Monografías Matemáticas *García de Galdeano* **37**, 103–113 (2012)

SIMULTANEOUS TRIANGULATION OF COMMUTING FAMILIES OF MATRICES – WHY AND HOW PRECISELY?

Vanesa Cortés, Juan Manuel Peña and Tomas Sauer

Abstract. We present an algorithm that provides an extension of the QR method in order to compute the joint eigenvalues of a family of commuting real matrices.

Keywords: QR method, commuting matrices, simultaneous triangulation.
AMS classification: 65F15.

§1. Introduction

In this paper, we present the algorithm that describes an extension of the QR method to simultaneously compute the joint eigenvalues of a finite family of commuting matrices defined in [1]. This problem is motivated by the task of finding solutions of polynomial systems of equations of the form

$$F(X) = 0, \qquad F = (f_1, \dots, f_m), \qquad f_j \in \mathbb{C}[x_1, \dots, x_n].$$

The idea behind this approach is to extend the *Frobenius companion matrix* to the multivariate case. Recall that if $f(x) = a_0 + a_1 x + \cdots + a_n x^n$, $a_n \neq 0$, is a polynomial in one variable, then its zeros are the eigenvalues of the *companion matrix*

$$A := \begin{bmatrix} 0 & & & & -a_0/a_n \\ 1 & 0 & & & -a_1/a_n \\ & \ddots & \ddots & & \vdots \\ & & 1 & 0 & -a_{n-2}/a_n \\ & & & 1 & -a_{n-1}/a_n \end{bmatrix}.$$

Since this result can be proved, for example, by division with remainder and considering multiplication modulo the (principal) ideal generated by f, it allows for an extension via computational ideal theory, especially the concept of Gröbner bases or H–bases. However, in n variables one does not have to find the eigenvalues of a single matrix A but the *joint eigenvalues* of a system $\mathcal{A} = (A_1, \dots, A_n)$ of *commuting* matrices, i.e. $A_j A_k = A_k A_j$. A naive and direct approach would be to compute the eigenvalues and eigenvectors for each of the matrices separately and then connect the eigenvalues (which are the components of one of the zeros) by means of the associated eigenvectors. This approach, however, faces difficulties as soon as the coordinate projections of the solutions are not well separated as then the eigenvectors will usually not be unique any more, hence the connection can only be made via a numerically instable intersection of eigenspaces. And while multiple eigenvalues do not

constitute a thread for the *QR* method of eigenvalue dermination, cf. [3], clustered eigenvalues are a numerical problem, hence should be avoided. To overcome these difficulties, we proposed a method that extends the QR method and the included concept of splitting matrices by treating the matrices as simultaneous as possible, making use of the fact that under certain circumstances a *split*, that is, a separation of eigenspaces, obtained for one of the matrices from the family carries over to the whole family. The eigenvalue approach applies only to the case when the set X of solutions to $F(X) = 0$ is *finite*, or, in algebraic terminology, if the ideal generated by F is zero dimensional. Then the size of the matrices is the number of zeros which is determined automatically by the algebraic reduction process.

The following lemma, proved in [1], provides the condition that will allow us to perform the simultaneous triangularization process of all commuting matrices.

Lemma 1 (Cf. [1], Lemma 2.1)**.** *Let A, B be two real $n \times n$ matrices such that $AB = BA$. If there exists a nonsingular matrix P such that*

$$P^{-1}AP = \begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix},$$

where the $p \times p$ ($1 \leq p \leq n-1$) matrix A_1 and the $(n-p) \times (n-p)$ matrix A_3 satisfy $Spec(A_1) \cap Spec(A_3) = \emptyset$, then

$$P^{-1}BP = \begin{bmatrix} B_1 & B_2 \\ 0 & B_3 \end{bmatrix} \tag{1}$$

and B_1 is a $p \times p$ matrix.

In Section 2, we comment the steps of the algorithm and present the core `MATLAB` code of each one. In Section 3, we add the subroutines used in the main program.

§2. Algorithm

Let $\mathcal{A}$ be a set of m real commuting matrices $n \times n$ such that $A, B \in \mathcal{A}$ implies $AB = BA$. Besides, let *threshold*, *toleranceEig* be variables holding (small) positive numbers used as thresholds in the algorithm. We apply the following algorithm in order to find the orthogonal matrix P that has to exist by Lemma 1.

