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*—An implementation supporting recursive block, or partitioned, matrices in Maple is described. A data structure is proposed and support functions are defined. These include constructor functions, basic operations such as addition and product, and inversion. The package is demonstrated by calculating the LU factors of a block-defined matrix.**

I. Introduction

Partitioning a matrix into blocks is an elementary concept in linear algebra [2], [5], [6]. 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. However, a data representation of matrices with elements themselves being matrices, and potentially recursively so, has several useful properties:

- 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].

It should be remembered that when a matrix is viewed as having blocks as elements, the elements are clearly non-commutative. It is easy to see how ring operations may be performed. For example, given

$$M_1 = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \qquad \text{and} \qquad M_2 = \begin{bmatrix} E & F \\ G & H \end{bmatrix},$$

one may compute $M_1 + M_2$ if corresponding blocks have the same shape so $A+E$, $B+F$, $C+G$, and $D+H$ are defined. Similarly, one may compute $M_1 M_2$ if the block dimensions allow the multiplications AE, BG, AF, AH, CE, DG, CF, and DH. These requirements apply recursively if A, B, *etc* are themselves block matrices. Provided M_1 and M_2 are non-singular, it is also possible to compute their inverses using only block operations, even if all of the sub-blocks are singular. A theoretical basis for the operations has been outlined in [10].

An implementation is described here which preserves and respects the block structure of matrices while allowing the usual matrix operations to be performed. The functions implemented are verified by using them to compute an LU factorization of a matrix.

The implementation described here 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 [8] may be used. All of these offer parametric types and operator overloading, allowing an elegant implementation.

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 gives some details about the implementation, pointing to some of the considerations in structuring the code. Section V shows how the block operations may be used to compute PLU decomposition of square non-singular matrices. Section VI gives examples of using the package and Section VII outlines some directions for future work. Finally, VIII gives some brief conclusions.

II. Data Structure

We describe here the data structure we have used for block matrices. As Maple does not provide lightweight structures with named fields, we have used a symbolic function application, `_BM(`*`kind`*`, `*`er`*`, `*`ec`*`, `*`val`*`)`. This is represented internally as a pair with 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 the `er` values of all the rows must equal the `er` value of the whole block matrix.
  - Each of the blocks in a given column must have the same `ec`, and the sum of the `ec` values of all the columns must equal the `ec` 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 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. It would also 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`.
- `genDiagBM(er, ec, v)` gives a block matrix of dimension $er \times er$. If `v` is zero, the `kind` is `zero`. If *er* = *ec*, the `kind` is `scalar`. Otherwise the `kind` is `matrix`. This operation is useful when constructing sub-blocks in arithmetic operations.
- `shapedDiagBM(M, v)` is the same as `genDiagBM(er, ec, v)` with `er` and `ec` taken from the dimensions of matrix *M*.

B. Operations to abstract the type:

- `` `type/BM`(B) `` test whether `B` is a block matrix. Returns `true` or `false`. Allows Maple statements of the form `if type(A, 'BM') then ..` and for block matrices to participate in Maple's structured type system. The grave characters "`" allow the solidus "/" to appear in the name.
- `nEltRows(B)` and `nEltCols(B)` give the number of rows and columns, respectively, when `B` is viewed as a usual matrix.
- `elt(B,r,c)` return the (r, c) element, when `B` is viewed as a usual matrix.
- `nBlockRows(B)` and `nBlockCols(B)` give the number of rows and columns of blocks. If `kind` is not `rblock` then these operations return 1.
- `block(B,r,c)` return the (r, c) block, 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 $R = \texttt{map}(f, A)$, then $R_{ij} = f(A_{ij})$, for $1 \le i \le er, 1 \le j \le ec$.
  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 $R = \texttt{zip}(f, A, B)$, then $R_{ij} = f(A_{ij}, B_{ij})$, for $1 \le i \le er, 1 \le j \le ec$.
  `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)` computes $A + B$. The same shape rules apply as for `zip`.
- `` `minus`(A, B) `` computes $A - B$. 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)` -A. The same shape rules apply 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`, provided the dimensions allow matrix multiplication.
  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 $\overline{z}$ 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 [9] 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 smaller 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)` attempt 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 M is invertible. However, if M and A are both invertible, then so will be S_A.
- `invByMTM(M)` computes the inverse of M using

