SPACE APPLICATION NOTE ABOUT IMPLEMENTING A GREEN API

S. de Graaf

Circuits and Systems Group
Faculty of Electrical Engineering
Delft University of Technology
The Netherlands

Report ET-CAS 02-02
July 19, 2002

Last revision: August 11, 2004.

1. INTRODUCTION

At Monday, 15 Apr 2002, i started the work for an implementation of a *Green API* (Application Programming Interface) for connecting a customer-specific Green's functions module. Which module can be used for 3D capacitance and/or 3D substrate resistance calculation in the *space3d* program.

1.1 Basic Idea

The basic interface will be a pair of boundary elements as input, expecting a Green's function value as return. Possible, something must be done to the *space3d* initialization interface for getting the technology data from the ".t" file. Also then a larger number of layers can be supported.

1.2 Result

We distribute all sources of the Green Module. Thus, a licensed user can modify it to its own wishes and create its own "green.a" library file. The make procedure is changed. The other parts are distributed in one big object module "Xspace.o". Thus you can make your own version of *Xspace (space3d)*. The library "auxil.a" is also distributed with all its source files. The required include files of SPACE are also distributed.

1.3 Working Environment

A new SPACE tree was setup in directory:

```
/u/25/25/space2/Shadow/hp700.test/src/space
```

The source files have symbolic links to directory:

```
/u/25/25/space2/Shadow/general/src/space
```

A symbolic link is removed to ".old", when the source file is changed.
To compile the *Xspace (space3d)* program, do:

```
% cd /u/25/25/space2/Shadow/hp700.test
% make space6
```

The compilation result (executable) can be found in directory:

```
/u/25/25/space2/Shadow/hp700.test/src/space/space
```

Tests are done in project directory:

```
/u/25/25/space2/Shadow/hp700.test/src/space/sub3term/projectname
```

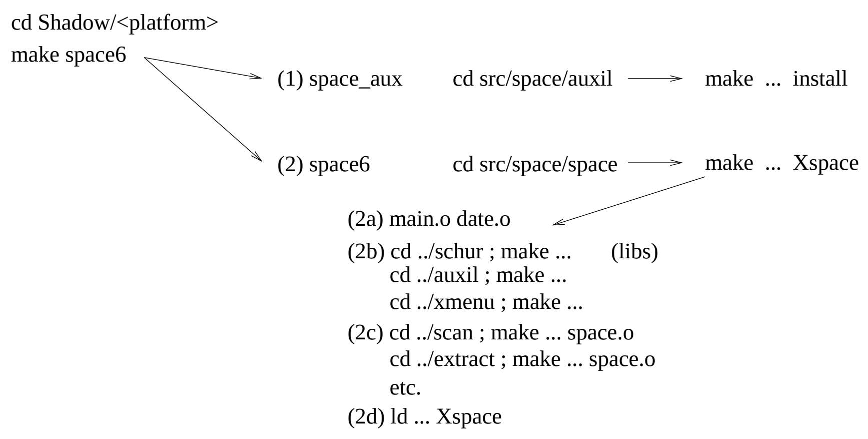
1.4 Old Xspace Loading Modules

Modules for building Xspace (space3d)	
Module	Notes
../scan/space.o	Layout scan functions
../extract/space.o	Extract functions
../lump/space.o	Lump functions
../bipolar/space.o	Bipolar functions
../substr/space.o	Substrate functions
../spider/space.o	Spider functions
../green/space.o	Green's functions
../X11/space.o	Xspace functions
../schur/schur.a	Schur functions
../auxil/auxil.a	other functions
../xmenu/libmenu.a	Xspace functions
main.o (*)	Dummy main module
date.o (*)	Version/date function
$ICDLIB/libcirfmt.a	Circuit DB access lib
$ICDLIB/liblayfmt.a	Layout DB access lib
$ICDLIB/libddm.a	Database function lib
$ESELIB/libese.a	other useful functions
-lXaw -lXmu -lXt -lX11 -lXext	X Window System libs
-lm -lbsdipc -lBSD -ldld	Other libs

(*) Place of loading is "src/space/space".

Note that "green/space.o" is replaced by the "green/green.a" library.

1.5 Old Xspace Make Procedure



```
cd Shadow/<platform>
make space6
        (1) space_aux     cd src/space/auxil  ---->  make ... install

        (2) space6        cd src/space/space  ---->  make ... Xspace

              (2a) main.o date.o
              (2b) cd ../schur ; make ...      (libs)
                   cd ../auxil ; make ...
                   cd ../xmenu ; make ...
              (2c) cd ../scan ; make ... space.o
                   cd ../extract ; make ... space.o
                   etc.
              (2d) ld ... Xspace
```

1.6 New Xspace Make Procedure

To make *Xspace (space3d)* the following steps need to be done:

```
1) make of "Xspace.o" (object file)
2) make of "green.a" (green module)
3) make (load) of Xspace (executable)
```

Object module "Xspace.o" contains everything except the green module. Note that this module is delivered to parties how want to make a new *Green API*. The "Xspace.o" object file is made with the following command:

```
ld $LDRFLAGS <object-modules> <object-libraries> -o Xspace.o
```

The new *Xspace* make procedure is now first making the "Xspace.o" file. Thus, the used "CONFIG/M.<architecture>" file must now contain the new LDRFLAGS variable. This LDRFLAGS contains the direct 'ld' flags and must at least contain the loader **-r** option.

The *Green API* can be compiled in directory "src/space/green" by typing the *make* command. But can also automatically be performed by the *Xspace make* command.

