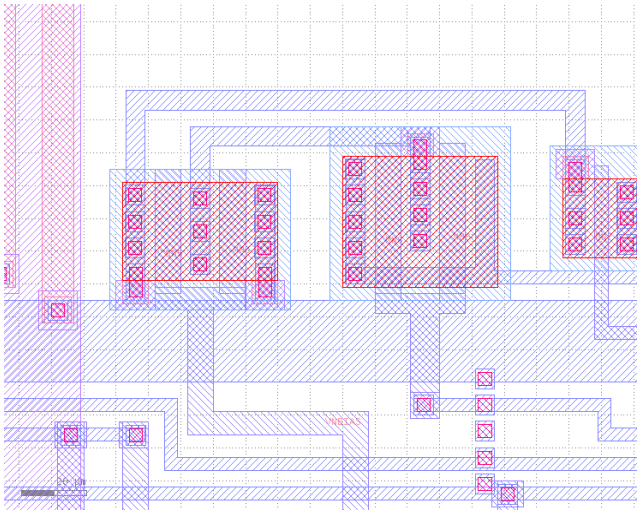
GDSII for the Rest of Us

Octave and MATLAB Toolbox for GDSII Libraries A Tutorial

Ulf Griesmann
ulf.griesmann@nist.gov, ulfgri@gmail.com



1. Purpose, Scope, and Limitations

The GDSII (**G**raphic **D**atabase **S**ystem **I**nformation **I**nterchange) library, or database, format [1] is a binary file format. Despite being poorly documented, it remains the dominant industry standard for the description of nano-structure designs that are fabricated with either photo- or electron lithography. GDSII library files are used to define the layout of integrated circuits, MEMS devices, nano-structured optics, and so on. This document describes a toolbox for Octave (www.octave.org) or MATLAB (www.mathworks.com) to create, read, and modify files in the GDSII library format. This toolbox is particularly useful when a layout is the result of numerical modeling as is often the case, e.g., for nano-structured optics or micro-fluidic devices. MATLAB or Octave then become very efficient tools for post-processing of modeling results and for creating a lithographic layout as input to the fabrication process.

GDSII was created as part of an electronics design system (EDS) by Calma, Inc. at about the same time the first IBM PC went on sale (1981). While the endurance of the file format is a testimony to the quality of its design, it is not surprising that parts of it are now thoroughly outdated. For example, the records that enable splitting of GDSII files and storing them on several tape reels are no longer useful at a time of multi-TB hard-drives. GDSII was originally part of EDS software and some records have little meaning outside the software. For example, FONTS records with the names of font definition files are read by the toolbox, but there is nothing to be done with them. MASK and ENDMASKS records are of no use when working with the toolbox. They are ignored if they are encountered during reading of GDSII files. Several other similarly obscure records are also ignored.

Of all programming languages devised, MATLAB / Octave may be the least suitable for writing and reading GDSII files. The lack of memory references (pointers), and the need to translate all data from and to array objects upon reading or writing a GDSII file, results in a substantial performance penalty. Still, the speed is acceptable for layouts with tens or even hundreds of millions of elements.

2. Toolbox Installation, Preparation, and Getting Help

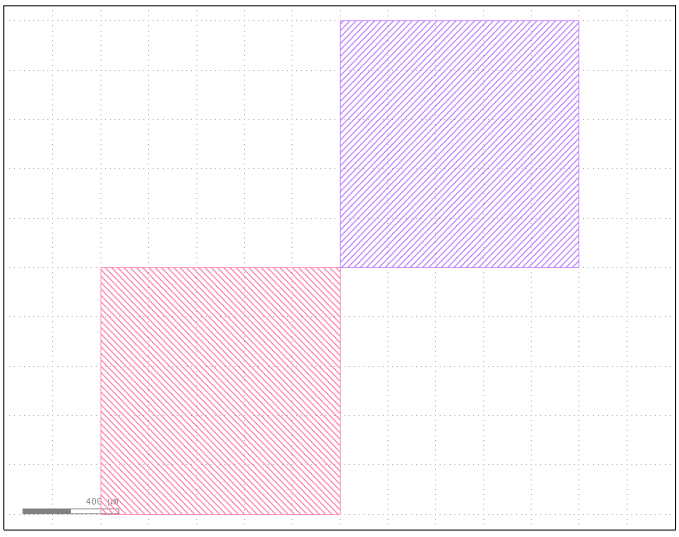
Unpacking the .zip archive 'gdsii-toolbox-nn.zip' creates a directory, or folder, './gdsii-toolbox' containing the software for the toolbox. The directory './gdsii-toolbox' and its subdirectories must be included in the search path of MATLAB or Octave. Next, external (MEX) functions that are part of the toolbox must be compiled. Detailed instructions are provided in the README file located in the ./gdsii-toolbox directory.

