonal entries can be stored in one single array, since their indexes are implicitly known from their position in the array. As for the *symmetric skyline*, only off-diagonal elements are stored in a special fashion, i.e. through a format akin to *CSR*.

In practice one uses two arrays, which we call `V` (Values) and `B` (Bind). In the first n entries of `V` we store the diagonal. The place $n+1$ in `V` is left with no significant value (the reason will become clear later). From place $n+2$ onwards off-diagonal elements are stored, row-wise. Hence `V` has length $nz+1$. The array `B` has the same length as `V`: from $n+2$ to $nz+1$ are the column indexes of the elements stored in the corresponding places in `V`; the first $n+1$ point to where rows begin in subsequent positions. Therefore `B(k)`, $1 \leq k \leq n$, contains the position *within the same array* `B`, and correspondingly in `V`, where the kth row begins to be stored (Fig. A.5, top). More exactly, column indexes of non-zero coefficients of row k will be stored between `B(B(k))` and `B(B(k+1))-1`, while the corresponding values ranges between `V(B(k))` and `V(B(k+1))-1`. The element `B(n+1)` plays the same role of `I(n+1)` in the format *CSR*: it points to a hypothetical row $n+1$. In this way `nz=B(n+1)-1`. The reason for sacrificing the element `V(n+1)` is now clear: one wants to set up an exact correspondence between the elements of `V` and `B`, starting from element $n+2$ till the last. The space needed is $12 \times (nz+1)$ bytes.

Concerning the example, the coding *MSR* reads as follows (the unused element in `V` is marked with $*$)

$$B = [14, 16, 20, 23, 27, 33, 37, 41, 47, 51, 54, 58, 60,\\ 2, 4, 1, 3, 4, 5, 2, 5, 6, 1, 2, 5, 7, 2, 3, 4, 6, 7, 8, 3, 5, 8, 9, 4, 5, 8, 10,\\ 5, 6, 7, 9, 10, 11, 6, 8, 11, 12, 7, 8, 11, 8, 9, 10, 12, 9, 11\ ]$$

$$V = [101., 105., 110., 115., 121., 127., 132., 138., 144., 149., 154., 158., *,\\ 102., 103., 104., 106., 107., 108., 109., 111., 112., 113., 114., 116., 117.,\\ 118., 119., 120., 122., 123., 124., 125., 126., 128., 129., 130., 131., 133.,\\ 134., 135., 136., 137., 139., 140., 141., 142., 143., 145., 146., 147., 148.,\\ 150., 151., 152., 153., 155., 156., 157.]$$

which occupies 708 bytes.

The format *MSR* turns out to be very efficient in memory terms. It is one of the most "compact" formats for sparse matrices, reason for which it is used in several linear algebra libraries dealing with large problems. As already mentioned, the drawback is that it only applies to square matrices.

The product matrix-vector is coded as

```
y=V(1:n).*x;
for k=1:n
 ik=B(k):B(k+1)-1;
 jcol =B(ik);
 y(k)=y(k)+dot(A(ik),x(jcol));
end
```

As for computational efficiency, its features are similar to those of *CSR*: whereas accessing rows is easy, extracting a column is more expensive an operation, for it requires finding the column index in the array B. Here, too, the cost of an extraction can be reduced to being proportional to $n \log_2 m$ by ordering the columns corresponding to each row and adopting a binary search algorithm (m is still the mean number of columns per row).

We present in the sequel a non-standard variant (that actually works for *CSR* as well), based on adding a third array that allows to access columns in a time lapse that is independent of the sparse matrix size and without a search on indexes (and hence without conditional branches).