The executable *Xspace* must be made in directory "src/space/space". Type *make* without any arguments to make *Xspace* with an existing "Xspace.o" object. If "green.a" is out of date, it is also automatically updated. Thus, the following load command is done:

```
c++ $LDFLAGS Xspace.o ../green/green.a ../auxil/auxil.a -ldld -o Xspace
```

The loading phase must be done with the *c++* compiler, because "green.a" contains at this moment a *c++* object. Note that the "auxil.a" library must be specified, because the green module must load (maybe) some parts from the "auxil.a" library. On HP-UX also the shared management functions (**-l***dld*) must still be loaded.

On HP-UX the following file *(ls -l)* sizes has been seen: (Xspace.o = 2,121,408) and (Xspace = 2,723,656).

The following files are needed to make *Xspace* with a *Green API*:

Directory:	Files:
.	Config.mk, M.architecture
auxil	auxil.a, *.h
extract	export.h
green	Makefile, green.a, *.h, *.c/C
include	*.h
scan	export.h
space	Makefile, Xspace.o
spider	define.h, export.h

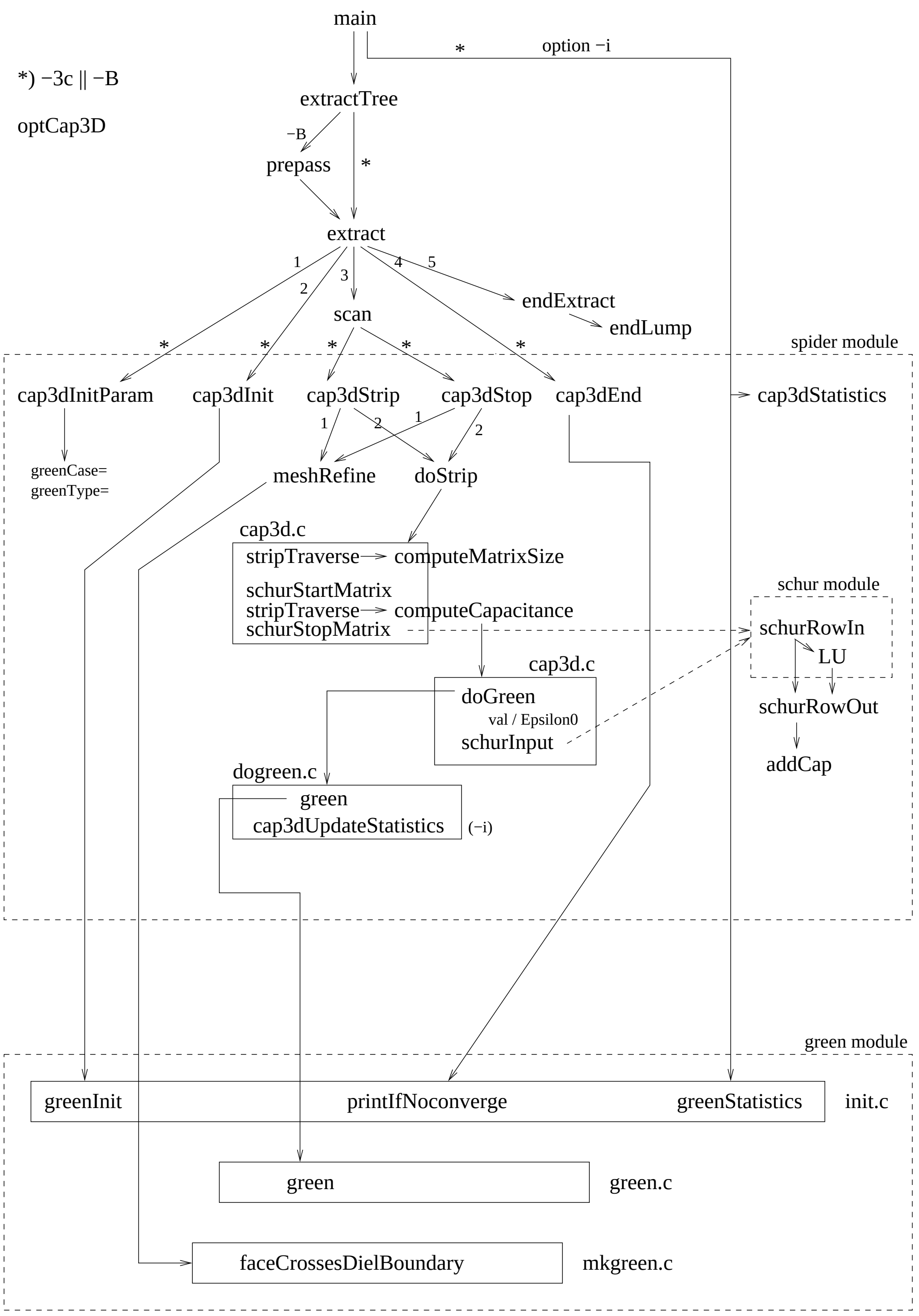
Note, that "CACDCONFIG=../M.architecture" is added to the Makefiles.
Note that all "auxil.a" sources are distributed.
Note that library "green.a" can also be recompiled.

1.7 List of Changed Files

The following list of new SCCS delta’s is made on July 10 - 19, 2002:

Directory:	File:	Delta:
scan	edge.c	4.11
scan	info.c	4.10
scan	sp_main.c	4.29
scan	tile.c	4.22
extract	nodepnt.c	4.10
lump	init.c	4.54
lump	node.c	4.64
schur	schur.c	4.6
spider	extern.h	4.12
spider	recog.h	4.5
spider	cap3d.c	4.35
spider	convex.c	4.13
spider	displace.c	4.3
spider	dogreen.c	4.5
spider	mesh.c	4.18
spider	pqueue.c	4.4
spider	recogm.c	4.15 - 4.16
spider	refine.c	4.16
spider	sppair.c	4.24
spider	strip.c	4.22
green	Makefile	4.11 - 4.12
green	export.h	4.5
green	extern.h	4.12
green	gputil.h	4.7
green	green.h	4.9
green	images.h	4.3
green	analytic.c	4.9
green	colloc.c	4.15
green	init.c	4.21 - 4.22
green	intadptv.C	4.4 - 4.5
green	integrat.c	4.9
green	galerkin.c	4.12 - 4.13
green	gputil.c	4.7
green	green.c	4.12
green	mkgreen.c	4.17
green	mpgreen.c	4.10
space	Makefile	4.38 - 4.40

2. The Main Flow



Before we can say something about the green module flow, we must first say something about the space main flow. The green module is called by the spider module of *space*.