3. Libraries, Structures, and Elements

A library file in GDSII format contains one or more structures, or *cells* in micro-electronics design parlance. Each of the structures in a library contains elements, which are the basic building blocks of a layout. Structures have names; elements are nameless, but have properties. The GDSII elements are boundaries (simple, closed polygons), boxes, paths, text elements, nodes, structure references (sref), and array references (aref). It is the two reference elements, sref and aref, that enable the design of

hierarchical layouts described by a structure tree.

4. A Basic Example



The script example_basic.m, shown below, illustrates the basic steps involved in creating a GDSII library and library file (the script is contained in the gdsii_docs.zip archive). At the beginning of the script, a structure BASIC is created, that will contain the elements of the library file. Next, two polygons are defined and two elements, each containing one of the polygons in its xy property. The layer property of the first element is set to 1, the layer property of the other element is set to 2. The elements are added to the structure using the familiar array indexing syntax. Then, a library is created and the structure is added to the library. The structure could also have been added to the library using array indexing. It is not important in which order elements, structures, and libraries are created. In addition to library name, the library constructor can have several more property arguments of which the user unit and the database unit are the most important. The database unit is the length unit that is used to store the layout in the library file as a fraction of one meter; it defines the design grid of a layout. A database unit of 10^{-9} means that all data are stored in nano-meters. The user unit is typically a multiple of the database unit. E.g.: 10^{-6} specifies a user unit of 1 µm[1]. A database unit of 1 nm and a user unit of 1 µm are the defaults, because they are suitable for many layouts. Finally, the library is written to a layout file.

```
% create a structure to hold elements
gs = gds_structure('BASIC');

% create two closed polygons
xy1 = 1000 * [0,0; 0,1; 1,1; 1,0; 0,0]; % 1mm x 1mm
xy2 = bsxfun(@plus, xy1, [1000, 1000]);
%xy2 = xy1 + [1000,1000]; % for Octave or Matlab >R2016

% create boundary elements and add to the structure (on different layers)
gs(end+1) = gds_element('boundary', 'xy',xy1, 'layer',1);
gs(end+1) = gds_element('boundary', 'xy',xy2, 'layer',2);

% create a library to hold the structure and add the structure
glib = gds_library('TWO_BLOCKS', 'uunit',1e-6, 'dbunit',1e-9, gs);

% finally write the library to a file
write_gds_library(glib, 'basic.gds');
```

1 The header of a GDSII file stores the ratio database unit / user unit instead of the user unit itself.

Running this script with MATLAB or Octave will create the file basic.gds, which can be inspected using a layout viewer. The KLayout software [2] is an excellent GDSII viewer for the small and medium size layouts that are encountered in a typical research setting. When the file basic.gds is loaded into KLayout, the two boundary elements will be displayed as shown in the picture on page 3.

4.1 A Second Example

The following example script illustrates how to create a library that contains more than one struc-ture:

```
% create a structure containing a compound boundary element
xy1 = 1000 * [0,0; 0,1; 1,1; 1,0; 0,0]; % 1mm x 1mm
xy2 = bsxfun(@plus, xy1, [-1000, -1000]);
be = gds_element('boundary', 'xy',{xy1,xy2}, 'layer',4);
sa = gds_structure('A', be);

% create another structure containing a compound boundary element
xy3 = bsxfun(@plus, xy1, [-1200, 200]);
xy4 = bsxfun(@plus, xy1, [200, -1200]);
be = gds_element('boundary', 'xy',{xy3,xy4}, 'layer',6);
sb = gds_structure('B', be);

% create a top level structure
ts = gds_structure('TOP');
ts = add_ref(ts, {sa,sb});  % add sref elements to top level

% create a library to hold the structure
% NOTE: ALL structures must be added to the library !
glib = gds_library('FOUR_BLOCKS', 'uunit',1e-6, 'dbunit',1e-9, ...
                sa,sb,ts);

% finally write the library to a file
write_gds_library(glib, '!basic2.gds');
```

