TESLA-Report 2001-25

CSC - A System for Coupled S-Parameter Calculations

H.W. Glock, K. Rothemund, U. van Rienen

Institut für Allgemeine Elektrotechnik, Universität Rostock

Abstract:

Large rf-structures are sometimes too complex to be calculated entirely in single simulation runs. If the structure has open ports usually the scattering properties with respect to these ports - the so-called S-parameters - are of primary interest . They can be derived from scattering properties of parts of the entire structure, which may be calculated in separate, less expensive simulations.

The paper describes briefly the underlying theory and introduces a Mathematica™ *application called CSC, that calculates the scattering properties of an object which is a combination of an arbitrary structure of segments with previously calculated S-parameters. For users who are not familiar with* Mathematica*, a short introduction is given. The commands available in the CSC package are explained and an example is illustrated in detail.*

1. Introduction

Our purpose is the determination of scattering properties of complex structures. A structure is called to be complex if the direct S-parameter calculation with a field solving code like MAFIA [1], MicroWaveStudio [2], HFSS [3] etc. exceeds the capabilities of the computing environment in terms of available memory. The method to be described here decomposes the full structure in segments that are calculated separately with the mentioned codes in order to determine the scattering properties of all parts. Each of those individual calculations has to cover structures less complex than the original one and may therefore fit into given restrictions of computing power. The composition to achieve the properties of the original structure is done by coupling the S-parameters of the parts using a dedicated calculation scheme. Therefore we call the procedure Coupled S-Parameter Calculation (CSC).

The primary aim of CSC is to give a possibility to calculate structures that cannot be handled otherwise. So it is accepted to need a longer computation time in total compared with a direct calculation - if the latter one would be possible. Nevertheless there are some occasions that make CSC advantegeous in terms of computation time, too:

- The calculation of the segments may be done in parallel on small machines (which does not reduce the overall effort, but the response time).
- The structure contains segments that can be described analytically (especially waveguides), it has pieces of rotational symmetry which may be calculated in two-dimensional runs, or segments of identical shape appear repeatedly.
- The decomposition avoids large unused mesh volumes or the need to connect structure ports with mesh boundaries by - often bend - waveguides.
- The influence of certain structure modifications, affecting only single parts, is under investigation. Then only the modified segments need to be recalculated. This is of special advantage if there exists an analytical description of the modified parts.

The procedure of coupling the segments needs in either case a very small computational effort. The CSC algorithm consists of some matrix multiplication steps and a single matrix inversion as we are going to show (comp. section 2). The matrices are of low dimension - depending on the number of ports in the system of coupled segments in the order of 10^1. We implemented the method with *Mathematica*™ [4], introducing some conventions how to handle the S-parameter data and how to express the neighbourhood relations of the components.

The underlying theory is described in the following section. A brief introduction in *Mathematica* is given in section 3, which should enable the reader to make use of CSC. In section 4, the CSC commands are described. The usage of CSC is demonstrated in section 5, showing calculations of the coupling properties of combinations of Higher-Order-Mode couplers and beam pipes as they are used in the Tesla Test Facility.

2. Theory

The underlying theory is neither new (compare for instance [5]) nor rather complicated. Though it is extremly versatile and not restricted to any special topology, how the segments are linked, it is (to the knowledge of the authors) not covered in any textbook. (There usually only the very special situation of a linear chain of 2-port-devices is explained, using an adapted representation of the 2x2-scattering matrix.) We use a notation that avoids some formal complication that were made in [5] to distinguish between different kind of objects. (It turns out that there is no need to handle single-port devices, i.e. terminations, in a manner different from that applied to other objects.) This allows us to use more compact formulas and simplifies implementation.

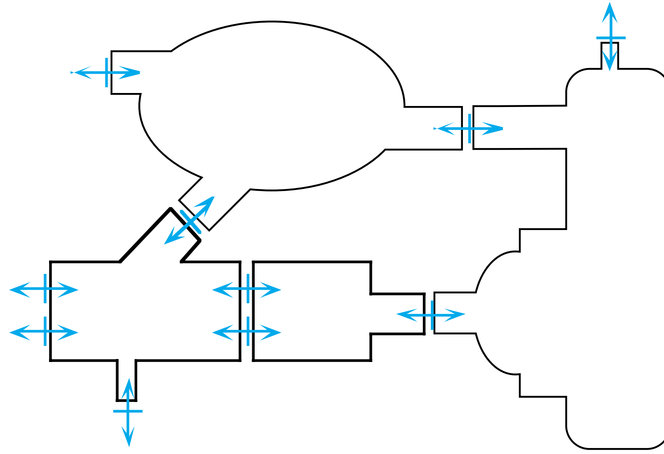

Fig. 1: Example of a complex structure split into four segments. The entire structure has four ports, three of them single moded, one with two modes. For that a 5x5-scattering matrix is searched. Three of the segments have 3x3-scattering matrices, the segment in the lower left corner has a 6x6-matrix. CSC performs the reduction of the characteristics of the individual segments towards the entire system behaviour.