The spider module is called by the *space* program when 3D capacitance extraction needs to be done (options: **-3c** or **-3C**). But the spider module is also called by accurate substrate resistance extraction (option: **-B**) in prepass 1* of the *space* extraction. Thus, both options can be used at the same time. The results of prepass 1 are stored in a temporary file and read in the last extraction pass.

Main calls function extractTree for each cell which must be extracted. Function extractTree does the needed extraction passes, always is function extract called. Function extract inits the extraction pass, does a complete layout scan and ends the extraction pass.

In function scan, a horizontal and vertical layout scan splits the layout up in small parts (tiles). The tiles of conductors (and contacts) are converted in a spider mesh. The capacitances (or resistances) in this spider network are computed and converted in a capacitance (and/or resistance) network. The calculation is done in one Schur matrix inversion.

Note that bigger layouts are divided in strips. And that the above step is done for each layout strip. And that the results of the strips are connected with each other. You can see in the flow chart, that function cap3dStrip is called when a layout strip is ready. Function cap3dStop is called when the last layout strip is ready.

Note that like tiles, the outer surface of each conductor (and contact) in the mesh is represented with a face. There are horizontal faces (flat 2D faces, corner points have all the same z-coordinate) and there are vertical faces. The faces of contacts are always vertical faces. The thickness of the conductors (vdimension specification) is responsable for the vertical conductor faces. Conductors with a thickness of 0 (sheet conductors) do not have vertical faces. Function meshRefine finishes the refinement of all faces. Vertical faces are split when they cross a dielectrical boundary. By piecewise linear (pwl) all faces are converted to triangular faces. At last, in meshRefine the faces get all a center spider. By piecewise constant (pwc) mode these center spiders are used for the network calculation (by pwl only the corner spiders are used).

Via function doStrip is stripTraverse called, which calls computeCapacitance. Function doGreen is called, which calls function green (of the Green module). Every computed green value is put in the Schur matrix on some position (function schurInput). Note that by capacitance calculation each value is divided by Epsilon0.

* In the new version of *space3d*, the program calls another program, called *makesubres*, to calculate the substrate resistances.

3. The Green Module Interface

The Green module interface starts with a call to function greenInit. This function has one argument. This argument specifies the unit of the spider coordinates in meters. The function returns a greenFactor, which is the input argument multiplied by 4*PI. The function scales the technology boundaries, which are specified in microns, thus they can be compared with the spider layout coordinates.

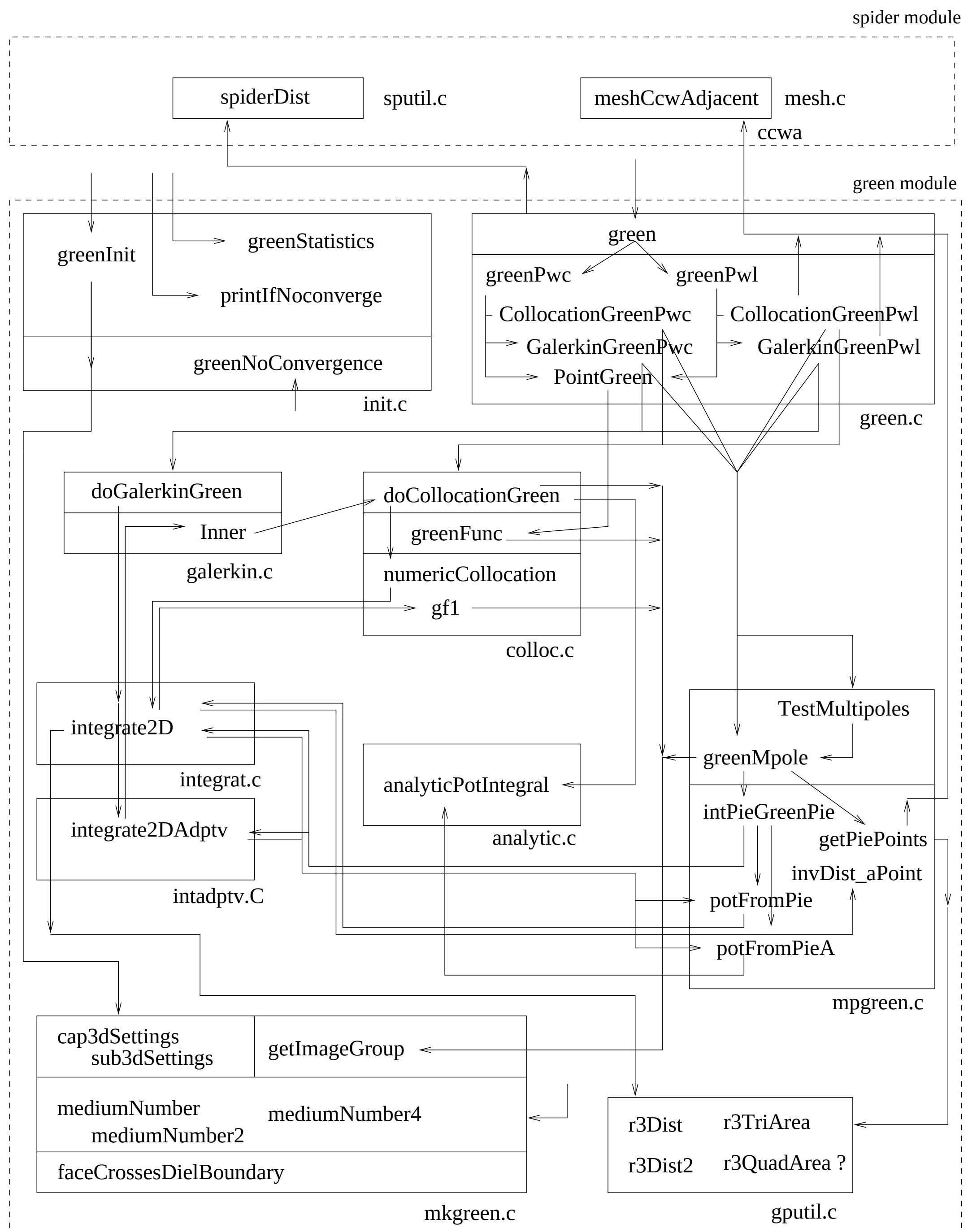
The Green module can be used for two green cases, for cap3d and sub3d extraction. The global variable greenCase specifies which case is used (DIEL=0 or SUBS=1). For greenCase DIEL, the technology globals diel_cnt and diels[] (datatype dielectric_t) are read. For greenCase SUBS, the technology globals substr_cnt and substrs[] (datatype substrate_t) are read. Also, depending on greenCase, extraction parameters "cap3d.xxx" or "sub3d.xxx" are used (see appendix A). Note that the Green module can only once be initialized for both green cases. Note that diel_cnt must be >= 0 and <= 3. Note that substr_cnt must be > 0 and <= 2 (3 for "metalization"). The global variable subcontZposition (default 0) is set by "metalization" and must be used in the spider module to set the act_z position of all substrate spiders.