The script creates two gds_structure objects sa and sb, each containing a compound boundary element (see section 6) with two polygons each. Then a top level structure ts is created and references (sref elements) to structures 'A' and 'B' are added to the top level structure, which establishes the structure hierarchy. Next, **a gds_library** is created, and **all structures that make up the layout are added to the library.** The last line of the script writes the library to a file. In MATLAB/Octave a gds_library object can be thought of as an array of gds_structure objects, each of which is an array of gds_element objects. A gds_library is not represented by a tree data structure, as it would be e.g. in a language like C/C++. This implies that it is not sufficient to store only the top level structure in the gds_library object, because it does not contain memory references to other gds_structure objects. The structure tree is displayed using the treeview method of the gds_library class.

5. Elements – Types and Properties

In the the GDSII toolbox, elements are created with a call to the constructor `gds_element` of the gds_element class:

```
el = gds_element('type', 'prop1',val1, 'prop2',val2, ...);
```

The first argument of the gds_element function is a string specifying the element type; it can be one the the element types in the first row of Table 1. Element properties are specified in the usual MATLAB fashion using property/value pairs. Property names recognized by the toolbox are listed in the left column of the table in section 5. The properties of a gds_element object are displayed when the name of an element variable is typed at the command prompt. For example:

```
>> be1
be1 =

Boundary:
polygons: 1
layer = 1
dtype = 0
```

Once elements are created in memory, their properties can be modified using field name indexing. For example,

```
be.layer = 10;
```

will change the layer of element be to 10. Similarly,

```
be.prop = struct('attr',2, 'value','nameless');
```

will change the property structure associated with the element be. Element properties can be retrieved in the same way.

```
L = be.layer;
```

copies the layer value of element be to the variable L. Alternatively, element properties can be retrieved with the get method and changed with the set method of the gds_element class. For example:

```
P = get(be, 'xy');
```

copies the polygon (or cell array of polygons) defining the shape of element be to the variable P.

	boundary	box	path	node	text	sref	aref
layer	Layer where element is located. 0..255						
dtype	Data type 0..255				optional		
xy	Up to 8191 vertices of a polygon or cell array of polygons	Corners of the box (5)	Up to 8191 Vertices along a path	Up to 50 node locations	Text location [x,y]	Location(s) of the referenced structure; can be many.	Location of the referenced structure
strans					strans.reflect strans.absmag strans.absang **strans.mag** **strans.angle**		
sname						Referenced structure name	Referenced structure name
adim							Record with array dimension: **adim.row** **adim.col**
ptype			Path type: 0, 1, 2, or 4		Path type: 0, 1, 2, or 4		
width			width of path in user units		Width of line in user units		
ntype				Node type 0 - 255			
text					Text string		
ttype					Text type 0-63		
font					Text font 1-4		
verj					Vertical justification		
horj					Horizontal justification		
plex	A number that can be used for grouping of elements. A negative number signifies the plex head.						
prop	Structure array with attribute - name pairs that can be assigned to the element. **prop(k).attr, prop(k).name**						
elflags	A string ‘T’ for template data and ‘E’ for external data. Can be ‘TE’. Purpose unknown.						

Table 1: gds_element objects (top row) and their properties (left column). Valid properties for each element object type are highlighted in yellow.

6. Compound Elements, and How to Work With Them

Some layouts contain a large number of, e.g., boundary elements on the same layer and with identical properties. An example would be a diffractive optical element that is described by many boundary elements on the same layer. The GDSII toolbox introduces the concept of compound boundary, path, and sref elements, which make it more convenient and much more memory efficient to describe layouts with many similar elements in Octave / MATLAB. Boundary elements can contain more than one polygon with otherwise identical properties. Similarly, a path element can contain more than one path with the same properties, and an sref element can define references to a structure at many coordinates. When these compound elements are written to a file, the toolbox creates the appropriate number of individual elements in the layout file, unless the layout is written as a compound GDS (.cgds) file (see section 15).