Step 1. Storing and ordering matrices. The matrices of $\mathcal{A}$ which have only one eigenvalue or a unique pair of complex eigenvalues will be recognized by a routine `uniqueEigenvalue` and will not be considered for decomposition. The other ones will concatenated into a single matrix.

```
j=0;
for i=1:m
    [H]=hessenberg(A(:,(i-1)*n+1:(i)*n));
    [only]=uniqueEigenvalue(H,n,threshold,toleranceEig);
    if(only==0)
        j=j+1;
```

```
        A(:,(j-1)*n+1:(j)*n)=𝒜(:,(i-1)*n+1:(i)*n);
    end -end if
end -end for
```

The assumption that the ideal is zero dimensional and that there is more than one solution of $F(X) = 0$ ensures that $\mathcal{A}$ is nonempty.

We then compute quantities that will allow us to order the matrices of $\mathcal{A}$ according to their spread. The *spread* is an easily computed estimate how widely the eigenvalues of a given matrix vary. A matrix with large spread is more likely to have well–separated eigenvalues and hence the *QR* iterations will be expected to lead to a separation of eigenspaces after a smaller number of iterations. To estimate the spread of the eigenvalues of a matrix in $\mathcal{A}$, we use Gerschgorin circles as well as an estimate based on determinant and trace of the matrix, cf. [1].

```
for i=1:j
    A=A(:,(i-1)*n+1:(i)*n);
    L(i)=Gerschgorin(A);
    F(i)=means(A);
    G(i)=max(L(i),F(i));
end -end for
```

The matrix **A** is then reordered according to the values in G. This is in fact crucial as a large spread makes an early split after only a few iterations more likely.

Step 2. Initializing variables for starting the process. We will initialize a variable `matrix` with the first unused matrix in **A** and other variables (some of them related to possible found blocks on the process) that will play an important role through the algorithm. Since we also consider parts of the matrices after a *QR*–split, the size of the theses blocks is stored in a variable `newBlockMatrix` which is initialized with the full matrix.

```
newBlockMatrix=[1,n,1];
matrix=A(:,(i-1)*n+1:i*n); initialMatrix=A(:,(i-1)*n+1:i*n);
auxStoringQ=eye(n); change=0;
```

Step 3. Selecting the matrix which starts the process with the process matrix. A variable `processMatrix` will be initialized with the first unused matrix in **A**, and transformed into Hessenberg form with the routine `hessenberg` based on Householder reflections and stored again in the variable `processMatrix`.

Step 4. Selecting the process matrix if it has not order n. If we have to consider `blocksNum` subblocks of `matrix`, we extract those into a variable `processMatrix` to be treated by a *QR* factorization thereafter.

```
for v=1:blocksNum
    p=blockMatrix(v,1);q=blockMatrix(v,2);
    processMatrix=matrix(p:q,p:q);
    processMatrix=hessenberg(processMatrix);
    storingQ=eye(q-p+1);
    auxQ=eye(length(processMatrix));
    Process the matrix ...
end
```

Step 5. Initial transformation of the process matrix when it has trace zero. If the matrix `processMatrix` has trace zero, then we first perform two consecutive steps of the shifted QR algorithm with shifts (using the routine `qrShift`) in order to make our spread estimation work. The tolerance of 10^{-10} is just chosen as an example and can be adapted if necessary.

```
if(abs(trace(processMatrix))<=1e-10)
    [processMatrix,storingQ1]=choosingShift(processMatrix);
    storingQ=auxQ*storingQ1;
    auxQ=storingQ;
end -end if
```

Step 6. Localizating an special subdiagonal element to find an eigenvalue. If some subdiagonal element in `processMatrix` has an absolute value less than a given `tolerance`, that is, the matrix is already "almost split", we will apply a step of the *QR* algorithm with a shift whose value will be given by the routine `choiceShift`. As is well–known, a properly chosen shift will significantly improve the performance of the *QR* iteration, cf. [2]. This process will be continued until either we get the absolute value of this subdiagonal element smaller than a positive number `threshold`, less than `tolerance`, of course, or until we arrive at a maximal number of iterations.

```
[subdiagonal]=subdiagonal(processMatrix,w);
[j,lessThanTolerance] = min(subdiagonal <= tolerance);
while(subdiagonal(j)>threshold and approxNum<=n)
    [shift]=choosingShift(processMatrix,j);
    [processMatrix,obtainedQ]=qrShift(processMatrix,shift,w);
    storingQ=auxQ*obtainedQ;
    auxQ=storingQ;
```

```
    matrix(p:q,p:q)=processMatrix;
    subdiagonal(j)=abs(processMatrix(j+1,j));
    approxNum = approxNum + 1;
end
if(subdiagonal(j)<=threshold)
    first=p; last=p+j-1; first1=p+j; last1=q;
    lessThanThreshold=1;
    foundBlock=1;
    matrix(p:q,p:q)=processMatrix;
else
    lessThanThreshold=0;
end -end if
```