Starting point are the S-matrices of the n individual segments that were calculated previously with a field solving programm in (usually) n different runs:

$$\vec{b}_1 = \underline{\underline{S}}_1 \vec{a}_1 \; ; \ldots ; \; \vec{b}_j = \underline{\underline{S}}_j \vec{a}_j \; ; \ldots ; \; \vec{b}_n = \underline{\underline{S}}_n \vec{a}_n \qquad (1)$$

Herein the vector $\vec{a}_j$ contains all signal amplitudes of the waveguide modes that are incident into object j, in similar convention contains $\vec{b}_j$ the amplitudes of the respective scattered waves. They are correlated by the scattering matrix $\underline{\underline{S}}_j$. With the existence of $\underline{\underline{S}}_j$ a linear behaviour of the object is presumed. The objects need not to be reciprocal, which means that $\underline{\underline{S}}_j$ does not need to be symmetric.

Naturally the scattering properties are frequency dependent. We omit the notation of this dependency assuming that all considerations are done for a given frequency point. If there are several frequencies to be calculated, the procedure has to be applied repeatedly. (The CSC package is prepared to operate on frequency tables of S-matrices.)

The first step is to put the individual S-matrices of all objects together into a block diagonal matrix $\underline{\underline{S}}_{total}$:

$$\begin{pmatrix} \vec{b}_1 \\ \vdots \\ \vec{b}_n \end{pmatrix} = \begin{pmatrix} \underline{\underline{S}}_1 & 0 & 0 \\ 0 & \underline{\underline{S}}_j & 0 \\ 0 & 0 & \underline{\underline{S}}_n \end{pmatrix} \begin{pmatrix} \vec{a}_1 \\ \vdots \\ \vec{a}_n \end{pmatrix} = \underline{\underline{S}}_{total} \begin{pmatrix} \vec{a}_1 \\ \vdots \\ \vec{a}_n \end{pmatrix} \quad (2)$$

Then the amplitudes of all the incident signals are re-ordered in a manner that all signals which come from the outside of the entire system are collected in a signal vector $\vec{I}_{incident}$, whereas those signals though being incident into one segment but coming from a neighbouring one within the system build a vector $\vec{A}_{internal}$:

$$\begin{pmatrix} \vec{a}_1 \\ \vdots \\ \vec{a}_n \end{pmatrix} = \underline{\underline{P}}_{sort->canonical} \begin{pmatrix} \vec{A}_{internal} \\ \vec{I}_{incident} \end{pmatrix} \quad (3)$$

We will refer to the order of signals as it was originally in (2) as "canonical" order whereas the new arrangement will be called "sorted". The transformation from the sorted into the canonical order is mediated by the permutation matrix $\underline{\underline{P}}_{sort->canonical}$. (A permutation matrix is characterized by a single entry "1" in every row and column. Its inverse is found as its transpose.) The order of the signals inside $\vec{I}_{incident}$ and $\vec{A}_{internal}$ need not to be specified as long as it is known and kept consistent throughout the calculation.

Two further definitions are needed: All signals that leave the system at the external ports are put together in a vector $\vec{R}_{scattered}$. The vector $\vec{R}_{scattered}$ has the same length as $\vec{I}_{incident}$ and follows the same indexing convention. That means that the k-th elements of $\vec{I}_{incident}$ and $\vec{R}_{scattered}$ respectively are the pair of incident and scattered wave at a common physical port k.

Then another permutation matrix $\underline{\underline{F}}_{feedback}$ (or shorter $\underline{\underline{F}}$) is needed which expresses the fact that for all internal ports the outgoing signal of one structure segment (a "b"-signal) is the incoming ("a"-) signal of another port. $\underline{\underline{F}}_{feedback}$ contains this permutation map and keeps all "b"-signals, which leave the structure, untouched ("1"-entry on respective diagonal positions). The vector

$$\underline{\underline{F}}_{feedback} \begin{pmatrix} \vec{b}_1 \\ \vdots \\ \vec{b}_n \end{pmatrix}$$

for that contains the feedback relation, but is still in canonical order. Since we are going to eliminate all internal signals we first have to find a correlated expression containing $\vec{A}_{internal}$. This is done using the permutation matrix $\underline{\underline{P}}_{canonical->sort}$, which is inverse to $\underline{\underline{P}}_{sort->canonical}$:

$$\begin{pmatrix} \vec{A}_{internal} \\ \vec{R}_{scattered} \end{pmatrix} = \underline{\underline{P}}_{canonical->sort} \, \underline{\underline{F}}_{feedback} \begin{pmatrix} \vec{b}_1 \\ \vdots \\ \vec{b}_n \end{pmatrix} \quad (4)$$

Now we are able to connect (2), (3) and (4) and find:

$$\begin{pmatrix} \vec{A}_{\text{internal}} \\ \hline \vec{R}_{\text{scattered}} \end{pmatrix} = \underline{\underline{P}}_{\text{can}->\text{sort}} \; \underline{\underline{F}} \; \underline{\underline{S}}_{\text{total}} \; \underline{\underline{P}}_{\text{sort}->\text{can}} \begin{pmatrix} \vec{A}_{\text{internal}} \\ \hline \vec{I}_{\text{incident}} \end{pmatrix} = \begin{pmatrix} \underline{\underline{G}}_{11} & \underline{\underline{G}}_{12} \\ \underline{\underline{G}}_{21} & \underline{\underline{G}}_{22} \end{pmatrix} \begin{pmatrix} \vec{A}_{\text{internal}} \\ \hline \vec{I}_{\text{incident}} \end{pmatrix} \tag{5}$$