The toolbox provides two mechanisms that make working with compound elements easier. For boundary and path elements, the `add_poly` method can be used to add additional polygons to an existing boundary or path element. For example,

```
cbel = add_poly(cbel, {xy1, xy2});
```

adds polygons `xy1` and `xy2` to a boundary (or path) element `cbel`. Boundary, path, and sref elements can be combined with other elements of the same type into single compound elements using the '+' operator for the gds_element class. For example,

```
cbel = be1 + be2 + be3;
```

transfers the polygons contained in elements `be1`, `be2`, and `be3` to a new element `cbel`. The other properties of element `cbel` are inherited from the first element `be1`.

Array indexing notation can also be used to address polygons in compound boundary and path elements and the locations in compound structure reference elements. For example:

```
cbel(3)
```

will return the 3rd polygon in a compound boundary element. Similarly,

```
cbel(3) = xy;
```

replaces the 3rd polygon in a compound boundary element `cbel` with a new polygon `xy`. Array indexing can be used to add additional polygons to an existing boundary element.

```
cbel(end+1) = xy;
```

adds a new polygon `xy` to the boundary element `cbel`, which is equivalent to using the `add_poly` method. Compound path and sref elements can be modified in the same way.

7. Element Conversion

Occasionally it may be necessary to convert elements to boundary elements. Box elements, for example, are ignored by some layout processing software and may have to be replaced by equivalent boundary elements. Text elements are only used for annotation in GDSII layouts and must be converted to boundary elements if they are to be part of the writable layout. Box, path, and text elements can be converted to boundary elements using the following methods of the gds_element class. The methods return boundary elements that are equivalent to the original elements. Text elements are rendered with polygons using the DEPLOF font by David Elata, Technion, Haifa, Israel.

Input element type	**`Conversion method`**
box	`boundaryel = poly_box(boxel);`
path	`boundaryel = poly_path(pathel);`
text	`boundaryel = poly_text(textel);`

8. Boolean Set Operations with Boundary Elements

The GDSII toolbox implements boolean set operations (union, intersection, difference, xor) for simple and compound boundary elements using the Clipper library by Angus Johnson (www.angusj.com). (The Clipper library is written in C++ and cannot be compiled with the inadequate LCC compiler that is included with pre-2012 versions of MATLAB on the Windows32 platform.) A boolean set operation is performed by invoking the `poly_bool` method as follows:

```
bc = poly_bool(ba, bb, 'and', 'layer',6);
```

where `ba` and `bb` are simple or compound boundary elements, and the third argument specifies the set operation. Four set operations are available: `'and'` specifies the set intersection, `'or'` the set union, `'notb'` (or `'diff'`) the set difference, and finally there is `'xor'`. The result is returned in boundary element `bc`, which has the same properties as element `ba`. Optionally, additional property/value pairs can be used to change the properties of the boundary element `bc`. Both `ba` and `bb` can be compound elements. Alternatively, Boolean set operations can be invoked using the operators in the following table:

Polygon union (or)	`bc = ba | bb;`
Polygon intersection (and)	`bc = ba & bb;`
Polygon difference	`bc = ba – bb;`
Polygon xor	`bc = ba ^ bb;`

Boolean set operations are performed on an integer grid with grid points spaced at one

database unit. Boundary vertices are snapped to the nearest grid point before a set operation is performed. Boolean set operations therefore require information about the user and database units, which are, however, not defined until the constructor for a GDS library, gds_library, is called. Either a library object must be created before the first call to poly_bool, or the function gdsii_units must be used to define the units before calling the poly_bool method. Otherwise, default values are used.