$$M^{-1} = (M^*M)^{-1}M^*,$$

  where M^* is the Hermitian transpose of M. For formally real or complex element rings, M^*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* function parameter. Optional extra arguments allows the return of some of the products used in computing the Schur complement. This is discussed in more detail below.
- `PLU_Decomp(M)` compute the PLU decomposition of M, if possible, or return $FAIL$. The result is a triple (P, L, U) of block matrices, where P is represents a permutation matrix, L is represents a lower triangular with diagonal elements equal to 1, and U is 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.

E. Local Utilities:

- `isWellFormed(A)` returns `true` or `false` according as whether the rules about row and column dimensions are satisfied.
- `isSameShape(A, B)` tests whether A and B have the same shape, meaning the same dimension and kinds of blocks, and recursively.
- `mustBeBM(A)` gives an error if A is not a block matrix.
- `orElse(bool, args)` has no action if *bool* is true. if *bool* is `false` it gives an error with *args* as arguments.
- `never(args)` is for locations in the code that should never be reached. It gives an error with *args* as arguments.

IV. Some Implementation Details

We now discuss some of the details of the implementation to illustrate the package.

A. Traversal Functions

We begin by showing how the traversal functions deal with blocks of the varying kinds. The `map` function is shown in Figure 1. The way in which the function f is applied depends on the kind of block matrix. On line 5 we see how `genDiagBM` is used to sort out the appropriate kind for a block containing elements equal to $f(0)$. For kind `matrix`,

```
export map := proc(f, A)
    local kind, er, ec, v;
    er, ec  := nEltRows(A),  nEltCols(A);
    kind, v := blockKind(A), blockVal(A);
    if   kind = 'zero'   then genDiagBM(er, ec, f(0))
    elif kind = 'scalar' then scalarBM(er,  f(v))
    elif kind = 'matrix' then matrixBM(:-map(f, v))
    elif kind = 'rblock' then rblockBM(:-map(b->map(f,b),
                                            v))
    else never("Malformed BlockMatrix", A)
    fi
end:

export neg := A -> map(a->-a, A):
```

Fig. 1. Implementation of `map` and `neg`

```
# Zip requires compatible sub-blocks.
export zip := proc(f, A, B)
    local ka, kb, va, vb, er, ec, br, bc, r, c, sfun;
    mustBeBM(A);  mustBeBM(B);
    orElse(isSameShape(A, B),
           "Incompatible matrices.", A, B);

    ka, va := blockKind(A), blockVal(A);
    kb, vb := blockKind(B), blockVal(B);
    er, ec := nEltRows(A),  nEltCols(A);   # Same as B
    br, bc := nBlockRows(A),nBlockCols(A); # Same as B

    if   (ka = 'zero' or ka = 'scalar') and
         (kb = 'zero' or kb = 'scalar') then
        genDiagBM(er, ec, f(va,vb))
    elif ka = 'matrix' and kb = 'matrix' then
        matrixBM(:-zip(f, va, vb))
    elif ka = 'rblock' and kb = 'rblock' then
        rblockBM(:-zip((a,b)-> zip(f,a,b), va, vb))
    elif ka = 'zero' or ka = 'scalar' then
        if kb = 'matrix' then
            sfun := proc(b)
                if r = c then f(va, b) else b fi
            end;
            matrixBM([seq([seq(sfun(vb[r,c]),
                      c=1..ec)], r=1..er)])
        elif kb = 'rblock' then
            sfun := proc(b) if r <> c then b
                else zip(f, shapedDiagBM(b, va), b) fi
            end;
            rblockBM([seq([seq(sfun(vb[r,c]),
                           c=1..bc)], r=1..br)])
        else never("Malformed BlockMatrix", B)
        fi
    elif kb = 'zero' or kb = 'scalar' then
        if ka = 'matrix' then
            sfun := proc(a)
                if r = c then f(a, vb) else a fi
            end;
            matrixBM([seq([seq(sfun(va[r,c]),
                      c=1..ec)], r=1..er)])
        elif ka = 'rblock' then
            sfun := proc(a) if r <> c then a
                else zip(f, a, shapedDiagBM(a, vb)) fi
            end;
            rblockBM([seq([seq(sfun(va[r,c]),
                      c=1..bc)], r=1..br)])
        else never("Malformed BlockMatrix", A)
        fi
    else
        never("Malformed BlockMatrix pair", A, B)
    fi
end:

export plus    := (A, B) -> zip((a,b)->a+b, A, B):
export `minus` := (A, B) -> zip((a,b)->a-b, A, B);
```

