Revision of Schur Module

S. de Graaf

Circuits and Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
Delft University of Technology
The Netherlands

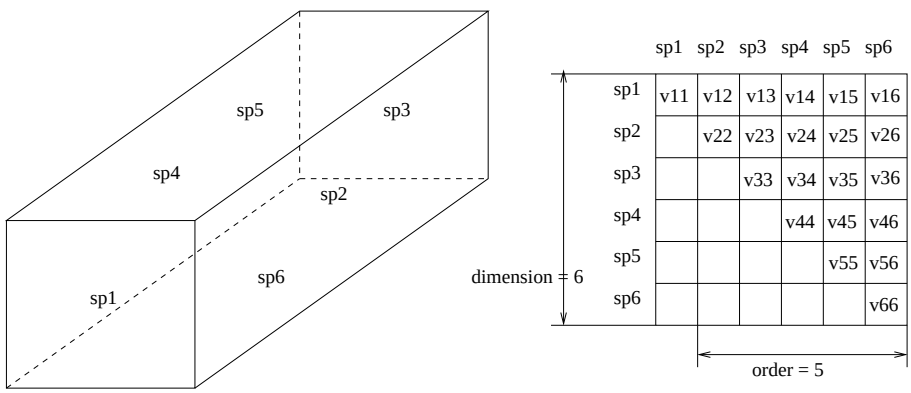
Report EWI-ENS 11-01
March 9, 2011

Copyright © 2011 by the author.
All rights reserved.

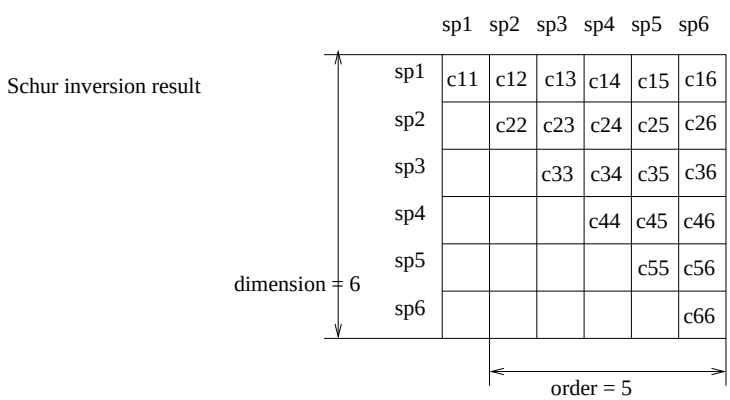
Last revision: March 14, 2011.

1. INTRODUCTION

The *space3d* program uses the Schur module for 3D capacitance extraction. The Schur module inverts a matrix of Green values where after couple capacitances between conductor points are the result. The conductor points are laying somewhere in the dielectrical medium and are laying close enough to each other to calculate a couple capacitance value. The conductor points are called spiders, because they are laying in the center of some conductor face. The faces are the outerside 3D conductor boundary parts. For example a 3D conductor cube has at least 6 faces. And can be classified as 4 sidewall faces and 1 top and 1 bottom face. When all spiders are in the same window, then a Green value is calculated for each pair of spiders. The first spider can be paired with the five other spiders. And the second spider be paired with four other spiders. An so on. The resulting Green values can be placed in an upper matrix of order 5 (dimension 6). See the figure below.

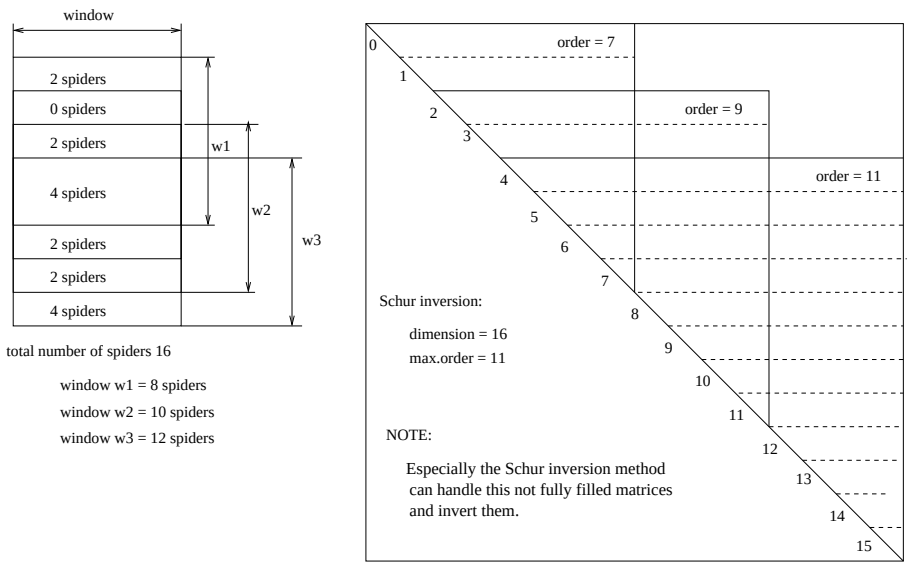


Because of the symmetric case, some calculated Green values must be equal. For example v12, v14, v23 and v34 must be equal. After Schur inversion of this Green values matrix the result is again an upper matrix. See the figure below.