9. Methods for gds_element Objects

display	Method is invoked automatically to display the element properties.
etype	Returns a string with the element type. t = etype(elm);
is_etype	Checks if an element is of a specific type. is = is_etype(elm, 'path');
is_ref	Returns 1 if an element is either an sref or an aref element.
get	Retrieve element properties. l = get(elm, 'layer');
xy, prop, strans, sname	Return the respective element properties. strans and sname only operate on reference elements.
set	Modify element properties. elm = set(elm, 'layer',10)
bbox	Calculates the rectangular bounding box of boundary, path, and box elements.
add_poly	Adds one or more polygons to boundary and path elements. be = add_poly(be, {xy1, xy2, xy3});
poly_cw	Changes all polygons in a boundary element to clockwise orientation. be = poly_cw(be);
poly_iscw	Checks the orientation of polygons in a boundary element. Returns 1 for polygons with clockwise orientation.
poly_area	Calculates the area of a boundary element, or the sum of areas in a compound boundary element.
poly_bool	Performs Boolean set operations on boundary elements. E.g. pc = poly_bool(pa, pb, 'and'); calculates the set union of boundary elements pa and pb.
poly_box, poly_path, poly_text	Conversion functions that convert box, path, and text elements into equivalent boundary elements.
&, |, ^, -	Operators that invoke methods for Boolean set operations intersection (and), union (or), exclusive or (xor), and difference.
+ (plus)	Operator that combines boundary, path, and sref elements into compound elements.

10. Structures

In a GDSII library, elements are organized into named structures. Every library must contain at least one structure. A library can have multiple top level structures, but since this can lead to confusion when the layout is fabricated, it may be best to use only one top level structure in a layout file. A structure object is created by calling the constructor of the gds_structure class:

```
S = gds_structure(sname, varargin).
```

The first argument is the name of the new structure. It can be followed by one or more elements or cell arrays of elements which are added to the structure when it is created.

10.1. Accessing elements in structures

Elements can be added to structures, or copied from structures, using the familiar array indexing notation of MATLAB (Octave).

```
S(end+1) = be;
```

adds element `be` to structure `S`. Similarly, elements can be copied out of structures. For example, the following statement returns a cell array containing the first three elements in structure `S`:

```
ca = S(1:3);
```

The expressions S(:) and S(1:end) return a cell array containing all elements in a structure. The array notation is also useful to modify the elements contained in a structure. The loop

```
for k = 1:numel(S)
  if ~is_ref(S(k))
    S(k).layer = 5;
  end
end
```

moves all the elements contained in structure S onto layer 5. The name of a structure S, and its creation and modification dates can be accessed with field name indexing:

```
name = S.sname;    date = S.cdate;    date = S.mdate;
```

10.2. Creating a hierarchy of structures

GDSII layouts are generally a hierarchy, or tree, of structures, in which a structure can incorporate other structures by referencing them. A structure references other structures using sref and aref elements, which represent references to other structures, much like a scientific paper contains references to other papers. A structure only appears in a layout when it is a child of the top level structure via a chain of references. The gds_element constructor can be used to create sref and aref elements. A much more powerful, and easier to use, way of creating structure references (and array references) is the add_ref method of the gds_structure class. **'add_ref' is the most important function (method) of the toolbox.** In its simplest form, a reference is created using the name of the referenced structure. For example,

```
S = add_ref(S, 'CD_FEATURE', 'xy', [100,100]);
```

incorporates a reference to a structure with the name 'CD_FEATURE' into structure S at the position [100,100] in user units. add_ref implicitly creates an sref element that is added to structure S. Using the name of a structure in add_ref can be cumbersome and may result in errors when a structure is renamed. Instead of a structure name, gds_structure object can be passed to the add_ref function:

```
S = add_ref(S, cdfeature, 'xy', [100,100]);
```

The second argument of the add_ref method can also be a cell array of gds_structure objects or a cell array of structure names, which creates multiple structure references with the same properties. The position property 'xy' can be an nx2 matrix of locations, which creates n structure references at different positions. Internally, this multi-position structure reference is stored as a compound structure reference (see section 15) which is converted into individual sref elements when the library is written out to a file. When the adim property is present, an array reference is created instead of a structure reference. The add_ref method recognizes all properties of sref and aref elements (see Table 1 on page 5). For example, a structure R can be rotated and magnified using an strans property as follows:

```
sts.angle = 45; % degrees CCW
sts.mag = 2;    % scale up by factor of 2
S = add_ref(S, R, 'strans',sts);
```