A non-standard modification of *MSR*. The modification presented here has been adopted by the serial version of the finite-element library LIFEV ([lif10]), and exploits the fact that matrices coming from the finite-element method have a symmetric pattern. This means that if we run through off-diagonal elements on row k and detect that the coefficient A_{kl} is the pattern (i.e. is non zero), the pattern will also contain A_{lk}, in row l. If the position of A_{lk} in B (and V) is stored in a "twin" array of the part of B from $n+2$ to $nz+1$, then we have a structure yielding the elements of a given column. Let us call this array CB (Column Bind): to extract the column indexed k it is enough to read the elements of CB between B(k)-(n+1) and B(k+1)-1-(n+1) (subtracting $n+1$ *shifts* indexes, from those to which B points in V to those in CB). These elements point to the positions of B

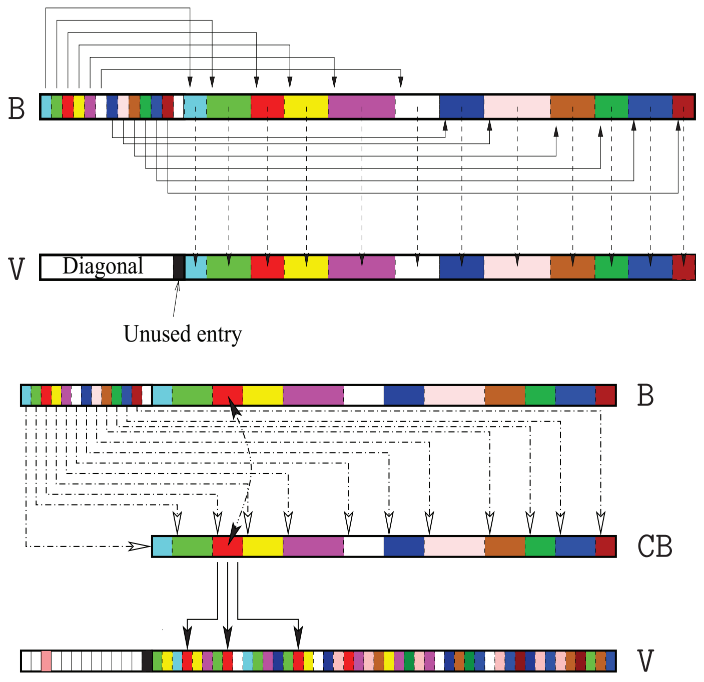

Fig. A.5. Above, the format MSR. The first components of B point (solid arrows) to column indexes contained in the second part of B, which in turn correspond to the array V (dashed arrows). Below: the modified *MSR* format: the array CB comes (dash-dotted arrows) from the first n elements of B. The elements of CB point to the elements of V belonging to a given column. For example, solid arrows denote in V the elements relative to the third column. The respective row indexes of these elements are located in the area corresponding to the second section of B (curved arrow). The elements of the third column are therefore the targets of the arrows, apart the one on the diagonal (which is highlighted in the first section of V)

and V where one can find the corresponding row indexes and the matrix values, respectively.

Basically, the first positions of B point to CB, which in turn points to the off-diagonal entries in B and V (Fig. A.5 bottom). This double-pointing system, though burdensome, allows to access the column of a sparse matrix at a cost that is independent of nz and without demanding conditional branches, provided it is programmed properly.

The structure `CB` in our case is

$$CB = [16, 23, 14, 20, 24, 27, 17, 28, 33, 15, 18, 29, 37, 19, 21, 25, 34, 38, 41, \\ 22, 30, 42, 47, 26, 31, 43, 51, 32, 35, 40, 48, 52, 54, 36, 44, 55, 58, 39, \\ 45, 56, 46, 49, 53, 59, 50, 57\,] \,.$$

The array `B` tells us that to extract the elements of the third column, say, we should find the pointers to the column elements in the positions between `B(3)-(n+1)=20-13=7` and `B(4)-1-(n+1)=22-13=9` in `CB` ($n + 1 = 13$ is the shift). At these places we read `17, 28, 33` corresponding to the positions in `V` of `106.,119.,125.`, that are precisely the off-diagonal elements of column number three.

Compared to *MSR*, the modified format requires storing $nz - n$ additional integers (the number of non-zero off-diagonal entries), so the memory is now $12 \times (nz + 1) + 4 \times (nz - n)$ bytes.

Again, the advantage of this sparse storage becomes important for larger sparse matrices, as the reader may verify easily.