The diagonal values must be positive and the off-diagonal values must be negative. The value c11 can be seen as a capacitance value between point sp1 and ground. The value

c12 can be seen as a couple capacitance value between points sp1 and sp2. And the c12 value is also substracted from the sp1 and the sp2 ground cap value. Note that possibly sp1 and sp2 are connected with the same conductor node, because the spiders are laying on the same condcutor. In that case the couple capacitance value of c12 does not give a capacitance element. And in that case only the ground capacitances give a capacitance element.

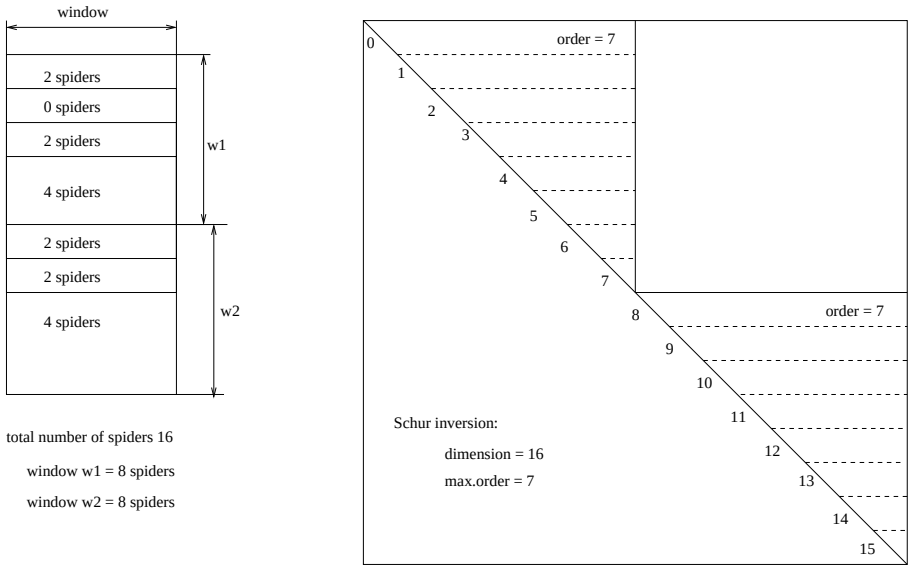


In the above figure you see a typical Schur matrix which is filled based on the spider points which are laying in the chosen cap3d.be_window. In this case there are in the y-direction 3 overlapping windows. The last window contains most spider points (12). Thus the maximum order of this Schur inversion is 11. And because window w1 has overlap with window w3 the maximum internal rows is 16. For the memory allocation is used the max. possible internal rows, that is 2 * max.order + 1 (dMax). In this case the dimension is smaller, thus there needs only be memory for 16 rows. You can calculate the needed amount of Schur memory in bytes (dMax=16, oMax=12) as follows:

array	formula	amount
IN/OUT	dMax * oMax * 8 + dMax * 4	1600
M/V	oMax * oMax * 8 + oMax * 4	1200
Order	dMax * 4	64
DIAG	dMax * 8	128
P/P1	2 * oMax * 8	192
	Needed memory in bytes total:	3184

(a double costs 8 bytes and a pointer/integer 4 bytes)

In the Schur module revision the needed amount of memory is reduced with a factor of 2. This was possible by combining the IN and OUT array together and also combining the M and V array. Also the use of a separate schurOut vector was not needed. And now also the Schur dimension is used in place of 2 * max.order + 1, when that is less.



In the new example above you see the filled Schur matrix in case the 2 windows in the y-direction are not overlapping. This two sub-matrices can completely independed be inverted. The maximum number of internal rows is in this case 8 (dMax). Thus, the total needed amount of Schur memory is in this case 1872 bytes. Note that row 0 can only be executed when Order (7) next rows are read-in. Thus, after row 7 is read-in, all rows (0 - 7) can be executed. And note that a row can only be outputted, for example row 0, when the Order of that row next rows are already executed. Thus, after row 7 is executed can all rows (0 - 7) be outputted.