10.3. Methods for gds_structure Objects

`add_element`	Adds one or more elements, or cell arrays of elements, to structures `S = add_element(S, e1, e2, {e3, e4, e5});`
`add_ref`	Used to build structure trees by adding references to structures. `S = add_ref(S, 'SOME_STRUC', 'xy',[0,0]);` adds a structure reference to structure SOME_STRUC to structure S.
`display`	Is invoked by MATLAB / Octave to display a structure.
`find`	Finds elements in a structure with specific properties. Example: `bel = find(S, @(x)is_etype(x,'boundary'));` returns a cell array containing all boundary elements in S.
`find_ref`	Finds the names of other structures referenced by a structure: `names = find_ref(S);` returns a cell array of structure names referenced in S.
`get`	When called only with a structure argument `get(S)` returns a cell array with all elements in a structure. When called with a numerical argument `get(S,k)`, the k-th element is returned. When called with arguments `sname`, `cdate`, or `mdate`, structure name or structure dates are returned.
`bbox`	Calculates the rectangular bounding box of a gds_structure object.
`numel`	Returns the number of `gds_element` objects in a structure.
`poly_convert`	Convert some or all box and path elements in a structure to boundary elements. Text elements are ignored.
`rename`	Renames a structure: `S = rename(S, 'new_name');`
`refrename`	Replaces a referenced structure name in all sref and aref elements that contain it.
`sdate`	Returns a structure's creation and modification dates: `[cdate, mdate] = sdate(S);`
`set`	Set method to set structure name `sname`, structure creation date `cdate`, and structure modification date `mdate`.
`sname`	`sname(S)` returns the name of structure S.
`structfun`	Iterator for the gds_structure class `res = structfun(S, @func);` applies function '`func`' to each element in the structure S and returns the function values as a cell array `res`.
`subsasgn`, `subsref`	Used by MATLAB/Octave for array indexing.

11. Libraries

A GDSII library contains all the structures that make up a layout and additional information, such as user and database units, needed for fabrication . A library object is created with the 'gds_library' constructor. For example:

```
L = gds_library('EXAMPLE.DB', 'uunit', 1e-6, 'dbunit',1e-9);
```

creates a library object L named 'EXAMPLE.DB' with user unit 10^{-6} m (1 µm) and database unit 10^{-9} m (1 nm). The units are the default units that are used when the unit properties are not specified. Structures are added to libraries in the same way elements are added to structures: either during library creation, using array index addressing, or by using the `add_struct` method. For example:

```
gdslib = gds_library('TESTLIB.DB', struc);
gdslib(end+1) = other_struc;
gdslib = add_struc(gdslib, {struc_a, struc_b});
```

Structures in libraries can be addressed either by their numerical index or by their name using structure field indexing. E.g. the 5th structure in a library L with the name DEMO can be copied either with `L(5)` or with `L.DEMO`[2].

11.1. Methods for gds_library Objects

`add_struct`	Adds structures or a cell array of structures to a library
`display`	Used by MATLAB/Octave to display the contents of a library
`get`	Returns all structures in the library when called with only the library argument, returns one structure when called with a numeric argument, or returns `lname, uunit, dbunit, numst, layer`.
`libraryfun`	Iterator for the gds_library class. Applies a function to all structures in a library and returns the return values in a cell array: `ca = libraryfun(L, @func);`
`numst, length`	Returns the number of structures in the library
`numel`	Returns the number of `gds_element` objects in the library.
`poly_convert`	Converts all path and box elements in a library into boundary elements. Text elements are ignored; the library hierarchy is preserved.
`snames`	Returns a cell array of the names of structures in a library. It also returns a struct with structure names as field names that can be

2 A warning is in order here. Structure field indexing of structures in libraries only works when the structure names are valid Octave/MATLAB variable names, which is more restrictive than the GDSII format permits.

	used to efficiently address structures by their names.
`rename`	Renames a library object.
`srename`	Renames a structure in a library and all structure references to the structure in the library.
`set`	Set or change library properties `lname, uunit, dbunit`, and `layer`.
`bbox`	Calculates the rectangular bounding box of a gds_library object.
`layerinfo`	Displays the distribution of elements over layers.
`subsasgn, subsref`	Used internally for array indexing notation.
`treeview`	`treeview(L);` displays the tree of structure references in a library. This is useful for finding errors in the library hierarchy.
`topstruct`	Displays the top level structures in a library object. Top level structures are not referenced by other structures in a library.
`subtree`	Returns a cell array containing a structure and all the structures that it references.
`write_gds_library`	Writes a gds_library object to a file: `write_gds_library(L, '!library.gds');` writes library object L to a file 'library.gds' the '!' at the beginning of the file name means that an existing file will be overwritten. Without the '!', existing files will be backed up.