In (5) we introduced the matrix $\underline{\underline{G}} = \underline{\underline{P}}_{\text{can}->\text{sort}} \; \underline{\underline{F}} \; \underline{\underline{S}}_{\text{total}} \; \underline{\underline{P}}_{\text{sort}->\text{can}}$ immediately with its internal block structure. If N is the total number of ports and m the number of externally incident (i.e. externally scattered) waves, then the diagonal blocks of $\underline{\underline{G}}$ have the following dimensions: $\underline{\underline{G}}_{22}$: $\{m \times m\}$; $\underline{\underline{G}}_{11}$: $\{(N-m) \times (N-m)\}$, whereas the non-diagonal elements are rectangular: $\underline{\underline{G}}_{12}$: $\{(N-m) \times m\}$; $\underline{\underline{G}}_{21}$: $\{m \times (N-m)\}$.

A more detailed description of the construction of the matrix, illustrated by a step-by-step example, is found in [6].

Some elementary calculations are needed to eliminate the internal signals $\vec{A}_{\text{internal}}$, which finally lead to the relation between the incident and scattered waves at the external ports:

$$\vec{R}_{\text{scattered}} = \left[\underline{\underline{G}}_{21} \left(\underline{\underline{1}} - \underline{\underline{G}}_{11} \right)^{(-1)} \underline{\underline{G}}_{12} + \underline{\underline{G}}_{22} \right] \vec{I}_{\text{incident}} \tag{6}$$

The entire matrix in (6), which is the overall S-matrix of the system, is for that found from

- the S-matrices of the segments (expressed in $\underline{\underline{S}}_{\text{total}}$, which was found from the field solving codes),
- the structure of the system, that defines the feedback relations $\underline{\underline{F}}_{\text{feedback}}$,
- and some arbitrary convention about the signal order, which determines the shape of the permutation matrices (and of course influences $\underline{\underline{F}}_{\text{feedback}}$ too).

So the necessary steps to be performed by the CSC-routine are:

- import the S-matrices of the segments (usually found as frequency dependent tables) and keep them available;
- allow the user to define segments of analytical description (waveguides, rotations, terminations);
- allow the user to define neighbourhood relations;
- test these relations according consistency of numbers of modes, that are propagating in neighbouring ports;
- define a sorted order of signals (according (3));
- calculate the permutation matrix and the feedback matrix;
- calculate the total S-matrix and display it, perform further evaluation and/or export it.

3. A brief introduction in Mathematica

Mathematica is a popular software for any kind of algebraic and numerical calculation. It is now available in its version 4.1; the CSC package is written mainly in version 3, some newer parts in version 4.0. CSC should be compatibel with version 3. *Mathematica* is available for almost any computer platform. Here we give some basic information necessary for a novice user to apply the CSC package and to perform some elementary further steps.

Mathematica is divided into two instances, FrontEnd and Kernel. The FrontEnd is the user interface, used for all input and output operations. It uses worksheet-like documents, which are called notebooks. The Kernel performs all calculations and stores intermediate results during a *Mathematica* session.

The user enters some textual input in the FrontEnd, which is passed to the Kernel and evaluated there, the result is returned to the FrontEnd. These results may be either of textual form, graphics or sounds. The Kernel keeps all the inputs and outputs available during a session. For that a definition may be still valid, even if the user has modified or deleted the according input in the FrontEnd. In spite of this, only the contents of the FrontEnd, i.e. the actual notebook, are stored with a saving operation. The notebooks are based on pure ASCII coding, which makes them transferable between different computer platforms.

During a *Mathematica* session the user may jump to any position in the notebook. Priority of definitions and operations are given by the order of evaluation by the Kernel. Nevertheless one should keep the notebook entries in their logical order, first for the reason of readability and second to allow for automatic evaluation which is done up to down in the notebook.

Mathematica provides a vast amount of commands, covering almost any kind of mathematics. Therefore a powerful help tool is available, accessible both with a menu entry (Hilfe>Help Browser) and a keyboard shortcut (Strg+Shift+f) (modifier keys may differ for different platforms). If a certain command is selected when invoking the help facility, specific help is given. Otherwise a search by category is used.

Mathematica discriminates between uppercase and lowercase characters. The *Mathematica*-own commands are characterized by a leading uppercase letter. You should obey this convention by using lowercase characters for the first letter of own definitions.