Note that when in the example above the two windows have the smallest possible overlap of 1 row, for example when row 7 and row 8 overlap each other. The order of row 7 changes from 0 into 8, because window w2 contains now 9 spider points. Thus, row 7 can now only be executed when row 15 is read-in. This explains why the maximum number of internal rows is maximal equal to 2 * max.order + 1 and it is save to use that number to allocate memory. Before a schurInit had a more accurate value be calculated, but this is not done. Note that normally a second schurInit uses two times more Schur memory, because an inversion of a double window is done.

Note that only the Schur method can invert this not fully filled matrices. For the LU decomposition method is a fully filled matrix needed. And also by a fully filled matrix is the Schur method faster.

2. USE OF SCHUR MODULE

To use the Schur module in a program source file, you need to #include the file "space/schur/export.h". The following declarations can be found in this include file:

```
void initSchur (int maxr, int maxo);
void schurRowIn (int kr, schur_t *r, int ord);
void schurStatistics (FILE *fp);
void printUpperMatrix (FILE *fp, int k, schur_t *r, int ord);
extern bool_t schurShowProgress;
```

Now, the initSchur function has get an extra argument (maxr). This argument specifies the maximum (or last) row number and is equal to the matrix dimension - 1. In your program, you need to have a schurRowOut function with the following prototype:

```
void schurRowOut (int k, schur_t *r, int ord);
```

Thus, a simple Schur matrix inversion program can look like this:

```
int main ()
{
    FILE *fp_in = fopen ("AU512", "r");
    readUpperMatrix (fp_in);

    initSchur (dimension - 1, maxorder);
    for (r = 0; r < dimension; ++r) schurRowIn (r, In[r], Order[r]);

    schurStatistics (stderr);
    return (0);
}

void schurRowOut (int row, schur_t *buf, int nr_cols)
{
    printUpperMatrix (stdout, row, buf, nr_cols);
}
```

The readUpperMatrix function must allocate memory for the "In" array and "Order" vector and must set the "dimension" and "maxorder". The printUpperMatrix function can print a row in a standard format for you and flush the output. On the end of the program you can ask for printing of schurStatistics. The global "schurShowProgress" variable can be set, if you want to see Schur progress information. This information is written by the schurRowIn function to stderr.

Note that initSchur is needed each time you want to start a Schur matrix inversion. It allocates enough memory for the matrix inversion and init some used variables. By a second call to initSchur the old allocated memory can possibly be reused or new memory must be allocated, because there is too less allocated before. The initSchur argument "maxo" must be >= 0 and "maxr" must be >= "maxo". The schurRowIn function must supply the rows in correct order to the Schur module (starting with row number 0). The specified order of the row must be correct and must be >= 0. The order may not be less than the previous order - 1.

The working of the Schur module can be explained on the hand of the following source code fragments:

```
void initSchur (int maxr, int maxo)
{
    if (maxo > globalMaxo) { /* init or increase memory */
        max_used = newSchurMem (maxr, maxo);
        globalMaxo = maxo;
    }
    maxrow = maxr;  maxorder = maxo;  schur_row = k = k2 = 0;
    calls++;
}

void schurRowIn (int kr, schur_t *r, int ord)
{
    n = kr - k2;
    for (j = 0; j <= ord; ++j) scIN[n][j] = r[j];
    scOrder[n] = ord;
    scDIAG[n] = 1 / sqrt (r[0]);

    while (kr - k >= scOrder[k - k2]) {
        execSchurRow (); /* execute schur for row k */
        if (++k > kr) break;
    }

    if (kr == maxrow) { /* last row */
        if (kr >= k) say ("Ho, not all rows computed"), die ();
        schur_row = -1;
    }

    while (k - k2 > scOrder[0]) { /* compute the entries of row k2 */
        inSave = scIN[0];
        for (j = 0; j <= scOrder[0]; ++j) {
            val = 0;
            for (n = j; n <= scOrder[0]; ++n)
                val += scIN[n][maxorder - n] * scIN[n][maxorder - n + j];
            inSave[j] = val;
        }
        schurRowOut (k2, inSave, scOrder[0]);
        if (++k2 == k) break;
        for (n = 0; n <= kr - k2; ++n) { /* push memory */
             scIN[n] = scIN[n+1];  scDIAG[n] = scDIAG[n+1];
             scOrder[n] = scOrder[n+1];
        }
        scIN[n] = inSave;
    }
}
```

You see that schurRowIn calls schurRowOut when "k - k2 > scOrder[0]". Position scOrder[0] is the order of row k2, which may only be outputted when row k is executed. The memory of scIN, scDIAG and scOrder is each time shifted after schurRowOut. This is done because schurRowIn wants to reuse the memory of the ready row.

