

An Abstraction-Preserving Block Matrix Implementation in Maple

Stephen M. Watt

Ontario Research Centre for Computer Algebra
and Cheriton School of Computer Science
University of Waterloo, Canada
smwatt@uwaterloo.ca

D. J. Jeffrey

Ontario Research Centre for Computer Algebra
and Department of Mathematics
University of Western Ontario, Canada
djeffrey@uwo.ca

Abstract—A Maple implementation of partitioned matrices is described. A recursive block data structure is used, with all operations preserving the block abstraction. These include constructor functions, ring operations such as addition and product, and inversion. The package is demonstrated by calculating the *PLU* factorization of a block matrix.

I. INTRODUCTION

Partitioning a matrix into blocks is an elementary concept in linear algebra [2], [5]. It is supported in all major mathematical software systems, but typically only as a means for building a matrix or specifying submatrices – a matrix is flattened before operations can be performed with it. In more advanced studies, however, the abstraction of a block matrix is often used to present and analyse algorithms [8]. Therefore, an ability to work with block matrices as computational objects is desirable. In addition to theoretical studies, block matrices offer potential benefits for computation:

- dense, sparse and structured matrices can be represented with reasonable space efficiency [1], [4],
- block matrices provide a middle ground that avoids pathological communication bottlenecks in row-major or column-major code [3], and
- multiplication and related algorithms can have improved computational time complexity [7].

The basic algebra of block matrices is covered in standard references [6], and a theoretical basis for implementing operations has been outlined in [11].

An implementation is described here that preserves and respects the block structure for all the usual matrix operations, including inversion. The functions are demonstrated by using them to compute a matrix *PLU* factorization.

The implementation is in Maple, allowing the ultimate matrix elements to be symbolic expressions. To work efficiently with block matrices with floating point or finite field entries, a compiled language such as C++, Rust, Julia or Aldor [9] may be used. All of these offer parametric types and operator overloading, allowing elegant implementations.

The remainder of the paper is organized as follows: Section II describes the Maple data structure we use for recursive

block matrices and some of the considerations in the choice. Section III presents a Maple module used to encapsulate the data representation and describes the operations provided. Section IV shows how the block operations may be used to compute *PLU* decomposition of square non-singular matrices. Section V gives examples of using the package and Section VI outlines some directions for future work. Finally, VII gives some brief conclusions.

II. DATA STRUCTURE

We describe here the data structure we have used for block matrices. We have used an expression tree structure with the interior nodes being symbolic function applications, $\text{_BM}(kind, er, ec, val)$. This is represented as a pair with the first element being a pointer to the name _BM and second element being a pointer to a four element “expression sequence” which is basically four consecutive words in memory plus a header.

The first element, *kind*, is a name being one of *zero*, *scalar*, *matrix* or *rblock*. The meaning of these is described below.

The second and third elements, *er* and *ec*, are integers giving the number of leaf rows and columns, respectively. For example, the block matrix

$$\begin{bmatrix} \begin{bmatrix} 11 & 12 \\ 13 & 14 \end{bmatrix} & \begin{bmatrix} 15 & 16 & 17 \\ 18 & 19 & 20 \end{bmatrix} \\ \begin{bmatrix} 21 & 22 \end{bmatrix} & \begin{bmatrix} 23 & 24 & 25 \end{bmatrix} \end{bmatrix}$$

would have $er = 3$ and $ec = 5$.

The meaning of the fourth element, *val*, depends on the value of *kind*:

- When $kind = zero$, the block is interpreted as a zero matrix whose entries are not explicitly stored. The value of *val* is always the integer 0.
- When $kind = scalar$, the block is interpreted as a diagonal matrix equal to *val* times the $er \times er$ identity. To be well-formed, it is required that $er = ec$.
- When $kind = matrix$, *val* is a $er \times er$ Maple object of type *Matrix* whose entries are leaf elements of the block matrix.

- When $kind = rblock$, val is a Maple object of type `Matrix` whose entries are themselves block matrices. (The name connotes “recursive block”).