In the first case, we have obtained a candidate for the split and we continue with the process described in Step 7, while in the second case we continue as described in Step 8. Keep in mind here that according to Lemma 1 a split is only useful as a common split if it decomposes the `processMatrix` in such a way that the spectra of the submatrices are disjoint.

If there is no such a subdiagonal element, on the other hand, we will apply one step of the *QR* algorithm (using the routine `qr` of `MATLAB`) and repeat the above process up to reach a maximal number of iterations (depending on the order of the matrix in `processMatrix`).

```
if(lessThanTolerance==0)
    [Q,R]=qr(processMatrix);
    storingQ=auxQ*Q;
    auxQ=storingQ;
    processMatrix=R*Q;
    matrix(p:q,p:q)=processMatrix;
end -end if
if(lessThanThreshold==0)
    [processMatrix,storingQ1]=choosingShift(processMatrix);
    storingQ=auxQ*storingQ1;
    auxQ=storingQ;
    matrix(p:q,p:q)=processMatrix;
end -end if
```

If after a certain number of iterations which depends on the order of the matrix in `processMatrix` we still did not obtain a split, we pick the next unused matrix from **A**, store it in `processMatrix` and restart the iteration.

Step 7. Storing the matrix obtained in the previous process. In a variable called `storingMatrix` of order n, we store the `processMatrix` at the same relative position with respect to the `initialMatrix`. Initially the variable `storingMatrix` is a zero matrix.

```
newstoringQ=eye(n);
newstoringQ([p:q],[p:q])=storingQ;
auxStoringQ=auxStoringQ*newstoringQ;
storingMatrix=auxStoringQ;
```

Then we split this `storingMatrix` into two blocks with "break" at the relative position of the subdiagonal element from Step 6 with absolute value less than the variable `threshold` in the `storingMatrix` and compute the spectra of these blocks:

```
totalBlocksNum = totalBlocksNum + 1;
newBlockMatrix=zeros(totalBlocksNum,3);
[takeFirstBlock]=consideredBlock(matrix, first, last);
[takeSecondBlock]=consideredBlock(matrix, first1,last1);
[newBlockMatrix]=doingNBM(v, first, last, first1, last1, takeFirstBlock,
                          takeSecondBlock, totalBlocksNum, oldBlockMatrix)
[endProcess]=spectraIntersection(matrix, last, first1, toleranceEig);
```

If the spectra are disjoint, then the joint triangularization process is essentially finished and we just have to perform some conclusive computations as described in Step 9. If the spectra are not disjoint and we can split the `storingMatrix` into blocks of order 1 or 2, these final blocks contain either a single real value to be checked by the routine `uniqueEigenvalue` or a pair of conjugate complex eigenvalues. Then we carry out an appropriate symmetric permutation in order to separate the spectra of the blocks. So, we are either in the situation of Step 3 again, now with the freshly computed `storingMatrix`, which will then be stored in the variable `processMatrix`, or the procedure pointed out in Step 4 will be applied to the `storingMatrix`, as the variable `processMatrix`, with properly chosen blocks of this matrix to be considered in Step 4. If one of these blocks has only a unique eigenvalue or a block of order two with a couple of conjugate eigenvalues, we do not consider this block in Step 4.

Step 8. Not all the eigenvalues of the process matrix have been located. If we have not found all the eigenvalues of the matrix in spite of performing the *QR* algorithm with shifts the maximal number of iterations given by the order of the variable `processMatrix`, then we apply a step of the *QR* algorithm with shift. If all the subdiagonal elements are greater than the `tolerance`, then we select a new matrix of the family **A**. If there exists a subdiagonal element with absolute value less than the `tolerance`, we switch to Step 6 which we can repeat until we arrive at a maximal number of iterations given by the order of the `processMatrix`. If this maximal number of iterations is exceeded, then we again choose a new matrix of the family **A**. This can be performed cyclically and after each cycle the maximal number of iterations can be raised, if necessary.