You can also see that "inSave" is set to "scIN[0]" and that "inSave" is used as the output buffer. Thus, no separate output buffer needs to be allocated.

After execSchurRow the test for "k > kr" is added, because when this happens the test against scOrder[] must not be done, because scOrder[] is not yet set. Such test is also added after schurRowOut, because k2 can never be greater than k.
Note that also a test for the last row is added to the code. When the last row is read-in all rows must be executed. And only when all rows are executed then the rows can all be outputted.

What is more changed? Function execSchurRow is changed to make the code faster (> 10% speed-up). Also the arrays scV and scM could be combined together because they are a lower and an upper triangular matrix. Matrix scOUT is also combined with scIN, this is possible because scIN is copied to the scV matrix after calculations. Note that all global used matrices have get a leading "sc" prefix.

All the "schur.xxx" parameters are now obsolete. The setting of these parameters was once tested in function initSchur. The use of these parameter settings is only useful for the Schur module test program. Thus is it not useful to try to use the LU matrix inversion method. Because it has known limitations and it is much slower. For the test program, i had to make changes to let the LU method working again. I added the TEST_SCHUR compile define, thus that the LU code and other test code in schurRowIn is only added to the test program. Also library file "schur.a" does not contain the LU functions anymore. Code fragment of schurRowIn:

```
#ifdef TEST_SCHUR
    if (luFact) {
        if (kr == maxorder) { /* complete matrix is known */
            LU (maxorder + 1); /* perform LU decomposition */
        }
        return;
    }
#endif
```

The memory counting facility for the schurStatistics function is revised. Now the exact number of used bytes is calculated by the newSchurMem function. In the old situation you was not getting the correct information. Note that this function has also get the maxrow argument and if needed shall use maxrow in place of 2 * maxorder, when smaller. Also the test program has get a new option to print the statistics.

Note that the allocated memory is not more "random" initialized to 999. This is completely unneeded, we must only be careful not to use or test unset elements (like "scOrder" mentioned before). All the allocated memory is now freed before new memory is allocated. This makes it possible to reuse a part of the freed memory.

Note that the LU method is not realy using scPIV at this moment. To save memory scOrder is used for the scPIV vector and the scV matrix is replaced by the scIN matrix.

3. THE SCHUR MODULE TEST PROGRAM

The Schur module test program can now be compiled for every configuration. The CMAKE file is changed (also the TEST_SCHUR compile_defines is added).
To compile, give the following command:

```
% cmake -v testschur
```

The made program is called "schur" and is copied to the configuration "bin" directory.
To request help give simply the following command:

```
% schur

Usage: schur [options] infile [ order ]

arguments:
  infile: A positive definite (partly specified) input matrix.
          Default, the input matrix is specified such that the upper
          triangular part of each (partly specified) row starts on a
          new line.  Example:
            a11 a12 a13
            a22 a23 a24
            a33 a34
            a44
   order: An optional bandwidth (order=0 means only the main diagonal,
          order=1 means main diagonal + first upper and lower diagonal).

options are:
      -t: The inputfile is of a form like:
             4
             a11 a12 a13 a14
             a21 a22 a23 a24
             a31 a32 a33 a34
             a41 a42 a43 a44
      -b: The band of the matrix is specified as a vector in a file
          called 'infile.b', where infile is the first program argument.
      -l: Perform LU decomposition to invert the matrix; in this
          case the input matrix must be a full matrix.
      -s: Print statistics about matrix inversion.
      -d: Print debug information (not for option -l, use this
          option twice to get also a 'result' file).
      -f: Print full matrix (in case of not -t).
      -u: Print upper triangular part (in case of -t).

The program produces an output matrix that has a similar form as
the input matrix.
```

New are the options -s, -f and -u. The -h option is obsolete and the debug option -d has been changed.
For example, use the following input file and see what the inversion result is:

```
% cat AU
5 1 1 1
5 1 1
5 1
5
```

Schur matrix inversion result for the "AU" file:

```
% schur -s AU
schur: dimension found =    4
schur: max order found =    3
schur: using max order =    3
2.187500e-01 -3.125000e-02 -3.125000e-02 -3.125000e-02
2.187500e-01 -3.125000e-02 -3.125000e-02
2.187500e-01 -3.125000e-02
2.187500e-01

SCHUR TIME 0.00 s

schurStatistics:
    schur calls        : 1
    max. dimension     : 4
    max. maxorder      : 3
    max. int. rows     : 4
    max. matrix memory : 400
```