The Green module can use four different boundary element calculation methods (piecewise constant or linear and collocation or Galerkin). Parameter "cap3d.be_mode" or "sub3d.be_mode" specifies the mode used. The default mode is "0c", piecewise constant collocation. Besides that, default the multipole method is used (parameter "use_multipoles"). Note that function greenMpole allocates multipole moments, which are remembered by spider → moments. The spider module must free these allocated moments after usage.

The Green module is called by function green, which has two spiders as argument. Function green returns the green value for these two spiders. The Green module uses the following spider datatypes: spider_t, spiderEdge_t and face_t. Datatype spiderEdge_t is only used by piecewise linear (pwl). The Green module uses internally datatype pointR3_t for the spider coordinates. The Green module uses only the act_x, act_y and act_z spider coordinates. It uses spider → face by the piecewise constant (pwc) case and spider → edge by the pwl case. By pwc, the spider must be the center spider of a face. By pwl, the spider must be a face corner spider (spider → face must be null). Member face → area is used in both cases, face → len is used only in the pwc case. Note that member edge → face is used by pwl to find the face. Note that member face → corners[] is used to find the corner spiders. Note that face → corners[3] must be null for triangular faces.
The pwl case uses spider macro's NEXT_EDGE and ccwa. Note that both cases use function spiderDist (from spider/sputil.c) to calculate the distance between two spiders.

Function faceCrossesDielBoundary tests for the cap3d greenCase of the face crosses the dielectric boundary. This function returns TRUE, when a crossing boundary is found. In that case global variables crossingIndex and crossingRatio are set.

4. The Green Flow



In the green module flow the functions are shown which call each other. Private (internal) functions of each module are in most cases not shown. The green module calls two functions out of the spider module (meshCcwAdjacent via macro ccwa and spiderDist). The green module calls also a number of functions from the "auxil.a" *space* library. There are also functions, which are only for test purposes (like TestMultipoles).

Note that some parts of the green module have a DRIVER. That part can separate be compiled and tests some functionality of that part.

Note that with CAP3D defined, everything is compiled with DEBUG defined. See the "config.h" file of *space*. Note that you can activate Debug output by placing the name of the module in the ":debug" file. Note that ASSERT statements always are tested. If an ASSERT fails, a message is given by the *space* program. Almost all ASSERT failures leads to incorrect *space* result. By some failures the space program stops the execution by calling the die() function. Else the execution can be stopped by the signal handler of *space*.

The green module uses global variables (for example: greenCase and greenType) which are set in the spider module. But on the other hand the spider module uses also global variables (for example: FeModePwl) which are set by greenInit. Thus care must be taken by replacement of functionality.

Note that one module is written in C++ (intadptv.C). This function is only used when some (undocumented) parameters are put on. Because of this C++ module, the modules must be loaded with the *c++* compiler. Note that the green module contains more undocumented space parameters.

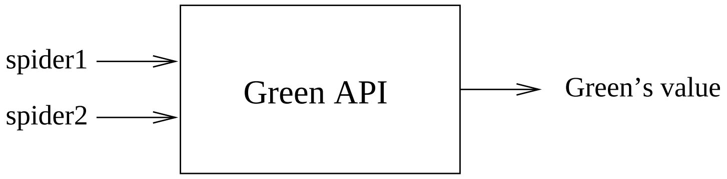
As can be seen in the control flow, function green calls greenPwc or greenPwl, depended of parameter "be_mode" (value of variable FeModePwl). Depending on mode function CollocationGreen or GalerkinGreen is called. Default both functions call greenMpole (which speed-up the calculations). Thus the "mpgreen.c" part is important. Note that function PointGreen is called for very large spider distances.