Fig. 2. Implementation of `zip`, `plus` and `` `minus` ``

```
export times := proc(A, B)
       local ka, kb, va, vb,
             era, eca, erb, ecb, bra, bca, brb, bcb,
             r, c, rows, row, i, elt;
       mustBeBM(A);  mustBeBM(B);

       ka, va  := blockKind(A),  blockVal(A);
       era,eca := nEltRows(A),   nEltCols(A);
       bra,bca := nBlockRows(A), nBlockCols(A);

       kb, vb  := blockKind(B),  blockVal(B);
       erb,ecb := nEltRows(B),   nEltCols(B);
       brb,bcb := nBlockRows(B), nBlockCols(B);

       orElse(eca = erb and bca = brb,
              "Incompatible Shapes", eca,erb,bca,brb);

       if   ka = 'zero' or kb = 'zero' then
           zeroBM(era,ecb)
       elif ka = 'scalar' then map(b->va * b, B)
       elif kb = 'scalar' then map(a->a * vb, A)
       elif ka = 'matrix' then
           orElse(kb = 'matrix',
               "Incompatible BlockMatrix values", A,B);
           matrixBM(va . vb)
       elif ka = 'rblock' then
           orElse(kb = 'rblock',
               "Incompatible BlockMatrix values", A,B);
           rows := NULL;
           for r from 1 to bra do
               row := NULL;
               for c from 1 to bcb do
                   elt := times(block(A,r,1),
                                block(B,1,c));
                   for i from 2 to brb do
                       elt := plus(elt,
                           times(block(A,r,i),
                           block(B,i,c)))
                   od;
                   row := row, elt;
               od;
               rows := rows, [row];
           od;
           rblockBM([rows])
       else
           never("Malformed BlockMatrix", A)
       fi
   end:
```

Fig. 3. Implementation of `times`

```
# Arguments named opt_... optionally pass back values.

export schurComp :=
proc(A, i, j, inv, opt_ainv, opt_cainv, opt_ainvb)
    local a,b,c,d, ainv, cainv;

    orElse(nBlockRows(A)=2 and nBlockCols(A)=2,
        "Unhandled Schur comploement");

    # If i is 1 or 2 then 3-i is 2 or 1, respectively.
    a := block(A,i,j);   b := block(A,i,3-j);
    c := block(A,3-i,j); d := block(A,3-i,3-j);

    ainv  := inv(a);
    cainv := times(c, ainv);

    if nargs >=5 then opt_ainv  := ainv  fi;
    if nargs >=6 then opt_cainv := cainv fi;
    if nargs >=7 then opt_ainvb := times(ainv, b) fi;

    `minus`(d, times(cainv, b))
end:
```

Fig. 4. Implementation of Schur complement in 2×2 matrix

the builtin Maple map is used to apply f to the matrix elements. It is invoked using `:-map` on line 7 as otherwise the `map` being defined would be called recursively. For kind `rblock`, the builtin `map` is used to apply the anonymous function `b->map(f,b)` to the sub-blocks. The call to `map` in the anonymous function is a recursive call to the `map` being defined. In all cases, the last expression evaluated is returned from the function, as is usual in Maple.

With `map` defined this way, it is possible to define `neg` as shown on line 14.