New is also the SCHUR TIME information (user time) which is given.
Matrix inversion result for the LU method:

```
% schur -l AU
schur: dimension found =    4
schur: max order found =    3
schur: using max order =    3
USING LU DECOMPOSITION
2.187500e-01 -3.125000e-02 -3.125000e-02 -3.125000e-02
2.187500e-01 -3.125000e-02 -3.125000e-02
2.187500e-01 -3.125000e-02
2.187500e-01
```

Schur matrix inversion result using only the 0-th order:

```
% schur AU 0
schur: dimension found =    4
schur: max order found =    3
schur: using max order =    0
2.000000e-01
2.000000e-01
2.000000e-01
2.000000e-01
```

The inversion results of a full matrix with dimension 528:

```
% schur -s AU528 > z1
schur: dimension found =  528
schur: max order found =  527
schur: using max order =  527

SCHUR TIME 0.34 s

schurStatistics:
    schur calls        : 1
    max. dimension     : 528
    max. maxorder      : 527
    max. int. rows     : 528
    max. matrix memory : 4479552

% schur -sl AU528 > z2
schur: dimension found =  528
schur: max order found =  527
schur: using max order =  527
USING LU DECOMPOSITION

SCHUR TIME 0.72 s

schurStatistics:
    LU calls           : 1
    max. dimension     : 528
    max. maxorder      : 527
    max. int. rows     : 528
    max. matrix memory : 4466880

% diff z1 z2
% wc z1 z2
    528  139656 1954632 z1
    528  139656 1954632 z2
   1056  279312 3909264 total
```

The results for the old test program are:

```
% schur_old -s AU528 > z1

SCHUR TIME 0.41 s

    max. matrix memory : 13436580

% schur_old -sl AU528 > z2

SCHUR TIME 0.72 s

    max. matrix memory : 8952784
```

You can see that the new Schur module is more than 17% faster.
And that significant less memory is used by the new module.
And that too much memory is allocated for Schur by the old module.

4. TWO MORE SIGNIFICANT SCHUR TEST EXAMPLES

```
% schur_old -s AU1231 > z1

SCHUR TIME 41.62 s

schurStatistics:
    schur calls        : 1
    max. dimension     : 6303
    max. maxorder      : 1231
    max. int. rows     : 2298
    max. matrix memory : 72983716

% schur -s AU1231 > z2

SCHUR TIME 36.40 s
    ...
    max. matrix memory : 36481968

% fpdiff2 z1 z2 |& wc -l
613
```

You can see that the new Schur module is more than 12.5% faster. The floating point diff program found 613 printed numbers which were not 100% equal. In most cases is only the least significant digit one less or more. Note that the small output difference can come from the fact that directly the answer 1.0 is used for the diagonal value in place of the multiplication of (1 / sqrt(d)) * (1 / sqrt(d)) * d.

```
% schur_old -s AU2238 > z1

SCHUR TIME 313.32 s

schurStatistics:
    schur calls        : 1
    max. dimension     : 11226
    max. maxorder      : 2238
    max. int. rows     : 4251
    max. matrix memory : 240862700

% schur -s AU2238 > z2

SCHUR TIME 270.65 s
    ...
    max. matrix memory : 120413404
```

You can see that the new Schur module is more than 13.5% faster. Also the memory usage is in the new Schur module less than 50%.
In the last example is also too much memory allocated, because max.int. rows is smaller than 2*maxorder + 1. The difference is 226 rows.
And that gives a total wast of dMax_diff * (16 + oMax * 8) = 4,051,728 bytes.

5. TWO SIGNIFICANT SPACE3D TEST RUNS

When setting parameter "print_time", you get more detailed timing information of some space procedures. Only the most important procedures are listed below. Procedure computeCapacitance is almost equal to the "overall" time information given. Procedure computeCapacitance is the sum of procedures green and schurRowIn and some small overhead. The name schurRowIn is a little bit misleading, because also schurRowOut belongs to it. Note that these values are now printed with two digits behind the floating point, just to be more secure. (This revision was made in "space/auxil/clock.c".)

```
% space3d_old -FC3v pixel_ext -Scap3d.be_window=1.0 -Scap3d.max_be_area=0.1
procedure                     real        user        sys
computeCapacitance         6:38.70     6:28.25       10.16  99.9%
green                      3:02.08     2:56.99        4.57  99.7%
schurRowIn                 3:27.00     3:26.44        0.11  99.8%

overall resource utilization:
    memory allocation  : 62.992 Mbyte
    user time          :      6:28.5
    system time        :        10.2
    real time          :      6:39.0  99.9%
```

```
% space3d -FC3v pixel_ext -Scap3d.be_window=1.0 -Scap3d.max_be_area=0.1
procedure                     real        user        sys
computeCapacitance         6:02.57     5:52.35       10.02  99.9%
green                      3:02.49     2:57.12        5.11  99.9%
schurRowIn                 2:50.13     2:50.21        0.18 100%

    memory allocation  : 37.360 Mbyte
```