The "mkgreen.c" part is also very important. This part is inited by greenInit by calling cap3dSettings or sub3dSettings. This part contains the technology dielectric information and can deliver the requested green term with function getImageGroup. Note that a green term consists out of a term group. The number of dielectric layers used is very important.

4.1 Parameters

The Green module uses a large number of parameters. The *space* parameters are read by function main from the command line and the *space* parameter file. Note that the technology data file is also read ones by main. Note that the DRIVER test programs do not read these files. The value of a parameter can be retrieved with a paramLookup function. With paramLookup a default value can be specified, for the case that the parameter is not set. There are 4 types of parameter values (bool_t, int, double, char *). The bool_t values can be "on" and "off".

5. The Green API



The *Green API* calculates a Green's value based on positions of the spiders laying somewhere in the silicon wafer material or the air. One spider is the observation point and the other the charge point. The Green's value is a measure of the influance of the charge point on the observation point. This influance is depended of the distance between both points and the different materials around the points. The spiders are the center points of a face (triangular or rectangular shape) by piecewise constant or the corner points by piecewise linear. **The dimension of the face is also important**. The influance is measured in the corner points of the observation face. All spider values are put in a Schur matrix. The inverted matrix gives a network with capacitance or substrate resistance values. The size of the matrix and the number of Green's values which must be calculated is depended of the number of spiders, see table:

spiders	calculations	matrix size
1	1	1 x 1
2	4	2 x 2
3	9	3 x 3
10	100	10 x 10
50	2500	50 x 50

The number of spiders can be reduced by parameters "be_window" and "max_be_area". The calculation of each Green's value cost also some time and is depended of the method (Green's function) used. *Space3d* is using a Green's function, which consist out of a limited number of Green's summation terms. The Green's value calculation (summation) is stopped after some convergence accuracy is reached. Default after Abs(term/value) < 0.001 (parameter "green_eps"). Each term consists out of an image group. Note that for 3 material layers one image group can contain more than 100 images (sub terms). How higher the group number, how more images! Thus it is important to stop as soon as possible. The number of image groups can be controlled with parameter "max_green_terms".
The following source functions are calling each other:

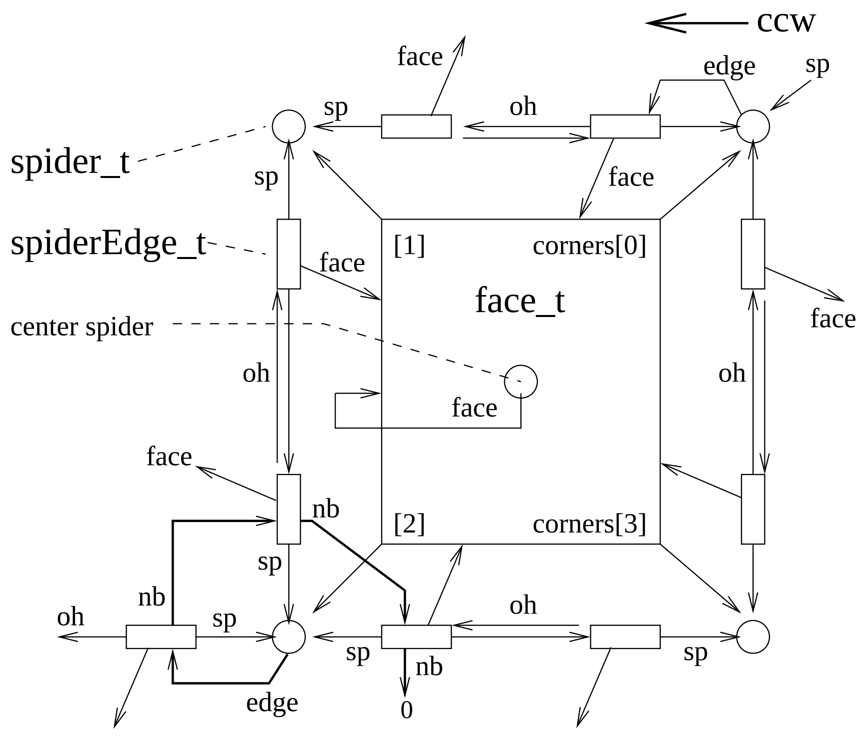
```
double green (sp1, sp2) spider_t * sp1, * sp2;
{
    return FeModePwl ? greenPwl (sp1, sp2) : greenPwc (sp1, sp2);
}

Note: Default "be_mode" is "collocation" (FeModeGalerkin = FeModePwl = 0).

green_t greenPwc (sp1, sp2)
{
    r = sp1 -> face -> len + sp2 -> face -> len;
    d = spiderDist (sp1, sp2);

    if (d >= r * pointGreenDist && d > 0) return PointGreen (sp1, sp2);
    if (d >= r * collocationGreenDist) {
        val = CollocationGreenPwc (sp1, sp2);
        if (d < r * asymCollocationGreenDist && sp1 != sp2)
            return (val + CollocationGreenPwc (sp2, sp1)) / 2;
        return val;
    }
    return GalerkinGreenPwc (sp1, sp2);
}

Note: Default pointGreenDist="infinity" and collocationGreenDist=0.
      Default asymCollocationGreenDist="infinity".

green_t CollocationGreenPwc (spo, spc)
{
    if (useMultiPoles && greenMpole (0, spo, spc, &val)) return val;
    ...
    return doCollocationGreen (opp, cp1, cp2, cp3, cp4) / area;
}

Note: Default useMultiPoles=1 (parameter use_multipoles=on).

bool_t greenMpole (typeOfShapeFunction, spo, spc, val)
int typeOfShapeFunction;
spider_t *spo, *spc; green_t  *val;
{
    if (!getPiePoints (spc, &cCenter, cPeriPoints, &cNrpp, &cMedium, &cRadius))
        return (FALSE);

    if (!spc -> moments)
        makeTotalMoments (cMom, &cCenter, cPeriPoints, cNrpp, ...);

    R3Set (oCenter, spo);
    oMedium = mediumNumber2 (oCenter.z, oCenter.z);
    ...

    value = 0;
    for (g = 0; g < maxGreenTerms; g++) {
        grp = getImageGroup (cMedium, oMedium, g);
        if (grp.size == 0) { greenNoConvergence (cMedium, oMedium); break; }

        term = 0;
        for (i = 0; i < grp.size; i++) {
            R.z = oCenter.z + ZQ_SIGN * cCenter.z + DISTANCE;
            Rabs = R3Norm (R);
            distratio = Rabs / convergenceRadius;
            if (distratio > multipolesMindist && distratio > 1) {
                /* use multipole expansion */
                gterm = STRENGTH * mpConnect (.., R, Rabs);
            }
            else { /* use ‘traditional’ integration */
                ...
                gterm = STRENGTH * intPieGreenPie (...);
            }
            term += gterm;
        }
        value += term;

        if (Abs(term/value) < collocationEps) goto ret;
    }

    if (nLayers[greenCase] > 1 && g == maxGreenTerms)
        greenNoConvergence (cMedium, oMedium);
ret:
    *val = value;
    return (TRUE);
}

Note: ZQ_SIGN, DISTANCE and STRENGTH are from grp.images[i].
```