The `zip` function is considerably more involved and illustrates how different kinds of blocks may be combined, as shown in Figure 2. After some initial checks, the function begins by handling the cases where both `A` and `B` are either of kind `zero` or `scalar` on line 15. That is, neither are stored as explicit matrices and the resulting block is computed with only a single call to `f`. Next, the cases where both blocks are of the same kind are handled on lines 17 and 19. The more complicated cases come when explicit and implicit blocks are combined, that is when a `zero` or `scalar` block is combined with a `matrix` or `rblock`. These are handled in the sections starting on lines 20 and 35. With these details sorted out, it is simple to define `plus` and `` `minus` `` as shown lines 55 and 56.

B. Ring Operations

We have already seen how `neg`, `plus` and `` `minus` `` can be implemented with functional traversal operations. It would be possible to write a general convolution function and use that for multiplication, but since there is only one use at present, we specialize it for `times` as shown in Figure 3. The cases of multiplying by a `zero` block are simple to handle. Multiplying by a `scalar` block is likewise easy. When both blocks are `matrix` blocks, the builtin matrix multiplication is used. When both are `rblock` blocks, classical multiplication with sub-blocks as elements is performed, in which case the multiplication of sub-blocks is done with a recursive call to `times`.

The implementation of Hermitian transpose is obvious and not shown. The placement of the sub-blocks is transposed and they are individually Hermitian-transposed. For `zero` blocks, the row and column dimensions are swapped. For `scalar` blocks, the value is conjugated, and for `matrix` blocks, the builtin Maple `HermitianTranspose` from the `LinearAlgebra` package is used.

C. Inversion-Related Operations

We implement inversion only for matrices with a single block or with 2×2 blocks. If matrices have more blocks, then it would be possible to group them.

To compute inverses of block matrices, the Schur complement is required. The implementation is shown in Figure 4. The function `schurComp` implements the formula $S_A = D - CA^{-1}B$ where A can be any of the $(1,1)$, $(1,2)$, $(2,1)$ or $(2,2)$ blocks and D is the block diagonally opposite. This is achieved on lines 11 and 12 using the indexing trick that if i is 1 or 2, then $3-i$ is 2 or 1, respectively. The fourth argument to the function is the inversion routine to use, as different ones may be used in different situations. In computing the Schur complement, certain matrix products are or may be formed. These may be useful to the caller, so the function `schurComp` allows optional extra arguments to pass back values. This occurs on lines 17 to 19 where arguments allow A^{-1}, CA^{-1} and $A^{-1}B$ to be returned.

With the Schur complement in place, it is possible to provide block matrix inverse. The first step is to try to apply the formula given for `tryInv` in Section III-D. The function `tryInv` sets up a catch point so that attempting to invert any singular sub-block can immediately exit to the top level. We note that it is possible that all sub-blocks be singular even though the block matrix as a whole is non-singular, *e.g.*

$$\begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \end{bmatrix}.$$

The recursive function `tryInvInner` is shown in Figure 5. On line 8 an error is raised, throwing the value of the variable `singularError`. This is caught by the top-level `tryInv`. The functions `tryInvElement` and `tryInvMatrix` are trivial wrappers of the built in inversions of scalars and

```
1  local tryInvInner := proc(A)
2      local ka, va, br, bc, ainv, cainv, ainvb,
3            sa, sainv, ainvbsainv;
4
5      ka, va := blockKind(A), blockVal(A);
6
7      if   ka = 'zero' then
8          error singularError
9      elif ka = 'scalar' then
10         scalarBM(nEltRows(A), tryInvElement(va))
11     elif ka = 'matrix' then
12         matrixBM(tryInvMatrix(va))
13     elif ka = 'rblock' then
14         br, bc := nBlockRows(A), nBlockCols(A);
15         if br = 1 and bc = 1 then
16             rblockBM([[tryInvInner(va[1,1])]])
17         elif br = 2 and bc = 2 then
18             sa  := schurComp(A, 1, 1, tryInvInner,
19                              ainv, cainv, ainvb);
20             sainv      := tryInvInner(sa);
21             ainvbsainv := times(ainvb, sainv);
22             rblockBM([
23                 [plus(ainv, times(ainvbsainv,cainv)),
24                  neg(ainvbsainv)],
25                 [neg(times(sainv, cainv)),
26                  sainv]
27             ])
28         else
29             error "Unhandled block structure";
30         fi
31     else
32         never("Malformed BlockMatrix", A)
33     fi
34 end:
```

Fig. 5. Implementation of `tryInv`

matrices that likewise raise `singularError` as required. On line 19 the function `schurComp` is called with all the optional arguments to receive the auxiliary matrix products useful in computing the inverse.