Numbers in *Mathematica* are either integer (of infinite length), rational (as ratio of integers), machine precision decimal (usually 16 digits), real (of given arbitrary precision) or complex (as pair of reals with `I` as reserved name for the imaginary unit). Although *Mathematica* is able to import and export numbers in Fortran format, there is internally no 1.23e45 syntax. This number would be defined in *Mathematica* as `1.23*10^45` or `1.23 10^45` (since the standard multiplication sign can be omitted). A complex number reads for instance like `0.321 + I * 1.1 * 10^–3` or

0.321 + I 1.1 10^–3 . The numerical functions are working on complex numbers.

One major concept in *Mathematica* are functions. One extremely simple example is:

```
f[x_] := x^2
```

This defines an object with name f that returns the square of a (during the execution of the function locally used) variable x. Entering f[3] results in 9, f[I] in -1 and f[undefinedObject] in undefinedObject^2.

If you want to make an immediate assignment, you use a syntax like x=2. If you now enters f[x] (be aware of the difference between f[x_] and f[x] !) *Mathematica* will return 4. You may change now the definition of f like:

```
f[x_] := Sin[x^2]
```

Entering now f[3] gives Sin[9]. *Mathematica* leaves this function of an integer argument unevaluated in order to keep it exact. You may enforce further evaluation both by asking for numerical evaluation of the last output denoted by a %-sign: N[%] or by giving a real instead of an integer argument f[3.]. In either case *Mathematica* will return 0.412118. If you need a higher precision than machine accuracy (which usually is higher than the displayed number of digits) you can give the N[]-function a second argument specifying the requested number of digits, e.g N[f[3],1234] .

There are different syntaxes for applying functions. Two more are quite useful, the postfix-notation and implicit functions. Single-argumented functions - e.g the N[]-function in its standard form or our self-defined function f - can be applied on an argument as a postfix using the //-operator. This reads for instance like 3.//f or f[3] // N, or - which is also possible - 3 // f // N. Note, that the //-operator works on the entire preceding expression. Try 1+3 // f . If you are in any doubt about the precedence of operators you should use standard brackets () like in usual mathematics.

Implicit functions are, although they are a bit difficult to understand in the beginning, extremely important, especially if operations are to be performed on several arguments. Multiple values are grouped in lists, denoted by curly brackets {}. A simple list could be

```
{3, -I, Sin[x], Exp[y], 3.}.
```

As indicated in the example, there is virtually no limitation about the members of a list: numbers, functions, lists - a matrix is represented as list of lists - or any other expression may be grouped together with arbitrary depth and structure of nesting. Now we map our funtion f on this sample list, using the following syntax:

```
f[#]& /@ {3, -I, Sin[x], Exp[y], 3.}.
```

This results in the list

{Sin[9], -Sin[1], Sin[Sin[2]^2], Sin[Exp[2 y]], 0.412118}.

Analyzing the command sequence, we first find the implicit function f[#]& itself delimited with the ampersand & and with the hash # as placeholder for an unnamed argument. (If an implicit function does not work as it should, it´s a good habit first to test the presence of the &. Mostly it´s forgotten.) The /@-operator maps the function on the elements of the following list. It is a shortcut for the build-in Map-function which would read in our case as

```
Map[ f[#]&,  {3, -I, Sin[x], Exp[y],3.} ].
```

(Maybe this is a good point to test the online help to find out about the variety of possibilities to apply operators on arguments found in the help section "Functional programming".)

A look on the result of the last operation is also interesting: the first two elements Sin[9], -Sin[1] are like to be expected, but the third result Sin[Sin[2]^2] may be surprising if we look on the third element of the argument list Sin[x]. But we have to remember our assignment x=2, which we made at an earlier point, and which is still valid. A definition is lost only if the session is closed, if it is reassigned (like x=3), if the assignment is deleted (use x=.) or if the symbol x is deleted explicitly (by Clear[x]).

Implicit functions are used in different situations. One operation often needed is the selection of elements of a list that fulfill a certain criterion. If there is for instance a list named freli containing a set of frequencies

```
freli = {2.2 10^9, 2.25 10^9, 2.27 10^9, 2.3 10^9, 2.4 10^9, 2.5 10^9}
```

and we want to extract these elements that belong to a frequency range from 2.25 GHz to 2.3 GHz, we use the *Mathematica* function Select together with a criterion written as an implicit function and resulting in the logical values True or False to construct a new list shortfreli:

```
shortfreli = Select[ freli, (2.25 10^9  <= # <=  2.3 10^9)& ];
```

Again the command line contains several interesting aspects: 1.) It is possible to give two range tests within a single expression. 2.) The logical operators are functions themself. 3.) The semicolon at the end of an expression prevents the result from being printed as output in the notebook. This is particular important if the expressions are extremely long. Then output preparation becomes severly time consuming (and may not be interrupted like other kernel operations - so keep calm if *Mathematica* seems to be not responding). Nevertheless the definition of the result symbol is done independently from the output.

If you are in doubt about "size" and structure of a result, it is a good method first to store it without output (like above) and then to test list length or in case of a multidimensional list its dimension. The corresponding commands are

Length[shortfreli]   resp.   shortfreli // Length

```
Dimensions[shortfreli]    resp.    shortfreli // Dimensions
```

If you need a list with equidistantly distributed elements you can use the Table command, that works similar to usual loop specifications in many programming languages. In order to provide a list going from 170 MHz to 235 MHz in 5 MHz-steps one uses:

```
otherfreli = Table[ fre, {fre, 170. 10^6,  235. 10^6,  5. 10^6} ]
```

There could be any complicated expression depending on fre put in the first argument of Table instead of the iterator fre itself.