Step 9. Computing the ortogonal matrix P**.** In the end, after the iterations have been performed, we calculate the product of all the matrices corresponding to all the intermediate

transformations carried out over the `initialMatrix` in order to obtain the orthogonal matrix P mentioned in Lemma 1.

§3. Subroutines

In this section we give more detailed descriptions of the subroutines used in our algorithm. As an easy starting point we list the very simple function that just checks whether a matrix is normal, that is, whether $A^TA = AA^T$.

The subroutine that chooses the shift for the QR step is already more complicated as the choice of the shift depends on whether the matrix is normal and the shift also has to be adapted to the eigenstructure of the lower left 2×2 block of B.

choosingShift

```
[processMatrix,storingQ]=choosingShift(B)
    N=length(B); storingQ=eye(N); a=eig(B([N-1:N],[N-1:N]));
    if( checkingNormalMatrix(B,N)==1 )
        shift1=sqrt(norm(B,'fro')/N);
        [Q,R]=qr(B-shift1*eye(N));
        storingQ = storingQ * Q;
        processMatrix=R*Q+shift1*eye(N);
    else
        if(abs(imag(a(1)))<1e-10)    % The real case
            if( max( abs( real(a) ) < 1e-10 )
                newShift=norm(B,inf)/N;
                [Q,R]=qr(B-newShift*eye(N));
                storingQ = storingQ * Q;
                processMatrix=R*Q+newShift*eye(N);
            else
                [Q,R]=qr(B-maximum*eye(N));
                storingQ = storingQ * Q;
                [Q,R]=qr( R*Q );
                storingQ = storingQ * Q;
                processMatrix=R*Q+maximum*eye(N);
            end -end if
        else    % The complex case
            if(N =2)
                [Q,R]=qr(B-a(1)*eye(N));
                storingQ = storingQ * Q;
                [Q,R]=qr( R*Q + ( a(1) -a(2) ) * eye(N));
                processMatrix=R*Q+a(2)*eye(N);
                storingQ = storingQ * Q;
            else
```

```
                processMatrix=B;
            end -end if
        end -end if
    end -end if
```

The next subroutine, `consideredBlock`, checks whether a given block of a matrix needs to be considered for the *QR* method. This is the case if the block is not of size 1 or does not have a unique single eigenvalue, the latter being checked by the subroutine `uniqueEigenvalue` which does a quick check whether that is the case.

consideredBlock

```
[takeFirstBlock]=consideredBlock(matrix, first, last)
    firstBlock=matrix(first:last,first:last); takeFirstBlock=1;
    l1=length(firstBlock);
    if l1==1
        takeFirstBlock=0;
    elseif uniqueEigenvalue(firstBlock,last-first+1,threshold,toleranceEig) == 1
        takeFirstBlock=0;
    else
        if l1==2 && eig(firstBlock)  = 0
            takeFirstBlock=0;
        end -end if
    end -end elseif
```

In `doingNBM`, the block matrices are extracted and the so far unused ones "advance" one position in the queue.

doingNBM

```
[newBlockMatrix]=doingNBM(v,first, last, first1, last1, takeFirstBlock,
                    takeSecondBlock, totalBlocksNum, oldBlockMatrix)
    if(v==1)
        newBlockMatrix(1,:)=[first,last,takeFirstBlock];
        newBlockMatrix(2,:)=[first1,last1,takeSecondBlock];
        if(totalBlocksNum>2)
            newBlockMatrix(3:totalBlocksNum,:)=oldBlockMatrix(2:totalBlocksNum-1,:);
        end -end if
    else
        newBlockMatrix(1:v-1,:)=oldBlockMatrix(1:v-1,:);
        newBlockMatrix(v+2:totalBlocksNum,:)=oldBlockMatrix(v+1:totalBlocksNum-1,:);
        newBlockMatrix(v,:)=[first,last,takeFirstBlock];
```

```
        newBlockMatrix(v+1,:)=[first1,last1,takeSecondBlock];
    end -end if
```

The subroutine `Gerschgorin` estimates the spectrum of A by means of Gerschgorin circles, cf. [2].