A.2 Imposing essential boundary conditions

The demand for efficient storage of sparse matrices must come to terms with the need of accessing and manipulating the matrix itself. These operations are especially important to impose Dirichlet-type (essential) boundary conditions. In a finite-element code, in fact, the stiffness matrix is often generated ignoring essential boundary conditions, which are then introduced by modifying the algebraic system suitably. This happens because the assembling operation is characterized by several cycles in which it would not be efficient to introduce tests on the nature of a degree of freedom (whether boundary or not) and on the type of associated boundary condition.

Henceforth we shall denote by $\widetilde{\mathrm{A}}$ and $\widetilde{\mathbf{b}}$ the matrix and the source term *before* the essential boundary conditions are imposed.

A.2.1 Elimination of essential degrees of freedom

The way to impose Dirichlet conditions that is “closest to the theory” consists in eliminating freedom degrees corresponding to the nodes where the conditions should apply, since there the solution is known.

If k_D is the generic index of a Dirichlet node and g_{k_D} the (known) value of u_h at the node, eliminating the degree of freedom means that

1. the columns of index k_D of $\widetilde{\mathrm{A}}$ are erased, correcting the right-hand side. Better said, the rows of index $k_{nD} \neq k_D$ are “shortened” by eliminating all non-zero coefficients $A_{k_{nD}k_D}$, used to update the right-hand side to $\widetilde{b}_{k_{nD}} = \widetilde{b}_{k_{nD}} -$

$A_{k_{nD}k_D}g_{k_D}$. Therefore the columns corresponding to the Dirichlet nodes are truly *eliminated* from $\widetilde{\mathrm{A}}$;

2. the rows of index k_D in the matrix and right-hand side are erased from the system, eventually producing a square matrix A and a source **b** with size equal to the problem's effective number of freedom degrees.

Operation 1 coincides actually with lifting the boundary datum, cf. the discussion of Chapter 3.

The only advantage of this procedure is that we eventually have to solve a system with just the "true" unknowns of our problem. However is has many practical downsides. First, the complexity of the implementation, since in general the numbering of the Dirichlet nodes is arbitrary. Furthermore, it alters the pattern, which could be inconvenient in case we wanted to share it between several matrices to save memory space. This happens if the problem has several unknowns, each with its own boundary condition. Eventually, considering that normally one requires to have the solution *at all nodes* with the original numbering for post-processing, one must store the array that allows to recover it.

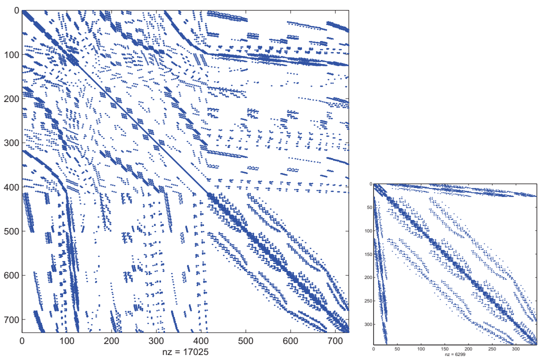

Fig. A.6. Elimination of Dirichlet freedom degrees on a 3D cube (from a real case). The matrix before the boundary conditions (left), and afterwards (right)

A.2.2 Penalization technique

If one rates techniques on how involved their programming is, at the opposite end of the scale to the above is the so-called node-based *penalization*. The basic idea is to add a term `hv` to the diagonal elements in $\widetilde{\mathrm{A}}$ matching the rows of index k_D relative to Dirichlet degrees, and correspondingly add the term $\mathtt{hv}g_{k_D}$ to the source element k_D.