Often a part of a list is needed. This is accessed either with the Part command or its shortcut [[ ]]. To get the second element of the list freli, one uses freli[[2]]. If you want to take several parts, you can enter a list in Part: freli[[{2,4}]] returns {2.25 10^9, 2.3 10^9}. Compare the help for Part for further details.

Some numerical functions should be mentioned, needed often when operating with CSC: the absolute value of a complex number z is returned by Abs[z], its argument (given in radians) by Arg[z] and if the dB-scaled value of an S-parameter s11 is searched, you can calculate it using 20 Log[10, Abs[s11]].

The graphical features of *Mathematica* are impressive, but sometimes difficult to access. In order to plot the absolute value of - let´s say - the S23-parameter of a CSC-S-parameter table with name smattab (that generally has the structure {{fre1, S-matrix1}, {fre2, S-matrix2},...} with each S-matrix a n x n-dimensional list) one would use:

```
ListPlot[ { #[[1]],  20 Log[ Abs[ #[[2,2,3]] ]] }& /@ smattab,
    Axes -> False, Frame -> True ]
```

In this sequence, that looks a bit confusing at the first time, an implicit function {#[[1]], 20 Log[Abs[#[[2,2,3]]]]}& was applied on smattab, that extracts the frequency entry (the first part of every element) and the dB-scaling of every corresponding S23 value and puts them together in a two-element-list. Using the maping on the entire S-matrix-list a new frequency table of this individual parameter is constructed, which is displayed by the ListPlot funtion. The following parameters (Axes -> False, Frame -> True) modify the look of the graph. You may change several parameters of the graph using the appropriate options, that are documented in the online help. (Like the typeseting here, a line feed does not affect the command interpretation.)

File import and export is possible in many ways. In either case the first step is to specify the location in the file system where the files are searched for or where exported files are placed. The most easiest way is to define a common starting point. In the current release of CSC (V0.9) this has to be the same path where the CSC package resides. This is done using the SetDirectory["<path>"] command with <path> being an installation specific string. This is one of the very few situations where differences between different computer platforms occur, which differ in the sign used for delimiting directory names (UNIX: " / ", MacOS: " : ", Windows: " \ " or " / ")

and of course in the actual path names. Such a SetDirectory command has to be evaluated in every CSC application notebook before the CSC package is invoked.

Data export and import is very simple. You can export any *Mathematica* expression using the Put command that takes an expression and a filename string as parameters. The export of our previosly defined frequency list into a file named "freqs" being located in the current directory reads:

```
Put[ otherfreli, "freqs" ]
```

or in its short form:

```
otherfreli >>"freqs"
```

Import is as easy using the Get command. Nevertheless there are two important differences. First you have to use an assignment operation (" = ") in order to put the file contents into a specified symbol. If we for instance re-import our frequency list into a new symbol, we have to use one of the following syntaxes:

```
frelinew = << "freqs";    resp.    frelinew = Get[ "freqs" ];
```

The use of the delimiting semicolon is here often quite important, since all contents are printed in the notebook otherwise.

One special form of import is the use of predefined functions (and other *Mathematica* expressions) that are grouped into so-called packages. These packages are used to expand *Mathematica* functionality, both coming with the standard installation and for user-specific applications (like CSC). They have to obey some common rules that allow for a very compact syntax to make them available for *Mathematica*. The command to use the CSC package looks very similar to the file import above:

```
<< "csc`"
```

This causes Mathematica to look in the directory currently set for a folder (indicated by the accent "`") named "csc" that at least has to contain one code file and a file named "init.m". In this init-file all the functions that are provided by the package are declared and their location in the packages´s code files is defined. Usually the user should not be affected by this question. Only in the case of a package modification one should be aware, that it is likely that a change of code files furthermore needs an updated init file.

4.Use of the CSC-package

The CSC package makes a set of functions specific for the S-parameter based chaining of objects available to the user. The following conditions are necessary for a proper function:

1.) *Mathematica* (v 4) and the CSC package folder must be available in the filesystem. (CSC should run with Mathematica v 3, but this is not entirely tested.)

2.) The working directory in the *Mathematica* session is set to the path where the CSC folder is located. (With other words: the CSC folder is a subdirectory of that path.)

3.) The CSC package was loaded in the session (using: << "csc`").

Please note that you may not use or distribute the CSC package without the explicit permission of the package authors.