To be well formed, two conditions are required:

- Each of the blocks in a given row must have the same er , and the sum of their ec values must equal the ec value of the whole block matrix.
- Each of the blocks in a given column must have the same ec , and the sum of their er values must equal the er value of the whole block matrix.

There are several optimizations that could be done to make this representation more space efficient, but the above representation is convenient for development purposes. For example, it would be possible to use the symbolic $kind$ names in place of `_BM` and have different length expression sequences for each kind, or it would be possible to have a $kind$ for general diagonal matrices, and so on. It would also be possible to omit the er and ec fields in some of the kinds, since these can sometimes be determined by inspection, but having them always present gives more uniform code and avoids computation.

III. MODULE

All of the operations on block matrices are collected in a Maple module that hides the representation and presents an abstract interface. The following operations are provided.

A. Construction operations:

- `BM(vals)` constructs a block matrix from a list of lists or an object of type `Matrix`. The $kind$ is `rblock` if all the entries of $vals$ are block matrices. Otherwise the $kind$ is `matrix`.
Finer control is provided by the constructors below.
- `zeroBM(er, ec)` gives $kind$ zero of dimension $er \times ec$.
- `scalarBM(er, s)` gives $kind$ scalar of dimension $er \times er$, interpreted as a matrix with diagonal entries equal to s and off-diagonal entries equal to 0.
- `matrixBM(m)` gives $kind$ matrix with entries given by the matrix m .
- `rblockBM(m)` gives $kind$ `rblock` with entries themselves being block matrices given by the elements of matrix m .
- A number of other specialized constructors.

B. Operations to abstract the type:

- ``type/BM`(B)` tests whether B is a block matrix, returning `true` or `false`. This allows Maple statements of the form `if type(A, 'BM') then ..` so block matrices participate in Maple’s structured type system.
- `nEltRows(B)` and `nEltCols(B)` give the number of leaf element rows and columns.
- `elt(B, r, c)` return the (r, c) leaf element, *i.e.* when B is viewed as a un-partitioned matrix.

- `nBlockRows(B)` and `nBlockCols(B)` gives the number of rows of blocks and columns of blocks.
- `block(B, r, c)` return the (r, c) block, *i.e.* when B is viewed as a block matrix.

Operations to traverse the data structure:

- `map(f, A)` returns a block matrix whose elements are those of A with the function f applied.
Logically, if $R = \text{map}(f, A)$, then $R_{ij} = f(A_{ij})$, for $1 \leq i \leq er, 1 \leq j \leq ec$. The resulting matrix always has the same block structure as A , but the kinds of the blocks can change. For example if $f(0) \neq 0$ then a zero block will have a `matrix` block as its image.
- `zip(f, A, B)` returns a block matrix whose elements are the values of f applied to pairs of corresponding elements from A and B . Logically, if $R = \text{zip}(f, A, B)$, then $R_{ij} = f(A_{ij}, B_{ij})$, for $1 \leq i \leq er, 1 \leq j \leq ec$. A and B must have the same dimension and block structure and this will be the block structure of the result. As with `map`, the kinds of the blocks can change. The implementation handles combinations of different kinds of blocks. The following combinations of kinds are allowed:
 - zero with any kind and any kind with zero
 - scalar with any kind and any kind with scalar
 - matrix with matrix and
 - `rblock` with `rblock`.

- `blockKind(B)` and `blockVal(B)` return the $kind$ and val fields B respectively. These are lower-level operations, not intended for external use, but which are required when traversing or combining block matrix data structures such as with `map` or `zip`.

C. Matrix ring operations:

- `plus(A, B)` and ``minus`(A, B)` compute $A + B$ and $A - B$ respectively. The same shape rules apply as for `zip`. The grave characters “`” are required because `minus` is a keyword in Maple meaning set difference.
- `neg(A)` computes $-A$. The shape rules are as for `map`.
- `times(A, B)` computes $A \cdot B$. The inputs must be structured such that the required products of blocks is defined. The same combinations of blocks are allowed as for `zip`. At present, only classical matrix multiplication is used, though it would be straightforward to use Strassen recursive multiplication beyond a given size.
- `hermTrans(A)` computes the Hermitian transpose of A , that is, logically, $A_{ij} = \overline{A_{ji}}$, where \bar{z} denotes complex conjugation. The blocks of the result are of the same kinds as transposed blocks of A .