In this way the equation relative to row k_D becomes

$$\sum_{j=1}^{N} \widetilde{a}_{k_D j} u_j + \mathtt{hv} u_{k_D} = \widetilde{b}_{k_D} + \mathtt{hv} g_{k_D}.$$

If the coefficient `hv` is sufficiently large (`hv` stands for *high value*), the effect of the perturbation is to make the equation an approximation of $\mathtt{hv}u_{k_D} = \mathtt{hv}g_{k_D}$, whose solution is, naturally, $u_{k_D} = g_{k_D}$. Indeed, if a_{K_d} denotes the maximum absolute value of matrix elements in row k_D and $a_{K_d}/\mathtt{hv} < \mathtt{eps}$, `eps` being the machine epsilon number[7], the computer will in fact "see" $\mathtt{hv}u_{k_D} = \mathtt{hv}g_{k_D}$. In this way we impose the wanted condition without changing the problem's dimension nor the matrix pattern.

This approach (adopted by the software FreeFem++) has simplicity as its asset: the only requirement is the access to diagonal elements. Its weak point rests on the fact that to have an accurate approximation of the boundary datum the value `hv` must be much large enough (`FreeFem` sets to 10^{30} by default). This, generally speaking, will degrade the matrix condition number, since it introduces eigenvalues of order `hv`. However, the new introduced eigenvalues are all clustered around this value, so the situation is better than what one may think at first sight.

Other (more complex) penalization techniques operates on the variational problem. Notably, we mention the Nitsche's method, which provides a penalty formulation which is consistent and maintains optimal convergence rate also for high order elements. More details on the Nitsche's method may be found in [Ste95].

A.2.3 "Diagonalization" technique

A third option, which neither alters the pattern nor necessarily introduces ill-conditioning for the system, is to consider the Dirichlet condition as an equation of the form $\alpha u_{k_D} = \alpha g_{k_D}$ to replace row k_D of the original system. Here, $\alpha \neq 0$ is a suitable coefficient, often taken equal to 1 or to the average of the absolute values of the elements of row k_D (to avoid degrading the condition number). This substitution is performed by setting to zero the row's off-diagonal elements except for the diagonal one, which is set to α, without modifying the sparsity pattern. Accordingly, the corresponding element in the right hand side is set to αg_{k_D}.

The operation requires access to the sole rows, so it is efficient in formats like *COO*, *CSR* or *MSR* (Fig. A.7 left). This approach, that we termed *diagonalization* (not to be confused with the usual meaning in linear algebra), is without doubt a good compromise between easy programming and control of the conditioning of the problem.

Its major fault is to destroy the symmetry of the matrix (if the original matrix was). If one wishes to keep that symmetry (for instance, in order to use a Cholesky decomposition), then it is necessary to modify the columns as well, and consequently the source term. A possible strategy to address this issue is explained in the next section. When using a Krylov-based iterative method of solution, it is

[7] `eps` is the largest floating point number so that $1 + \imath\!\sim = 1$ in floating point arithmetic

worth noticing that this loss of symmetry does not affect the performances of the solver, as it has been proved in [EG04], Chapt. 8.

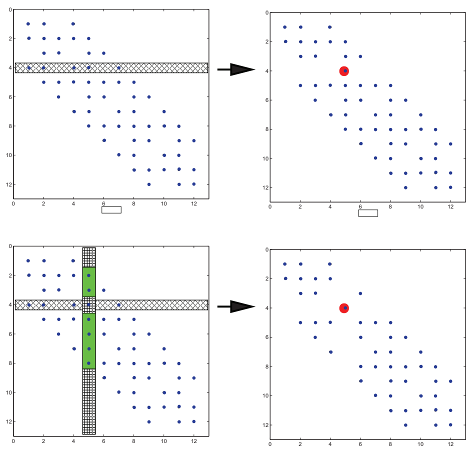

Fig. A.7. Treatment of essential conditions (e.g. on row 4 of the example matrix) using diagonalization: basic version (above) and symmetric one (below). For the symmetric version the column vector (bar the diagonal element) is used to update the source