The function `invByMTM` computes $(M^*M)^{-1}M^*$ using `tryInv`. If M is invertible, then the cal to `tryInv` will not fail.

V. PLU Decomposition

One of the goals of this work was to implement the ideas of [10], 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 elementwise, not just block-wise, and we take the disambiguating convention that the diagonal of L contains 1s.

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 [10], if

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

and we seek $M = LU$ such that

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

with L_1 and L_2 lower triangular and U_1 and U_2 upper triagular, 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 6. Notice that we are able to use the functions described so far to provide relatively easy to read, natural code.

```
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 u1inv      := inv(u1);                 # n1 x n1
21 x          := times(c, u1inv);         # 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 neede 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. 6. 2×2 `rblock` case of `PLU_Decomp`

VI. 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 is shown in Figures 7, 8 and 9. Figure 7 shows reading the file that defines the `BlockMatrix` package, sets some abbreviations and defines `B55` to be a square block matrix. Figure 8 shows the inversion of `B55`. This is a simple case since the block B_{11} is non-singular. The figure also shows a verification step that the matrix times its inverse is the identity. Figure 9 shows the factorization of `B55` into its PLU decomposition.

The second example, given in Figure 10, shows the inversion of a block matrix where the principal minors are singular.

VII. Future Work

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

- matrix transpose (non-Hermitian!)
- re-organizing block matrices to desired compatible shapes
- inversion of other than 1×1 and 2×2 block matrices, by grouping
- 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 $(M^*M)^{-1}M^*$ method *versus* preconditioning by a matrix other than M^*.

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.

```
> restart
                    "SrcDir is /Users/smwatt/Data.local/Documents/Lib/Maple"                    (1)
> readsrc("BM")
> pBM := BlockMatrix:
  BM := pBM:-BM:
> B55 := 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]]) ]]
  );
```

$$B55 := \_BM\left(5,5,rblock,\begin{bmatrix}\_BM\left(2,2,matrix,\begin{bmatrix}0&2\\1&2\end{bmatrix}\right) & \_BM\left(2,3,matrix,\begin{bmatrix}0&1&2\\1&0&1\end{bmatrix}\right)\\ \_BM\left(3,2,matrix,\begin{bmatrix}3&0\\2&1\\1&2\end{bmatrix}\right) & \_BM\left(3,3,matrix,\begin{bmatrix}4&0&4\\4&0&1\\1&4&0\end{bmatrix}\right)\end{bmatrix}\right) \quad (2)$$

Fig. 7. Definition of a block matrix

```
> pBM:-inv(B55)
```

$$\_BM\left(5,5,rblock,\begin{bmatrix}\_BM\left(2,2,matrix,\begin{bmatrix}-\frac{32}{35}&\frac{36}{35}\\\frac{4}{35}&\frac{13}{35}\end{bmatrix}\right) & \_BM\left(2,3,matrix,\begin{bmatrix}\frac{13}{35}&-\frac{24}{35}&\frac{8}{35}\\-\frac{6}{35}&\frac{3}{35}&-\frac{1}{35}\end{bmatrix}\right)\\ \_BM\left(3,2,matrix,\begin{bmatrix}\frac{12}{35}&-\frac{58}{105}\\\frac{3}{35}&-\frac{32}{105}\\\frac{12}{35}&-\frac{23}{105}\end{bmatrix}\right) & \_BM\left(3,3,matrix,\begin{bmatrix}-\frac{19}{105}&\frac{62}{105}&-\frac{3}{35}\\\frac{4}{105}&-\frac{2}{105}&\frac{8}{35}\\\frac{16}{105}&-\frac{8}{105}&-\frac{3}{35}\end{bmatrix}\right)\end{bmatrix}\right) \quad (3)$$