D. Inversion-related operations:

- `inv(M)` computes the multiplicative inverse of M , that is A^{-1} . If the block matrix is singular, then `FAIL` is returned. At present only 1×1 and 2×2 blocks are

handled, though it would be straightforward to handle more rows and columns of blocks by grouping them.

First, `tryInv` is called, attempting to compute the inverse assuming that the $(1,1)$ block is non-singular. If that fails, the more general method [10] provided by `invByMTM` is used.

In principle, if the $(1,1)$ block is singular, it would be possible to permute the blocks and try again to see if any other block is non-singular. But whether the multiple tries would give a better expected execution time less than the general method may depend on the element type. An alternative would be to precondition by multiplying with a random invertible matrix.

- `tryInv(M)` attempts to invert M using

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}BS_A^{-1}CA^{-1} & -A^{-1}BS_A^{-1} \\ -S_A^{-1}CA^{-1} & S_A^{-1} \end{bmatrix}$$

where $S_A = D - CA^{-1}B$ is the Schur complement of A . This may fail by having A singular, even if the whole matrix is invertible. However, if the whole matrix and A are both invertible, then S_A will be as well.

- `invByMTM(M)` computes the inverse of M using

$$M^{-1} = (M^\dagger M)^{-1} M^\dagger,$$

where M^\dagger is the Hermitian transpose of M . For formally real or complex element rings, $M^\dagger M$ will have invertible principal minors. In particular its $(1,1)$ will be invertible by `tryInv`.

- `schurComp(M, i, j, inv, ...)` computes the Schur complement of the (i,j) block in M . The inversion required to compute the Schur complement is performed using the `inv` functional parameter. Optional extra arguments allow the return of some of the products used in computing the Schur complement.
- `PLU-Decomp(M)` computes the PLU decomposition of M , if possible, or returns *FAIL*. The result is a triple (P,L,U) of block matrices, where P represents a permutation matrix, L represents a lower triangular with diagonal elements equal to 1, and U represents an upper triangular matrix. The block matrices L and U are triangular as matrices of elements, not only as matrices of blocks. For example, the $(1,1)$ block of L is also lower triangular, and recursively.

IV. PLU DECOMPOSITION

One of the goals of this work was to implement the ideas of [11], and, in particular, the description of PLU decomposition of non-singular block matrices using only block operations. As described in Section III-D, the PLU decomposition of a matrix M is a triple, (P,L,U) such that $M = PLU$, P is a permutation matrix, L is lower triangular and U is upper triangular. We require L and U to be triangular element-wise, not just block-wise, and we take the disambiguating convention that the diagonal of L contains 1s.

```

1 a := block(M,1,1); b := block(M,1,2);
2 c := block(M,2,1); d := block(M,2,2);
3
4 ra, ca := nEltRows(a), nEltCols(a);
5 rb, cb := nEltRows(b), nEltCols(b);
6 rc, cc := nEltRows(c), nEltCols(c);
7 rd, cd := nEltRows(d), nEltCols(d);
8
9 # Check shape.
10 orElse(ra = ca and rd = cd,
11        "Principal block not square");
12 orElse(ra = rb and rc = rd and
13        ca = cc and cb = cd, "Bad shape");
14 n1 := ra;
15 n2 := rd;
16
17 # Compute decomposition recursively.
18 p1, l1, u1 := PLU-Decomp(a);           # n1 x n1 all
19 l1inv      := inv(l1);                 # n1 x n1
20 ulinv     := inv(u1);                 # n1 x n1
21 x         := times(c, ulinv);         # n2 x n1
22 y         := times(l1inv, b);        # n1 x n2
23 y         := times(inv(p1), y);      # n1 x n2
24 t         := `minus`(d, times(x,y)); # n2 x n2
25 p2, l2, u2 := PLU-Decomp(t);         # n2 x n2 all
26
27 # Zero blocks of the needed sizes.
28 z12 := zeroBM(n1,n2); z21 := zeroBM(n2,n1);
29
30 # Return P, L, U
31 BM([p1, z12], [z21, p2]), # P
32 BM([l1, z12], [x, l2]), # L
33 BM([u1, y], [z21, u2]) # U