Note that function getImageGroup() gets a new image group. The group number g is 0 to maxGreenTerms.

Note that the *Green API* is receiving two spiders. But most of the Green module is not working with spiders, but with 3D points. For the "pointR3_t" typedef see file "green.h". Thus the actual spider coordinates are put in a 3D point with macro R3Set. Macro R3Set uses macro R3Assign (from "gputil.h") to set the point coordinates. Note that the spider y-coordinate is multiplied with y_stretch (value is default 1). Function greenMpole (and TestMultipoles) is also receiving the two spiders. They are used by function getPiePoints to set the perimeter points. Also the spider→moments pointer is set when function makeTotalMoments is done. Thus, this is done only ones (see file "mpgreen.c").

Note that datatype pointR3_t is also used in the spider module in file "convex.c". That file is also using the point macro’s from file "gputil.h".

5.1 The spider face connections



In the above figure you can see that two spiders are connected with each other by a pair of spiderEdge objects. Each spiderEdge can also point to a face. The face has corners which points to the corner spiders and it contains ’len’ and ’area’ information. The face is a triangle when corners[3] is zero. With the NEXT_EDGE macro, the next edge of a spider can be found (is edge$\rightarrow$nb). With the ccwa macro, the counter clockwise adjacent spider of a face can be found (via the other-half pointer ’oh’). Note that there are extra spiders added to the mesh, which are center spiders. Only center spiders point directly to a face.

5.2 Green Statistics

Green statistics (use option **-i**) prints at the end of the space program the number of invocations of the functions PointGreen, CollocationGreen, GalerkinGreen, getImageGroup and the total number of images get. The variables used are not inited in greenInit. Thus, the statistics are the overall end result. Use option **-B** without options **-3C** to get statistics results only for substrate resistance extraction.

Note that the cap3d statistics are initialized by each pass which call function cap3dInit. Thus, the cap3d statistics results are only for the last extracted cell and only for the **-3C** pass when both **-B** and **-3C** options are used.

Note that the Schur statistics only ones are initialized by schurStartMatrix (initSchur).

5.3 Green Modes

Variable	cap3d (-3C)	sub3d (-B)
greenCase	0 (DIEL)	1 (SUBS)
greenType	0	2 (!metalization) 3 (metalization)

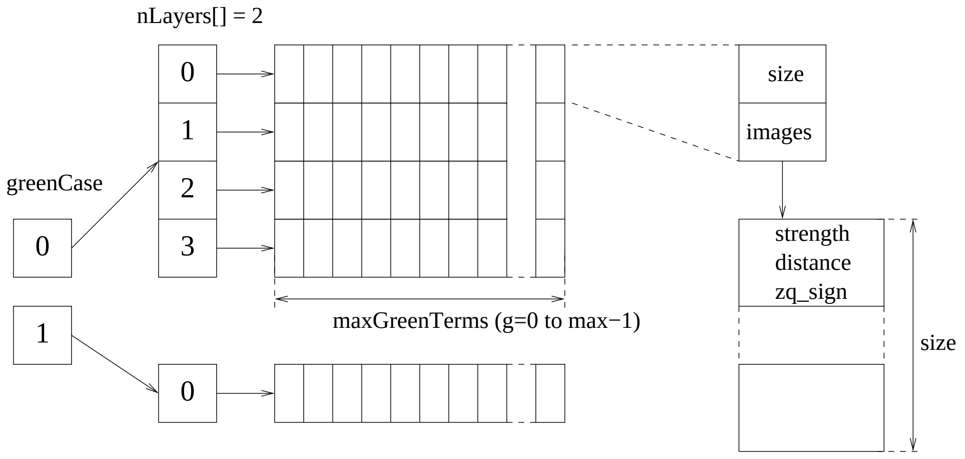
Note that for the substrate case a "metalization" sublayer can be specified in the technology file. This must be the last sublayer and is intended for a ‘‘grounded’’ substrate technology. In that case the cap3d green function calculation method is used.

be_mode	Collocation	Galerkin
Pwc	0c / collocation	0g / galerkin
Pwl	1c	1g

The default boundary element mode is "0c" (piecewise constant collocation).