```
> pBM:-times(%,B55)
```

$$\_BM\left(5,5,rblock,\begin{bmatrix}\_BM(2,2,scalar,1) & \_BM(2,3,zero,0)\\ \_BM(3,2,zero,0) & \_BM(3,3,scalar,1)\end{bmatrix}\right) \quad (4)$$

Fig. 8. Inversion of a block matrix

VIII. 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 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] David Poole. *Linear Algebra: a modern introduction*. Cengage, 2015.

[7] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[8] S.M. Watt. Aldor. In *Handbook of Computer Algebra*, page 265–270. Springer Verlag, 2003.

[9] 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.

[10] Stephen M. Watt. Algorithms for recursive block matrices. In *Proc. LALO60: Matrices and Polynomials in Computer Algebra*. also `arXiv:2407.03976v1`, 2024.

```
> P55, L55, U55 := pBM:-PLU_Decomp(B55) :
> P55
```

$$_BM\left(5, 5, rblock, \begin{bmatrix} _BM\left(2, 2, matrix, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\right) & _BM(2, 3, zero, 0) \\ _BM(3, 2, zero, 0) & _BM(3, 3, scalar, 1) \end{bmatrix}\right) \tag{5}$$

```
> L55
```

$$_BM\left(5, 5, rblock, \begin{bmatrix} _BM(2, 2, scalar, 1) & _BM(2, 3, zero, 0) \\ _BM\left(3, 2, matrix, \begin{bmatrix} 3 & -3 \\ 2 & -\frac{3}{2} \\ 1 & 0 \end{bmatrix}\right) & _BM\left(3, 3, matrix, \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & -\frac{8}{9} & 1 \end{bmatrix}\right) \end{bmatrix}\right) \tag{6}$$

```
> U55
```

$$_BM\left(5, 5, rblock, \begin{bmatrix} _BM\left(2, 2, matrix, \begin{bmatrix} 1 & 2 \\ 0 & 2 \end{bmatrix}\right) & _BM\left(2, 3, matrix, \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}\right) \\ _BM(3, 2, zero, 0) & _BM\left(3, 3, matrix, \begin{bmatrix} 1 & 3 & 7 \\ 0 & -\frac{9}{2} & -12 \\ 0 & 0 & -\frac{35}{3} \end{bmatrix}\right) \end{bmatrix}\right) \tag{7}$$

Fig. 9. PLU decomposition of a block matrix

```
> 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]]) ]
  );
```

$$M55 := _BM\left(5, 5, rblock, \begin{bmatrix} _BM\left(2, 2, matrix, \begin{bmatrix} x & 0 \\ 1 & 0 \end{bmatrix}\right) & _BM\left(2, 3, matrix, \begin{bmatrix} 0 & 0 & 0 \\ y & 0 & 0 \end{bmatrix}\right) \\ _BM\left(3, 2, matrix, \begin{bmatrix} 0 & z \\ 0 & 0 \\ 0 & 0 \end{bmatrix}\right) & _BM\left(3, 3, matrix, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & a \\ 0 & b & c \end{bmatrix}\right) \end{bmatrix}\right) \tag{10}$$

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

$$_BM\left(5, 5, rblock, \begin{bmatrix} _BM\left(2, 2, matrix, \begin{bmatrix} \frac{1}{x} & 0 \\ 0 & 0 \end{bmatrix}\right) & _BM\left(2, 3, matrix, \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{z} & 0 & 0 \end{bmatrix}\right) \\ _BM\left(3, 2, matrix, \begin{bmatrix} -\frac{1}{y\,x} & \frac{1}{y} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}\right) & _BM\left(3, 3, matrix, \begin{bmatrix} 0 & 0 & 0 \\ 0 & -\frac{c}{a\,b} & \frac{1}{b} \\ 0 & \frac{1}{a} & 0 \end{bmatrix}\right) \end{bmatrix}\right) \tag{11}$$

```
> pBM:-times(M55, %)
```

$$_BM\left(5, 5, rblock, \begin{bmatrix} _BM(2, 2, scalar, 1) & _BM(2, 3, zero, 0) \\ _BM(3, 2, zero, 0) & _BM(3, 3, scalar, 1) \end{bmatrix}\right) \tag{12}$$

Fig. 10. Inversion of a block matrix with singular principal minors