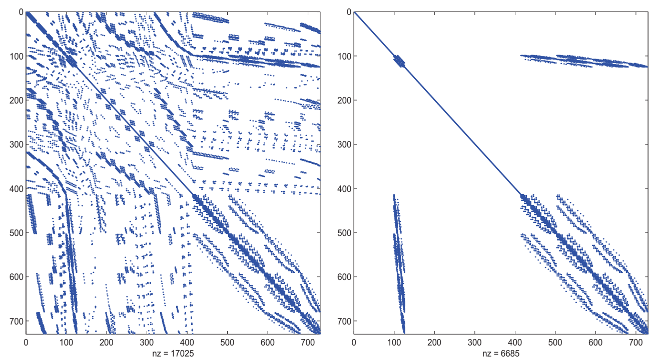

Fig. A.8. The effects of symmetric diagonalization on a 3D grid (from a real case): on the left, before the boundary conditions are imposed; on the right, afterwards

Symmetric diagonalization. Once we have "diagonalized" in the sense of the previous Section, we can think of modifying the column k_D in a way similar to what seen in Section A.2.1 (Fig. A.7 right): in practice we set to zero the elements $\widetilde{a}_{k_{nD}k_D}$ for all $k_{nD} \neq k_D$, and update the corresponding source term by adding to it $-A_{k_{nD}k_D}g_{k_D}$. The main difference with Section A.2.1 consists in annihilating off-diagonal coefficients in the columns corresponding to Dirichlet nodes, instead of erasing the whole columns of the system.

The technique just described (and adopted by `LifeV`) attains a most favorable balancing between easy programming and mathematical stability of the algebraic system. The con is that it needs an efficient access to columns. Therefore it can be convenient to use formats that warrant efficient access to columns as well, like the modified *MSR*.

A.2.4 Essential conditions in a vectorial problem

In presence of vectorial problems it can happen to impose essential conditions not on a single component in the vector of unknowns, but rather on a linear combination. Consider for example a Navier-Stokes problem, and suppose we want to impose a velocity condition like $\mathbf{u}^T\mathbf{n} = g$ on the boundary Γ_D, where $\mathbf{n}$ is the normal vector. The operation involves a linear combination of the components of $\mathbf{u}$: in 2D, $u_x n_x + u_y n_y = g$. If the normal is parallel to a coordinate axis we fall back into the case of a single component prescribing the condition. An example is $\mathbf{n} = [1, 0]^T$, forcing an essential condition on the first component: $u_x = g$. In general, though, $\mathbf{n}$ is arbitrary.

Let us then see how we can set up the problem by focusing on one boundary node. If the condition applies to more nodes (as normally happens), the procedure described below should be carried out at each node.

Suppose $\mathbf{U} \in \mathbb{R}^{N_h}$ contains all the problem's unknowns (hence, in particular, all the components of the vector), including those relative to the degrees of freedom at which we will impose the boundary condition. We shall only consider conditions that can be written in the form

$$\mathbf{N}^T\mathbf{U} = g, \tag{A.4}$$

where $\mathbf{N} \in \mathbb{R}^{N_h}$. Returning to the example, if we imposed $\mathbf{u}_i^T\mathbf{n} = g$ then $\mathbf{N}$ would have all components null except those corresponding, in $\mathbf{U}$, to the position of the velocity components $u_{i,x}$ and $u_{i,y}$ at node i, where they would equal n_x and n_y, respectively.

For simplicity (and without loss of generality) we shall assume $\mathbf{N}$ to be a unit vector, $\mathbf{N}^T\mathbf{N} = 1$. We will make use of

$$\mathrm{Z} = \mathbf{N}\mathbf{N}^T \in \mathbb{R}^{N_h \times N_h},$$

of components $Z_{ij} = N_i N_j$; this matrix enjoys the following properties[8]:

1. it is symmetric;
2. it has *rank one*, ie the set of $\mathbf{v} \in \mathbb{R}^{N_h}$ such that $\mathrm{Z}\mathbf{v} \neq \mathbf{0}$ is a vector space of dimension one. In fact, $\mathrm{Z}\mathbf{v}$ is the orthogonal projection of $\mathbf{v}$ along $\mathbf{N}$ because, by definition, $\mathrm{Z}\mathbf{v} = (\mathbf{N}^T\mathbf{v})\mathbf{N}$. Therefore $\mathrm{Z}\mathbf{v} = \mathbf{0}$ for all $\mathbf{v}$ orthogonal to $\mathbf{N}$, hence such that $\mathbf{N}^T\mathbf{v} = 0$.