6. A Simple Green API

In a simple *Green API*, we only need to replace function getImageGroup(). Thus that the user can deliver his/her own image groups. Or can experiment with other greenSeries(). Or can extend the existing set of greenSeries() with new ones for more than 3 layers.



In the above figure we see the amount of memory that is needed for greenCase=0 (diel) and greenCase=1 (subs), to store references to images. The greenCase=1 uses always only one type of images (gs2211 or gs3222, for nLayers=3 only gs3322 or gs3333).
But for greenCase=0 is used nLayersˆ2 images. For nLayers=2 this are 4 images (gs0211, gs0221, gs0212 and gs0222). For nLayers=3 this are 9 images (gs0311, gs0321, gs0331, gs0312, gs0322, gs0332, gs0313, gs0323 and gs0333).
Thus for nLayers=4 you need 4ˆ2 = 16 images.

```
greenValue = directTermsValue(g=0) + SUM [greenTermsValue(g=1 to max-1)]
```

From above Green’s function formule you see, that the greenValue is the sum of the directTerms and the other greenTerms. Each greenTerm consists out one of more images. The directTerms consists normally out 2 images.

6.1 Notes

The image groups are hold in memory during a *space3d* extraction. Thus, they can be reused for the extraction of other cells. This happens also by *Xspace*. As a consequence, these data must stay valid during all extractions. The data is hold only for one greenType. The greenType for substrate resistance extraction is based on the technology data (normal or grounded).

NOTE:
Also the database layout unit may not change during the execution of *space3d*.

7. APPENDICES

APPENDIX A -- Space Parameters used in the Green Module

Note that an 1) is added, when the parameter is evaluated only once (at first call of the function). Most parameters are evaluated by each call. Note that parameter values normally can not change, but (on user request) *Xspace* can read the parameter file again!

Source file: green/colloc.c

Variable	Name	Default	Function
collocationMode	collocation_mode	0	doCollocationGreen 1)
checkCollocation	check_collocation	0	doCollocationGreen 1)
an_turnover	an_turnover	100	doCollocationGreen 1)
state	accelerate_levin	off	seriesSum 1)

Source file: green/green.c

Variable	Name	Default	Function
testComputation	test_computation	off	greenPwl 1)

Source file: green/init.c, Function: greenInit, Name prefix: debug.

Variable	Name	Default	Note
printInit	print_green_init	off	
printGreenTerms	print_green_terms	off	
printGreenGTerms	print_green_gterms	off	

Source file: green/init.c, Function: greenInit, Name prefix: -

Variable	Name	Default	Note
useAdptvIntgrtn	use_adptv_intgrtn	off	
forceAdptvIntgrtn	force_adptv_intgrtn	off	
forceMpAnaInt	force_mp_anaint	on	
forceMpExInt	force_mp_exint	off	
useMultiPoles	use_multipoles	on	
testMultiPoles	test_multipoles	off	
divergenceTerm	min_divergence_term	3	value - 2
cMaxMpOrder	multipoles_cMaxMpOrder	cMaxMpOrder	range: 0...3
oMaxMpOrder	multipoles_oMaxMpOrder	oMaxMpOrder	range: 0...3

Note that the Collocation mode is only using cMaxMpOrder.

Source file: green/init.c, Function: greenInit, Name prefix: cap3d.

Variable	Name	Default	Note
useMeanGreenValues	use_mean_green_values	off	
mergedImages	merge_images	on	
multipolesMindist	mp_min_dist	2.0	
c/oMaxMpOrder	mp_max_order	2	range: 0...3
FeModeGalerkin	be_mode	False	True for 0g/1g
FeModePwl	be_mode	False	True for 1c/1g
useLowestMedium	use_lowest_medium	on	1)
useOldImages	use_old_images	off	1)
pointGreenDist	point_green	infinity	
collocationGreenDist	collocation_green	infinity	FeModeGalerkin
collocationGreenDist	collocation_green	0.0	!FeModeGalerkin
asymCollocationGreenDist	asym_collocation_green	infinity	
maxGreenTerms	-	1	nlayers < 2
maxGreenTerms	max_green_terms	500	range: 1...500
y_stretch	y_stretch	1.0	
greenEps	green_eps	0.001	
collocationEps	-	greenEps	!FeModeGalerkin
collocationEps	col_rel_eps	0.2*greenEps	FeModeGalerkin

Source file: green/init.c, Function: greenInit, Name prefix: sub3d.

Variable	Name	Default	Note
useMeanGreenValues	use_mean_green_values	off	
mergedImages	merge_images	on	
multipolesMindist	mp_min_dist	2.0	
c/oMaxMpOrder	mp_max_order	2	range: 0...3
sawDist	saw_dist	infinity	
edgeDist	edge_dist	0.0	
edgeEffects	-	False	edgeDist>sawDist
FeModeGalerkin	be_mode	False	True for 0g/1g
FeModePwl	be_mode	False	True for 1c/1g
useLowestMedium	use_lowest_medium	on	1)
useOldImages	use_old_images	off	1)
d	neumann_simulation_ratio	100.0	1) greenType=3
pointGreenDist	point_green	infinity	
collocationGreenDist	collocation_green	infinity	FeModeGalerkin
collocationGreenDist	collocation_green	0.0	!FeModeGalerkin
asymCollocationGreenDist	asym_collocation_green	infinity	
maxGreenTerms	-	1	nlayers < 2
maxGreenTerms	max_green_terms	500	range: 1...500
y_stretch	y_stretch	1.0	
greenEps	green_eps	0.001	
collocationEps	-	greenEps	!FeModeGalerkin
collocationEps	col_rel_eps	0.2*greenEps	FeModeGalerkin