CSC comes with a own small help facility. In order to learn about it, one should first invoke cscHelp[ ] without an argument between the brackets. It should return

"If you need specific help to any of the following commands, type cscHelp[<command>]"

together with a list of all commands that are currently available. In version 0.9 this list contains the following commands in alphabetical order:

{"cscHelp", "errkop", "extractFreqs", "gesmodnum", "gm", "importMWS", "interpolSmat", "intmodnum", "intpSmat", "kwg", "matvbl", "mfsgm", "nameOfVecs", "numOfVecs", "rotmat", "sgm", "unitarityCheck", "unitarityEnfIntp", "unitarityEnforce", "wgcom", "wgm"}

In the following, a short explanation (partially identical with the online help) of most of these commands will be given, grouped by functional context:

cscHelp returns either the list above, or invoked with a function name as argument the online help to the command (e.g. cscHelp[importMWS]). No ""-signs are used specifying the name.

importMWS[<object>] imports CST-MicroWaveStudio S-parameter data of <object>. It needs the two files object_b.txt and object_p.txt in the current directory. It is important that these files contains data of all S-parameters. MicrowaveStudio exports only the first 20 different parameters in a single file. If one has larger S-matrices one has to export the data of every parameter individually and to merge them together in the object_b.txt (S-parameter values) and object_p.txt (S-parameter arguments in degrees) files. The order of the S-parameters in the files is arbitrary and need not to be the same in both files.

extractFreqs[<S-matrix_table>] returns a list of the frequency points found in a frequency dependent S-matrix list. This is useful if one for instance want to provide a S-matrix list of a connected waveguide using wgcom with the same frequency points like the object loaded.

wgcom[<length>,<frequency>,<listOfCutoffs>] returns the S-matrix of a homogeneous waveguide of <length> with mode cut-off-frequencies given in <listOfCutoffs>. Even if there is only a single mode, its cut-off-frequency must be given in a list.

wgm[<length>,<ListOfWavenumbers>] returns the S-matrix of a homogeneous waveguide of <length> with mode wavenumbers given in <ListOfWavenumbers>. Even if there is only a single mode its wavenumber must be given in a list. This command is useful if one is working with a single frequency point.

kwg[<frequency>,<cut-off-frequency>] returns the according wavenumber in a waveguide.

rotmat[<angle>,<RotationalDegenerationList>] returns the S-matrix of a rotation of length 0 that rotates the frame of reference by <angle>. <RotationalDegenerationList> has an entry for every mode taken into account. Pairs of rotational degeneration have to be indicated using the same number. Monopol modes are indicated using 0. Example: If there is the constellation of a circular waveguide with the first two modes being the two polarizations of the TE11-mode, the third mode being of monopol type and the 4th and 5th mode again being polarizationally degenerated to each other. Then the <RotationalDegenerationList> would read {1,1,0,2,2}.

interpolSmat[<sMatrixTable>,<argumentName>] is intended to return a function of <argumentName> that itself is used to calculate interpolated S-matrices at arbitrary frequencies. It should be called as f[x_] = interpolSmat[<sMatrixTable>,x] (with the name f for instance). f[<frequency>] will give the interpolation of <sMatrixTable> at <frequency>. If one already has a list of frequencies where the interpolation is needed, intpSmat may be used instead.

intpSmat[<S-matrix_list>,<freq_list>] returns a list of elements {freq, S-matrix}, where freq is taken from <freq_list> and S-matrix is the result of a linear interpolation of <S-matrix_list> at freq. If <freq_list> contains frequencies outside the interval covered from <S-matrix_list> a *Mathematica* warning about the use of extrapolation appears and results will be unreliable.

unitarityCheck[<{frequency,S-Matrix}_List>] plots Sum[Abs[Flatten[1 - S S+]]] versus frequency thus providing a unitarity test of the S-matrices. A ideal loss-free object has an unitary S-matrix, so any deviation from 0 is incorrect. unitarityEnforce can be used to enforce unitarity.

unitarityEnforce[<{frequency,S-Matrix}_List>] corrects reciprocity and unitarity deviations of a S-matrix list by enforcing a pure imaginary expression which is similar to the impedance matrix beside the diagonal matrix factor of the waveguide impedances. (Thanks to M.Dohlus for this method.) In order to use unitarityEnforce for a single S-matrix, it has to be put into a list of list together with a frequency like {{ freq,S–matrix }}. The user should be aware that the use of unitarity enforce only makes sense for reciprocal and loss-free objects."

unitarityEnfIntp[<S-matrix_list>,<freq_list>] does the same like intpSmat[<S–matrix_list>,<freq_list>], but enforces unitarity of the S-matrices at every frequency point by scaling.

errkop[<sParameterList>,<couplingList>] checks whether the mode numbers of ports coupled to each other are compatible.The couplings are listed in <couplingList> with entries of the format {{objekt,port},{objekt,port}}. The mode numbers of the ports are found in <sParameterList> for all objects. errkop gives a list with the same length as <couplingList> marking every correct entry in <couplingList> ok, every linkage with wrong mode numbers by **.

gesmodnum[<sParameterList>] calculates the total number of modes in a system <sParameterList>.

numOfVecs[<sParameterList>,<couplingList>] returns the list of {object,port,mode}-specifiers in the same order as used for the result of sgm and mfsgm, resp..

nameOfVecs[<sParameterList>,<couplingList>] returns the list of literal {object,port,mode}-names in the same order as used for the result of sgm and mfsgm, resp..