```

Fig. 1. 2×2 rblock case of `PLU-Decomp`

For the present, we take the simplifying assumption that M is a square block matrix and the leading principal minors of M are non-singular. Then, as shown in [11], if

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix},$$

and we seek $M = LU$ such that

$$L = \begin{bmatrix} L_1 & 0 \\ X & L_2 \end{bmatrix} \quad U = \begin{bmatrix} U_1 & Y \\ 0 & U_2 \end{bmatrix}$$

with L_1 and L_2 lower triangular and U_1 and U_2 upper triangular, we may compute

$$\begin{aligned} L_1 U_1 &= A && \text{recursively} \\ X &= C U_1^{-1} \\ Y &= L_1^{-1} B \\ L_2 U_2 &= D - XY && \text{recursively.} \end{aligned}$$

The implementation of the core 2×2 rblock case is shown in Figure 1. Notice that we are able to use the functions described so far to provide relatively easy to read, natural code.

V. EXAMPLES

We now give examples of using the `BlockMatrix` module. In order to have results that can be shown in the article, we use small examples with simple elements. The first example, shown in Figure 2, inverts a block matrix all of whose blocks are singular. The second example, shown in Figure 3, performs a PLU factorization.

```

> pBM := BlockMatrix :
BM := pBM-BM :
> M55 := BM(
  [[ BM([[x, 0], [1, 0]]), BM([[0, 0, 0], [y, 0, 0]]),
    [ BM([[0, z], [0, 0], [0, 0]]), BM([[0, 0, 0], [0, 0, a], [0, b, c]]) ]])

```

$$M55 := R \begin{pmatrix} M \begin{pmatrix} x & 0 \\ 1 & 0 \end{pmatrix} & M \begin{pmatrix} 0 & 0 & 0 \\ y & 0 & 0 \end{pmatrix} \\ M \begin{pmatrix} 0 & z \\ 0 & 0 \\ 0 & 0 \end{pmatrix} & M \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & a \\ 0 & b & c \end{pmatrix} \end{pmatrix} \quad (1)$$

```

> pBM-map(normal, pBM-inv(M55))

```

$$R \begin{pmatrix} M \begin{pmatrix} 1 & 0 \\ x & 0 \\ 0 & 0 \end{pmatrix} & M \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ z & 0 & 0 \end{pmatrix} \\ M \begin{pmatrix} -1 & 1 \\ yx & y \\ 0 & 0 \\ 0 & 0 \end{pmatrix} & M \begin{pmatrix} 0 & 0 & 0 \\ 0 & -c & 1 \\ 0 & 1 & 0 \\ 0 & a & 0 \end{pmatrix} \end{pmatrix} \quad (2)$$

```

> pBM-times(M55, %)

```

$$R \begin{pmatrix} S_2(1) & Z_{2,3} \\ Z_{3,2} & S_3(1) \end{pmatrix} \quad (3)$$

Fig. 2. Inversion of a block matrix whose blocks are all singular. The symbols R, M, Z, S respectively indicate recursive, matrix, zero and scalar nodes. The last line verifies the matrix and inverse multiply to give an identity matrix.

VI. FUTURE WORK

There are a number of additional operations and tidying up that would be required for a generally useful block matrix package, including

- re-organizing block matrices to desired compatible shapes
- inversion of other than 1×1 and 2×2 block matrices
- PLU decomposition of non-square matrices and matrices with singular principal minors
- matrix norms
- eigenvalue computation
- and many others.

It would also be useful to examine the trade-offs in computing inverses — when to try to pivot blocks to give a non-singular $(1, 1)$ block *versus* going to the more general $(M^\dagger M)^{-1} M^\dagger$ method *versus* preconditioning by a matrix other than M^\dagger .