APPENDIX B -- Files of the Green Module

The following files are needed to make the TARGET library "green.a".

```
--------------
Include files:
-------------------------------------------------------------
export.h     - external functions: to be included
               in source files in other modules
extern.h     - external functions: to be included
               in source files in this module (*)
gputil.h     - pointR3_t macro’s
green.h      - green module defines and typedefs and structs
images.h     - greenSeries defines (included by mkgreen.c)
-------------------------------------------------------------
(*) Note that the external variables are declared in each source file.

-------------
Source files:
-------------------------------------------------------------
analytic.c   - calculates analytical the potential integrals
colloc.c     - contains collocation functions
galerkin.c   - contains galerkin functions
gputil.c     - contains pointR3_t functions
green.c      - contains module top function green()
init.c       - contains init function greenInit() and greenStatistics()
intadptv.C   - contains integrate adaptive functions
integrat.c   - contains integrate functions
mkgreen.c    - contains getImageGroup() to get greenSeries
mpgreen.c    - contains the multipoles functions
-------------------------------------------------------------

-----------
Make files:
-------------------------------------------------------------
Makefile           - contains make procedure for "green.a"
../Config.mk       - is included in the "Makefile"
../M.architecture  - is included in the "Config.mk" file
-------------------------------------------------------------
```

APPENDIX C -- Defines of the Green Module

The module is compiled with the following #define’s.

```
    /* see: include/config.h */

CONFIG_XSPACE  /* Must be defined (space3d is sym.link of Xspace) */
CAP3D          /* Must be defined (capacitance 3D extraction) */
DEBUG          /* Is defined by CONFIG_XSPACE */
NO_SUB_RES     /* Is NOT defined (substrate extraction possible) */
```

APPENDIX D -- Auxil Functions used in the Green Module

```
    tick ("green");    /* see: auxil/proto.h, auxil/clock.c */
    tock ("green");    /* see: auxil/proto.h, auxil/clock.c */
```

tick() starts the "green" timer and tock() stops the timer. A timer can be stopped and restarted many times.

At the end of the space program the result of each specified timer (in this case "green") can be printed. *Space* starts also the timer "total time" at the begin of the program and stops the timer "total time" at the end of the program. The "real", "user" and "sys" times are measured. Printing of timing info can be put "on" with space parameter "print_time". Note that space option **-v** prints only timer "total time".

```
    say ("Hallo %s", "world!");  /* see: auxil/proto.h, auxil/say.c */
```

say() prints the program name before each message when ’char *argv0’ is set. If the message begins with a tab(’\t’), it is assumed to be a continuation, in that case the program name is not printed. Each message is printed to the stderr stream. If the last char of the format is not a white space char, a newline char is automatically given at the end of the message. A variable argument format can be used. say() uses the max_message_cnt counter. Note that space parameter "max_message_cnt" (default: -1) can set the counter. Note that the space program dies after max_message_cnt is reached.

message() works like say() but prints the message to the stdout stream and flushes the stream. It does not print a program name and does not use a counter.

verbose() works like message(), but prints only when verboseLevel is set. verbose() uses also the max_message_cnt counter. Note that the verboseLevel can be set with verboseSetMode(TRUE) or verboseSetLevel(TRUE).

```
    if (...) die ();    /* see: auxil/die.c */
```

The default die routine (is also called by signal handlers defined with catchSignals). It can do specific clean-up, before it calls exit(1).

```
    char *sp = strsave ("vacuum");    /* see: auxil/strsave.c */
```

strsave() calls NEW() to alloc strlen+1 chars for arg-string. It uses strcpy() to copy arg-string in allocated memory. The string can be free’d with DISPOSE().

```
    /* see: auxil/mprintf.c */
    char * s = mprintf ("%s %d %g\n", "abc", 1, 2.345);
    int n = paramLookupI ("multipoles_oMaxMpOrder", mprintf ("%d", oMaxMpOrder));
```

mprintf() returns a formatted string in a static buffer (like sprintf). Note that there is only one internal buffer. (buffer size is 2000).

```
    /* see: auxil/param.c */
    void paramError (char *param, char *s, ...);
```

Report error for parameter <param> to the "stderr" stream. At the end of the message a newline is printed.

```
/* see: auxil/param.c */
void paramSetOption2 (char *key, char *val);
```

Set an option. Can be used to set parameters from the command line. The ’key’ must be the parameter name. The option parameter list is another list than the parameter file list. The option parameter list is searched first to find a parameter! A search starts at the end of the list. Thus, the last stored version of a parameter is first found.

The following functions are used for parameter lookup:

```
/* see: auxil/param.c */
char * paramLookupS (char *key, char *dflt);
double paramLookupD (char *key, char *dflt);
int    paramLookupI (char *key, char *dflt);
bool_t paramLookupB (char *key, char *dflt);
```

Function paramLookupS() does the lookup for all Lookup functions. A verbose_flag can be set as debugging aid to get a message(). If the given key (parameter name) is not found, the dflt (default string) value is used.
By functions paramLookupD() and paramLookupI() the value can also be "inf" or "infinity". By paramLookupB() TRUE (=1) or FALSE(=0) is returned. The string value can be "on" or "off" or NULL. Note that a parameter without a value is also given the TRUE value.

APPENDIX E -- Auxil Macro’s used in the Green Module

```
/* see: auxil/malloc.h */
NEW(type, cnt)      /* local malloc() returns obj */
CLEAR(obj, size)    /* clear obj after NEW */
DISPOSE(obj)        /* free obj; set obj = 0 */

/* see: auxil/auxil.h */
Min(z1, z2)    /* return minimum of z1 and z2 */
Max(z1, z2)    /* return maximum of z1 and z2 */
Abs(z1)        /* return absolute value of z1 */
```