sgm[<sParameterList>,<couplingList>] returns the S-matrix of the entire system according to its external ports for a single frequency point S-parameter list.

mfsgm[<device_list>,<coupling_list>] is the main calculation command for multiple frequency S-parameter lists. It returns frequency dependent lists of the S-matrix of the entire system according to its external ports. If the component´s S-parameter tables are given for identical frequency sets, no interpolation occurs and the S-matrices of the system are given for this frequencies.
If the frequency sets are different, all frequencies apearing in any of the lists are collected in order to identify those points that are most common. This is done by evaluating the square-sum of the distance of every frequency point to its closest neighbours. Only if this (frequency-normalized) quantity does not exceed a given limit, the frequency point is taken into the final list. In this way the influence of S-parameter interpolation is minimized. Standard setting of the limit is 0.3. It may be overwritten using a third parameter of type Real:
mfsgm[<device_list>,<coupling_list>,<limit>]

Warning: Using the frequency interpolation in mfsgm does NOT ensure S-matrix unitarity. If this is needed, and interpolation cannot be avoided, you should apply unitarityEnfIntp to provide interpolated S-matrices for a given list of frequency points.

5. Commented CSC demo file

In the following a sample CSC application file is demonstrated, as it may look in a *Mathematica* session. (The predefined "Classroom"-style sheet of *Mathematica* was used to achieve the layout.) Not all of the commands provided by the CSC package are used, but it should be helpful for a first start.

A session may start like the following. Since Pi is kept unevaluated until numerical evaluation is enforced, the kernel is started with the definition of a numerical approximation. Furthermore the path is set appropriately.

useful kernel startup and path definition

```
pi = Pi // N
```

```
3.14159
```

```
SetDirectory["Macintosh HD:Dokumente:koppMat:packerl"]
```

```
Macintosh HD:Dokumente:koppMat:packerl
```

Next the CSC-package is loaded and activated:

loading of the CSC package

```
<< "csc`"
```

In order to import MicroWaveStudio S-parameter data from the two files named "HKein_b.txt" and "HKein_p.txt", available in the directory given in the SetDirectory command above, the import filter is used:

import of MicroWaveStudio S-parameter tables using dedicated import filter

```
ssHKcom = importMWS["HKein"];
```

ssHKcom now contains a frequency dependent table of S-parameter data of a DESY-type HOM-coupler calculated with CST-MicroWaveStudio in an appropriate format for further evaluation.

Let's have a look at it:

```
ssHKcom // Dimensions
```

```
{100, 2}
```

```
ssHKcom[[2]]
```

```
{2.50152×10^9, {{-0.270585 + 0.915239 i, 0.0197237 + 0.0995754 i,
   -0.0332092 - 0.185125 i, 0.00254089 + 0.101218 i, -0.00320926 - 0.182762 i},
  {0.0196375 + 0.0991949 i, -0.0653034 + 0.0395959 i, -0.0289278 + 0.0256833 i,
   -0.539209 - 0.830309 i, -0.0468662 - 0.00319502 i},
  {-0.0330551 - 0.184452 i, -0.0289323 + 0.0256873 i, -0.00300034 + 0.010058 i,
   -0.0263297 + 0.0287137 i, -0.483715 - 0.852878 i},
  {0.00251878 + 0.100829 i, -0.539209 - 0.830309 i, -0.0263256 + 0.0287092 i,
   -0.0566833 + 0.0556443 i, -0.0419111 - 0.000863277 i},
  {-0.00317208 - 0.182092 i, -0.0468742 - 0.00319556 i, -0.483715 - 0.852878 i,
   -0.041919 - 0.000870758 i, 0.000797986 + 0.0332454 i}}}
```

Each of the 100 elements of the list carries a frequency and postponed the complex S-matrix of this frequency.

The next steps are needed to define the entire scattering system. First of all, the frequencies found in the HOM coupler S-parameter list are extracted:

```
flig = extractFreqs[ssHKcom];
```

Then a S-matrix table for a two-mode-short is defined for these frequency points, using an implicit function mapped on the frequency list:

```
spSHlig = {#, {{-1, 0}, {0, -1}}} & /@ flig;
```

Furthermore a S-parameter list of a waveguide with a length of 0.15 m and two (degenerated) modes with cut-off frequencies of 2.254 GHz is generated in a similar way.

```
spWGlig = {#, wgcom[0.15, #, {2.254 10^9, 2.254 10^9}]} & /@flig;
```

The next definition is of great importance. All elements building the system are collected in a list:

```
devSWHkSlig =
   {{{2}, spSHlig, {"Short left", {"p_r"}}},
     {{2, 2}, spWGlig, {"Waveguide left", {"p_l", "p_r"}}},
     {{1, 2, 2}, ssHKcom, {"HOM coupler", {"k", "p_l", "p_r"}}},
     {{2}, spSHlig, {"Short right", {"p_l"}}}};
```

The user has to provide the information for each section in the following order:

- a list of mode numbers to be associated with the individual ports (see below)
- the S-parameter frequency list itself (explicitly or usually by its name)
- a list of names with the object´s name as the first element and a list of port names as the second element

The list of mode numbers is needed to define in which way the S-matrix entries correspond with the physical ports. This is done by specifying the number of modes taken into account in each port. For that the length of the list is equal to the number of ports of the section.