Finally, it would be useful to experiment with a compiled programming language with parametric polymorphism and overloading to see how competitive these methods can be on larger numerical examples.

VII. CONCLUSIONS

We have described a Maple implementation of recursive block matrices and operations on them and we have shown that many of the operations of linear algebra can be performed using

```

> M55 := BM(
  [[ BM([[0, 2], [1, 2]]), BM([[0, 1, 2], [1, 0, 1]]),
    [ BM([[3, 0], [2, 1], [1, 2]]), BM([[4, 0, 4], [4, 0, 1], [1, 4, 0]]) ]])

```

$$M55 := R \begin{pmatrix} M \begin{pmatrix} 0 & 2 \\ 1 & 2 \end{pmatrix} & M \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \end{pmatrix} \\ M \begin{pmatrix} 3 & 0 \\ 2 & 1 \\ 1 & 2 \end{pmatrix} & M \begin{pmatrix} 4 & 0 & 4 \\ 4 & 0 & 1 \\ 1 & 4 & 0 \end{pmatrix} \end{pmatrix} \quad (4)$$

```

> P55, L55, U55 := pBM-PLU-Decomp(M55) :
> P55

```

$$R \begin{pmatrix} M \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} & Z_{2,3} \\ Z_{3,2} & S_3(1) \end{pmatrix} \quad (5)$$

```

> L55

```

$$R \begin{pmatrix} S_2(1) & Z_{2,3} \\ M \begin{pmatrix} 3 & -3 \\ 2 & -\frac{3}{2} \\ 1 & 0 \end{pmatrix} & M \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & -\frac{8}{9} & 1 \end{pmatrix} \end{pmatrix} \quad (6)$$

```

> U55

```

$$R \begin{pmatrix} M \begin{pmatrix} 1 & 2 \\ 0 & 2 \end{pmatrix} & M \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \end{pmatrix} \\ Z_{3,2} & M \begin{pmatrix} 1 & 3 & 7 \\ 0 & -\frac{9}{2} & -12 \\ 0 & 0 & -\frac{35}{3} \end{pmatrix} \end{pmatrix} \quad (7)$$

```

> pBM-'minus'(M55, pBM-times(P55, pBM-times(L55, U55)))

```

$$R \begin{pmatrix} Z_{2,2} & Z_{2,3} \\ Z_{3,2} & Z_{3,3} \end{pmatrix} \quad (8)$$

Fig. 3. PLU decomposition of a block matrix and check of result

block operations only, without breaking the block abstraction. The whole package is about 535 lines of Maple code.

REFERENCES

- [1] S.K. Abdali and D.S. Wise. Experiments with quadtree representation of matrices. In P. Gianni, editor, *Proc. ISSAC 88*, page 96–108. Springer Verlag LNCS 358, 1989.
- [2] Howard Anton. *Elementary Linear Algebra, 11th ed.* Wiley, 2014.
- [3] Jack J. Dongarra, Robert A. van de Geun, and David W. Walker. Scalability issues affecting the design of a dense linear algebra library. *J. Parallel and Distributed Computing*, 22:52–537, 1994.
- [4] Irene Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.
- [5] Peter J. Olver and Chehrzad Skakiban. *Applied Linear Algebra*. Springer, 2018.
- [6] Robert Reams. *Handbook of Linear Algebra*, chapter Partitioned Matrices, pages 11.1–11.10. CRC Press, 2nd edition, 2014.
- [7] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.
- [8] Lloyd N. Trefethen and David B. Bau III. *Numerical Linear Algebra*. SIAM, 1997.
- [9] S.M. Watt. Aldor. In *Handbook of Computer Algebra*, page 265–270. Springer Verlag, 2003.
- [10] Stephen M. Watt. Pivot-free block matrix inversion. In *Proc. 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2006)*, pages 151–155, 2006.
- [11] Stephen M. Watt. Algorithms for recursive block matrices. In *Proc. LALO60: Matrices and Polynomials in Computer Algebra*. also arXiv:2407.03976v1, 2024.