Now take the matrix $\widetilde{\mathrm{A}}$ and the right-hand side $\widetilde{\mathbf{b}}$ of our problem before the boundary condition (A.4) is imposed. To do so we can use Lagrange multipliers. The method consists in adding a further unknown λ and solve

$$\begin{cases} \widetilde{\mathrm{A}}\mathbf{U} + \lambda\mathbf{N} = \widetilde{\mathbf{b}}, \\ \mathbf{N}^T\mathbf{U} = g. \end{cases} \tag{A.5}$$

This problem has an extra unknown, but a series of algebraic manipulations will eliminate λ and reduce it to a system in $\mathbf{U}$ only, of the form $\mathrm{A}\mathbf{U} = \mathbf{b}$ where

$$\mathrm{A} = \widetilde{\mathrm{A}} - \mathrm{Z}\widetilde{\mathrm{A}} + \mathrm{Z}\widetilde{\mathrm{A}}\mathrm{Z}, \quad \mathbf{b} = \widetilde{\mathbf{b}} - \mathrm{Z}\widetilde{\mathbf{b}} + g\mathrm{Z}\widetilde{\mathrm{A}}\mathbf{N}. \tag{A.6}$$

In addition, one can prove the system can be further simplified to become

$$\left[\widetilde{\mathrm{A}} - \mathrm{Z}\widetilde{\mathrm{A}} + \alpha\mathrm{Z}\right] \mathbf{U} = \widetilde{\mathbf{b}} - \mathrm{Z}\widetilde{\mathbf{b}} + g\alpha\mathbf{N}, \tag{A.7}$$

where the parameter α can be chosen so to not ill-condition the system (although, in practice, one chooses $\alpha = 1$ often).

Looking at how matrix and source in (A.7) are built, the operation $\widetilde{\mathrm{A}} - \mathrm{Z}\widetilde{\mathrm{A}}$ is nothing more that a generalized version of the annihilation seen in Section A.2.3. The addition of $\alpha\mathrm{Z}$ generalizes the introduction of the diagonal term α, which in the case of a (non-trivial) linear combination of unknowns entails a change in the original pattern.

Likewise, the operations on the source term correspond to replacing the original component of the right-hand side along $\mathbf{N}$ with $g\alpha\mathbf{N}$. Actually, one can check easily that if the components of $\mathbf{N}$ are zero except for $N_i = 1$, the procedure corresponds *exactly* to set to zero the whole ith row except the diagonal term, and modify the source as we saw in Section A.2.3.

This routine has two shortcomings. The first is unavoidable and is due to the distinct patterns of A and $\widetilde{\mathrm{A}}$. Usually, the boundary condition (A.4) constrains some components of the solution that are instead uncoupled in the "unconstrained" system. We have seen that the majority of formats for sparse grids are not efficient when the pattern is altered (save for COO, which is however less efficient than other formats). One possibility to steer clear of the issue is to take this kind of boundary

[8] In concrete cases neither the matrix Z, nor $\mathbf{N}$, need be constructed explicitly, but they are useful algebraic tools that allow to describe the procedure in a concise and accurate way.

conditions into account already in the preliminary phase, when the pattern is identified.

The second fault is related to the fact that A, as of (A.6), just like the matrix of (A.7), is not symmetric in general, even if $\widetilde{\mathrm{A}}$ is. The operations considered so far, in fact, correspond to acting only on rows. So, if we want to keep the matrix symmetric it could be necessary to generalize the "symmetric diagonalization", a topic we will not discuss for lack of space.

One final, practical remark is that if one adopts iterative methods to solve the linear system it is unnecessary to construct A explicitly: one can use, instead, the definition in terms of $\widetilde{\mathrm{A}}$ and $\mathbf{N}$ directly. What one needs is to implement efficiently the products $\mathbf{N}^T\mathbf{v}$ and $\mathrm{Z}\mathbf{v}$, for any given $\mathbf{v}$.