I don't give the "overall" time information anymore, see procedure computeCapacitance. The overall speed improvement is more than 36 seconds (is more than 9%). This because of the speed improvement in the Schur module of circa 17.8%.

```
% space3d_old -FC3v pixel_ext -Scap3d.be_window=0.5 -Scap3d.max_be_area=0.1
procedure                     real        user        sys
computeCapacitance         1:54.06     1:50.71        3.27  99.9%
green                      1:29.47     1:28.09        1.71 100%
schurRowIn                   21.43       21.11        0.04  98.7%

    memory allocation  : 18.293 Mbyte
```

```
% space3d -FC3v pixel_ext -Scap3d.be_window=0.5 -Scap3d.max_be_area=0.1
procedure                     real        user        sys
computeCapacitance         1:48.16     1:44.74        3.38 100.0%
green                      1:28.63     1:27.13        1.59 100%
schurRowIn                   16.07       15.96        0.02  99.4%

    memory allocation  : 12.837 Mbyte
```

The overall speed improvement is more than 5.8 seconds (is more than 5%). This because of the speed improvement in the Schur module of circa 25%.

6. THE EVOLUTION OF FUNCTION EXECSCHURROW

The old code of function execSchurRow is given below. But the names of the arrays have been changed. Variable "koffset" is changed into "k2".

```
kk = k - k2; /* execute row k for scIN[kk] */

if (k == 0) scOUT[kk][maxorder] = scDIAG[kk];
else {
    max_i = 0;
    if (kk > 0)
    for (j = 0; j <= scOrder[kk]; j++) { /* A */
        e = scIN[kk][j];
        scV[j][1] = scIN[kk - 1][j + 1];
        if (j == 0) {
            for (i = 1; i <= kk && i <= scOrder[kk - i]; i++) { /* A0 */
                scP[i] = scV[j][i] / e;
                scP1[i] = 1 / sqrt (1 - scP[i] * scP[i]);
                e = sqrt (e * e - scV[j][i] * scV[j][i]);
            }
            max_i = i - 1;
        }
        else {
            for (i = 1; i <= kk && i <= scOrder[kk - i] - j; i++) { /* A1 */
                scV[j - 1][i + 1] = (scV[j][i] - scP[i] * e) * scP1[i];
                e = (e - scP[i] * scV[j][i]) * scP1[i];
            }
        }
    }

    for (j = maxorder + 1 - max_i; j <= maxorder + 1; j++) { /* B */
        L = (j == maxorder + 1)? 1 : 0;
        for (i = Max (1, maxorder + 1 - j); i <= max_i; ++i) { /* B1 */
            if (i == 1) {
                if (j == maxorder) scM[j][i] = 1;
                else if (j == maxorder + 1) scM[j][i] = 0;
            }
            else if (j == maxorder + 1) scM[j][i] = 0;

            scM[j - 1][i + 1] = (scM[j][i] - scP[i] * L) * scP1[i];
            L = (L - scP[i] * scM[j][i]) * scP1[i];
        }
        scOUT[kk][j - 1] = L * scDIAG[kk + j - maxorder - 1];
    }
}
```

You can see that for-loop A is only done when "kk > 0" is true. In that case becomes max_i > 0, because for-loop A0 is always done onces and i is incremented. This is true, because scOrder[kk-1] is always >= 1. For-loop B is always done. It contains a number of if-statements i don't like in the loops (thus i have rewritten that part). When max_i = 0, j = maxorder+1, L = 1. For-loop B1 is not done, because i <= 0 is not true. scOUT[kk][j-1] is set to scDIAG[kk]. Thus, the code can be changed into:

```
if (kk > 0) {
    e = scIN[kk][0]; /* note that this value is always 1 */
    scV[0][1] = scIN[kk - 1][1];
    for (i = 1; i <= kk && i <= scOrder[kk - i]; i++) { ... } /* A0 */
    max_i = i - 1;
    for (j = 1; j <= scOrder[kk]; j++) { /* A */
        e = scIN[kk][j];
        scV[j][1] = scIN[kk - 1][j + 1];
        for (i = 1; i <= kk && i <= scOrder[kk - i] - j; i++) { ... } /* A1 */
    }

    for (j = maxorder + 1 - max_i; j <= maxorder + 1; j++) { ... } /* B */
}
else {
    scOUT[kk][maxorder] = scDIAG[kk];
}
```