For example the HOM coupler has one coaxial port with a TEM mode and two waveguide ports (the beam pipes), each considered here with two polarizationally degenerated TE11 modes. The mode number list of the HOM coupler reads {1,2,2}, the "1" characterizing the coaxial port, the first "2" the number of modes in the left waveguide port, the second one in the right. This correlation of the order of S-matrix entries with physical ports was defined already in the field solving code and **has to be entered correctly by the user in CSC**. There is virtually no mean for the CSC algorithm to check for a correct correspondence, which makes that point inevitable for a proper result. The naming of ports is done in the same order as they appear in the mode number list.

Now all the sections are available, but the definition of neighbourhood relations is still missing. This is done by the following definition of a list:

```
kopSWHkSlig = {{{1, 1}, {2, 1}},
                     {{2, 2}, {3, 2}},
                     {{3, 3}, {4, 1}}};
```

This list specifies coupling of objects by entries of the shape {{object_1,port_1}, {object_2,port_2}}. The example above has to be interpreted as follows: The first (and only) port "p_r" of the first element of the section´s list, "Short left", is coupled with the first port ("p_l") of the second section ("Waveguide left"). The second port of the second section ("p_r" of "Waveguide left") is connected with the second port of the third section ("p_l" of "HOM coupler") and the third one of section no. 3 with the first of the fourth section ("p_r" of "HOM coupler" with "p_l" of "short right").

This format allows to describe any system topology one may think off, since it uses the most elementary correlation between the sections, their neighbourhoods. One may check plausibility of the coupling list, tested by looking for identical mode numbers in ports that were specified as neighbours, using the errkop command. The entire systems has only those open ports that do not appear in the coupling list. Our purpose is to calculate the systems S-matrix according to these open ports.

This is done by:

```
dht = mfsgm[devSWHkSlig, kopSWHkSlig];
```

using the mfsgm command with the device list and the coupling list as arguments. The result, which is stored here in a list named dht, obeys exactly the same format convention like the S-parameter tables of the sections, so it may be used in further chaining steps. Since in the example above there remains only a single port open, the total scattering "matrix" is of dimension 1 x 1. It is listed together with the according frequency:

```
dht
```

```
{{2.5 × 10^9, {{-0.188953 + 0.981909 i}}},
 {2.50152 × 10^9, {{-0.182955 + 0.983051 i}}},
 {2.50303 × 10^9, {{-0.176696 + 0.984222 i}}},
 {2.50455 × 10^9, {{-0.170503 + 0.985327 i}}},
 {2.50606 × 10^9, {{-0.164032 + 0.98645 i}}},
 {2.50758 × 10^9, {{-0.157446 + 0.98755 i}}},
 {2.50909 × 10^9, {{-0.15091 + 0.988559 i}}},
```

(altogether 100 entries).

A modification of the system is done very easily:

right waveguide port now remains open ...

```
devSWHkSli2 = {{{2}, spSHlig, {"Short left", {"p_r"}}},
                    {{2, 2}, spWGlig,
    {"Waveguide left", {"p_l", "p_r"}}},
                    {{1, 2, 2}, ssHKcom,
    {"HOM coupler", {"k", "p_l", "p_r"}}}};
```

```
kopSWHkSli2 = {{{1, 1}, {2, 1}},
                   {{2, 2}, {3, 2}}};
```

```
dht2 = mfsgm[devSWHkSli2, kopSWHkSli2];
```

```
dht2[[23]]
```

$\{2.53333\times 10^{9}$,
$\{\{-0.630998 - 0.671187\,i, -0.11078 + 0.0722936\,i, -0.0839508 - 0.354904\,i\}$,
$\{-0.110779 + 0.0722796\,i, -0.804828 - 0.54459\,i, 0.0486772 + 0.188344\,i\}$,
$\{-0.084205 - 0.356047\,i, 0.0488394 + 0.188958\,i, -0.193179 + 0.889521\,i\}\}\}$

Now two ports with in total three modes remain open, so the result are 3 x 3 matrices, shown here for a single frequency.

For further evaluations or graphical representations one may use *Mathematica* commands as described in section 3.

References

[1] MAFIA V. 4.024, CST, D-64289 Darmstadt
[2] MicroWaveStudio, CST, D-64289 Darmstadt
[3] HFSS, Agilent (formerly Hewlett Packard), Palo Alto (CA), USA
[4] *Mathematica*™ , Version 4, Wolfram Research, Champaign (IL), USA
[5] Th.-A. Abele: "Über die Streumatrix allgemein zusammengeschalteter Mehrpole", AEÜ, Band 14(6), S. 262 - 268, 1960
[6] K.Rothemund, H.-W. Glock, M. Borecky, U. van Rienen: "Eigenmode Calculation in Long and Complex RF-Structures Using the Coupled S-Parameter Calculation Technique", ICAP 2000, Darmstadt, and TESLA–Report 2000-33, DESY, Hamburg