Gerschgorin

```
[dif,maxi,mini]=Gerschgorin(A)
    n = length( A );
    r = abs( A ) * ones( n,1 ) - abs( diag( A ) );
    maxi = max( diag(A) + r );
    mini = min( diag(A) - r );
    dif = maxi - mini;
```

In `hessenberg` the matrix is transformed into Hessenberg form by means of Householder transforms, see again [2]; the extra part is to apply the same transformations to the variable `storedMatrix` that has to be processed in the rest of the method.

hessenberg

```
[H]=hessenberg(A)
    n=length(A);
    storedMatrix=A;
    for i=1:n-2
        v=zeros(n-i,1);
        if(sign(storedMatrix(i+1,i))>=0)
            v=norm(storedMatrix(i+1:n,i),2)*eye(n-i,1)+storedMatrix(i+1:n,i);
        else
            v=-norm(storedMatrix(i+1:n,i),2)*eye(n-i,1)+storedMatrix(i+1:n,i);
        end -end if
        v=v/(norm(v,2));
        P=eye(n-i)-2*v*v';
        U=eye(n);
        U([i+1:n],[i+1:n])=P;
        storedMatrix=U*storedMatrix*U';
    end -end for
    H=storedMatrix;
```

In `means` the difference between the algebraic and geometric means of a matrix is determined; in a really performant version of the algorithm (which will, of course, not use `matlab`), the computation of the determinant can be done more efficiently.

means

```
[dif]=means(A)
    n=length(A);
    mA=abs( trace(A)/n );
    mG=abs( det(A) )^(1/n);
    dif=abs( mA - mG );
```

The shifted *QR* method as in `qrShift` is simply standard.

qrShift

```
[B,E]=qrShift(A,a,n)
    [Q,R]=qr(A-a*eye(n));
    E=Q;
    B=R*Q+a*eye(n);
```

The subroutine `uniqueEigenvalue` is slightly more tricky as it tries to figure out whether a matrix A has only one, unique, real eigenvalue or, a single complex conjugate pair of eigenvalues. Also keep in mind that we assume that any input A will be in Hessenberg form so that all the computations below are relatively cheap. The first step it to consider the number $a = (1/n)\operatorname{trace} A$ which would be a guess for the single real eigenvalue and the real part of the complex eigenvalue. To check whether a is indeed a single real eigenvalue, we perform one *QR* step on a with shift a. If a were such an eigenvalue, the resulting matrix would be upper triangular with a on the triangle.

If that is not the case, we have to check whether there is a single complex conjugate pair $a \pm ib$ of eigenvalues which requires that the size n of the matrix is even. Then $\det A = (a^2+b^2)^{n/2}$ and we can guess b via $b^2 = (\det A)^{2/n} - a^2$, where for the determinant computation we can re–use the matrix R from the real *QR* decomposition. The we perform either a real *QR double step*, cf. [2], with the guessed eigenvalue $a + ib$ and check whether the result is block diagonal, or, for simplicity, we can do a *complex QR* step and again check for diagonality of the resulting complex matrix.

uniqueEigenvalue

```
[a]=uniqueEigenvalue(A,n,toleranceE)
    a=0;
    possibleEig=(trace(A))/n;
    [Q,R]=qr( A - possibleEig * eye(n) );
    auxMatrix = R * Q + possibleEig * eye(n);
    if ( norm( tril( auxMatrix - possibleEig * eye( n ) ) ) < toleranceE )
```

```
    a = 1;
elseif rem( n,2 ) == 1
    a = 0;
else    % Check for complex
    possibleIm = sqrt( prod( diag(R).^2 )^(1/n) - possibleEig^2 );
    possibleEig = possibleEig + I * possibleIm;
    [Q,R]=qr( A - possibleEig * eye(n) );
    auxMatrix= R*Q + possibleEig * eye(n);
    if ( norm( tril( auxMatrix - possibleEig * eye( n ) ) ) < toleranceE )
        a = 1;
    end
end
```

§4. Acknowledgement

Work partially supported by the Spanish Research Grant MTM2009-07315 and by Gobierno de Aragón and Fondo Social Europeo.

References

[1] Cortés, V., Peña, J. M., and Sauer, T. Simultaneous triangularization of commuting matrices for the solution of polynomial equations. *Central European Journal of Mathematics 10*, 1 (2011), 277–291.

[2] Golub, G., and van Loan, C. F. *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, 1996.

[3] Wilkinson, J. H. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965.

Vanesa Cortés and Juan Manuel Peña
Departamento de Matemática Aplicada
Universidad de Zaragoza
E-50009 Zaragoza, Spain
vcortes@unizar.es, jmpena@unizar.es

Tomas Sauer
Lehrstuhl für Numerische Mathematik
Justus–Liebig–Universität Gießen
Heinrich–Buff–Ring 44
D-35392 Gießen, Germany
tomas.sauer@math.uni-giessen.de