Because for-loop A0 is always done onces. The first code part can be rewritten, like:

```
if (kk > 0) {
    scP[1] = L = scV[0][1] = scIN[kk - 1][1];
    e = sqrt (1 - L * L);
    scP1[1] = 1 / e;
    for (i = 2; i <= kk && i <= scOrder[kk - i]; i++) { ... } /* A0 */
    max_i = i - 1;
    ...
}
```

Note that column 0 of scV is not used and that column 1 of scV must contain the previous scIN (scIN[kk-1]). When we look to the second code part with for-loop A, we see that scV[j][1] is filled with scIN[kk-1][j+1].
When we don't want to used scIN[kk-1] anymore, we can possible use scV[][1] instead. And, because scV[j-1][1] is not used in A1, we can write:

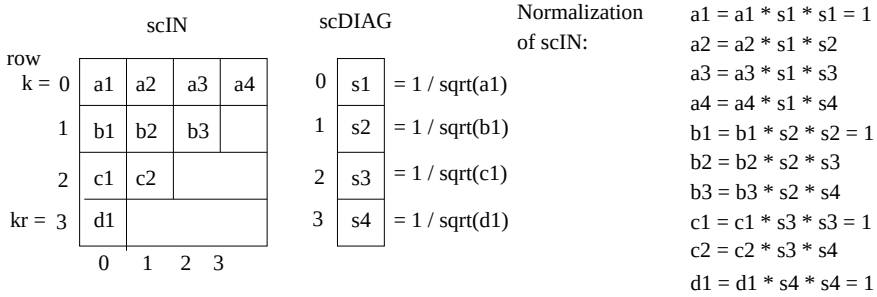
```
    for (j = 1; j <= scOrder[kk]; j++) { /* A */
        scV[j - 1][1] = e = scIN[kk][j];
        for (i = 1; i <= kk && i <= scOrder[kk - i] - j; i++) { ... } /* A1 */
    }
```

Note that also for kk == 0 scV[j-1][1] must be filled with scIN[kk][j] (for j >= 1). Note that, because scIN is first written to scV, we can reuse scIN[kk] for scOUT[kk].
For the inner for-loop A1, for i=1, must scOrder[kk-1] be > j. And because always scOrder[kk] >= scOrder[kk-1] - 1, we can also write:

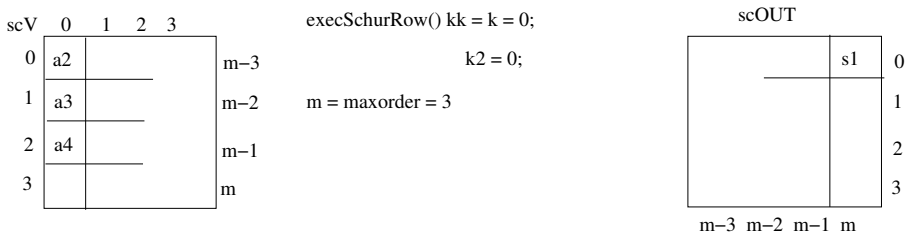
```
    for (j = 1; j < scOrder[kk - 1]; j++) { /* A */
        scV[j - 1][1] = e = scIN[kk][j];
        i = 1;
        do { ... } while (++i <= kk && i <= scOrder[kk - i] - j); /* A1 */
    }
    for (; j <= scOrder[kk]; j++) scV[j - 1][1] = scIN[kk][j];
```

And if we like, we can shift all scV columns one column to the left (this saves memory). The same can be done with scM and i found out that scV can be combined with scM.

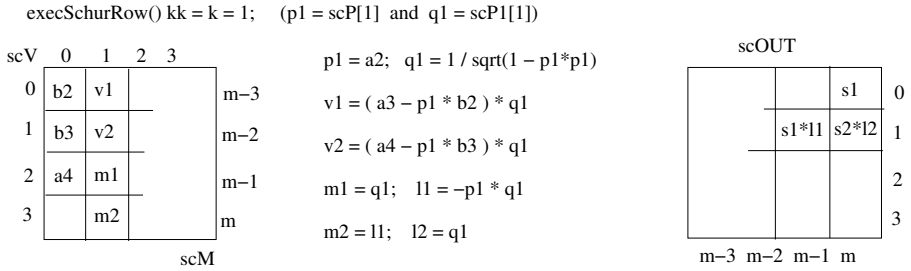
Example of execSchurRow for an input matrix of dimension=4 and maxorder=3:



After 4 input rows is kr=3 and is kr-k >= scOrder[0] and is row 0 executed. But, before the row is executed, it is first normalized. The figure above gives a schematic overview of this normalization step for all rows. You see, that the first value "a1" (the diagonal) becomes 1. You see, that the normalization of row 0 is only possible when inOrder next rows are read-in, because the diagonal values in scDIAG are used for normalization.
After row 0 is executed, you see what happens in the figure below:



Because kk=0 (k=0) only the input is stored in column 0 of scV and the first output value scOUT[m][kk] is equal to scDIAG[kk] (= s1). Row 1 is executed in the figure below:



First p1 and q1 are calculated and then a new column (1) for scV is calculated. Note that the values of column 0 are used before they are overwritten with b2 and b3. Also values for column 1 of scM are calculated. Note that the values v1, v2, m1 and m2 are used in the exec of the next row. The values l1 and l2 are used to set two new values in scOUT[kk].

The figure below gives a schematic overview of the exec of row 2:

execSchurRow() kk = k = 2;

scV	0	1	2	3	
0	c2	v1	v3		m−3
1	b3	v2	m3		m−2
2	a4	m1	m4		m−1
3		m2	m5		m

scM

p1 = b2; q1 = 1 / sqrt(1 − p1*p1)

p2 = v1*q1; q2 = 1 / sqrt(1 − p2*p2)

v1 = (b3 − p1 * c2) * q1
vv = (c2 − p1 * b3) * q1
v3 = (v2 − p2 * vv) * q2

m3 = m1 * q2; l3 = −p2 * m1 * q2
m1 = q1
m4 = (m2 + p1 * p2 * q1) * q2
l4 = (−p1 * q1 − p2 * m2) * q2
m2 = −p1 * q1
m5 = −p2 * l5; l5 = q1 * q2

scOUT

			s1	0
		s1*l1	s2*l2	1
	s1*l3	s2*l4	s3*l5	2
				3
m−3	m−2	m−1	m	

The figure below gives a schematic overview of the exec of row 3:

execSchurRow() kk = k = 3;

scV	0	1	2	3	
0	c2	v1	v3	m6	m−3
1	b3	v2	m3	m7	m−2
2	a4	m1	m4	m8	m−1
3		m2	m5	m9	m

scM

p1 = c2; q1 = 1 / sqrt(1 − p1*p1)

p2 = v1*q1; q2 = 1 / sqrt(1 − p2*p2)

p3 = v3 / sqrt(1 − p1*p1 − v1*v1)

q3 = 1 / sqrt(1 − p3*p3)

m6 = ...; l6 = −p3 * m3 * q3
m3 = ...; m7 = ...;
l7 = (−p2 * m1 * q2 − p3 * m4) * q3
m1 = ...; m4 = ...; m8 = ...;
l8a = (−p1 * q1 − p2 * m2) * q2
l8 = (l8a − p3 * m5) * q3
m2, m5, m9 =...; l9 = q1 * q2 * q3

scOUT

			s1	0
		s1*l1	s2*l2	1
	s1*l3	s2*l4	s3*l5	2
s1*l6	s2*l7	s3*l8	s4*l9	3
m−3	m−2	m−1	m	

Note that the new calculated m1 to m9 values are not more used. But the calculation can not easy be skipped, because the values l6 to l9 must be calculated for setting scOUT[kk]. Now schurRowOut can be done for row 0 (and the other rows), because scOrder rows are executed after row 0 and all needed result values are available to be used.
The figure below gives a schematic overview of how the output buffer needs to be filled for row 0 and row 1:

scOUT

			s1	0
		s1*l1	s2*l2	1
	s1*l3	s2*l4	s3*l5	2
s1*l6	s2*l7	s3*l8	s4*l9	3
m−3	m−2	m−1	m	

Output buffer for row 0:

rowOut[0] = s1*s1 + s1*l1*s1*l1 + s1*l3*s1*l3 + s1*l6*s1*l6

rowOut[1] = s1*l1*s2*l2 + s1*l3*s2*l4 + s1*l6*s2*l7

rowOut[2] = s1*l3*s3*l5 + s1*l6*s3*l8

rowOut[3] = s1*l6*s4*l9

Output buffer for row 1:

rowOut[0] = s2*l2*s2*l2 + s2*l4*s2*l4 + s2*l7*s2*l7

rowOut[1] = s2*l4*s3*l5 + s2*l7*s3*l8

rowOut[2] = s2*l7*s4*l9