12. Reading GDSII Library Files

A GDSII library file is read into memory with the `read_gds_library` function as shown in the following example:

```
L = read_gds_library('demo.gds');
```

This reads the contents of file 'demo.gds' and returns a `gds_library` object L. In verbose mode the function displays the structures in the layout file (in verbose mode), the number of elements in them, and the percentage of the file that has been read. Structure and element statistics are displayed once a library has been read into memory.

13. Multi-level Indexing

14. Higher Level Functions

The GDSII toolbox contains a number of higher level functions which are helpful in the creation of more complex layouts, for process control features, and to generate test patterns. The high level functions are grouped into functions that return gds_element objects (in ./gdsii/Elements) and functions that return gds_structure objects (in /gdsii/Structures). The help command can be used in the usual Octave / MATLAB fashion to display function-specific documentation for all functions.

14.1. Functions returning gds_element Objects

Function name	Description
gdsii_boundarytext	Renders a text string with characters made from boundaries suitable for lithographic reproduction.
gdsii_pathtext	Renders a text string with characters made from path elements.
gdsii_ptext	A wrapper for gdsii_boundarytext to maintain backwards compatibility.
gdsii_arc	A flexible function for arcs, arc segments, circles, and circle segments. Arcs are approximated by a boundary element.

14.2. Functions returning gds_structure Objects

Function name	Description
gdsii_bitmap	Creates a bitmap with black and white pixels. The black pixels are defined by an arbitrary boundary element.
gdsii_datamatrix	Creates an ISO/IEC 16022 DataMatrix barcode with ASCII encoding and ECC200 error correction from a user supplied text string.
gdsii_cdfeature	Generates a critical dimension pattern for lithography process control.
gdsii_pattern	Creates a gds_structure object containing one or more polygons.
gdsii_multref	Creates structure references at multiple, arbitrary locations. Easier to use and more flexible than an array reference.
gdsii_grating	Function for creating linear grating patterns.
gdsii_zonelens	A function to create generalized zone lenses with alternating black and white annular zones.
gdsii_replicate	Replicates a gds_structure object on a grid.
gdsii_checky	Generates a nested checker fractal pattern, useful for lithography tests.

gdsii_sierpinski	Generates a Sierpinski fractal pattern.

15. Miscellaneous Functions

Function name	Description
adjmatrix	Calculates the adjacency matrix that describes parent/child relationships in the tree of gds_structure objects. Input is a cell array of gds_structure objects, output is a sparse matrix.
topstruct	Calculates the top level structure in a cell array of gds_structure objects. This function is most useful when creating a reference element to the top structure of a cell array of gds_structure objects: S = add_ref(S, topstruct(caos))

16. Memory Management and Compound Layout Files

The concept of compound elements in the GDSII toolbox was introduced to enable the efficient creation of layouts with very large numbers of elements. For example, some layouts describing diffractive optics may require hundreds of millions of elements. When compound elements are written to a layout file, they are converted into sequences of individual elements. This causes a problem with Octave or MATLAB when the layout must be read back into memory, because far more memory is required to store the individual elements. For example, a compound sref element with 100 million coordinates requires only a relatively small amount of memory. Reading 100 million sref elements into Octave or MATLAB with the GDSII toolbox is, however, impossible because an excessive amount of memory is required. MATLAB was not designed for handling a very large number of small pieces of data. All data are represented by mxArray objects in Octave or MATLAB. An mxAarray object has a memory overhead of about 100 bytes, even if it is empty, or the data in the matrix contain only two bytes for information, as, for example, in the case of the layer property of an element. Another problem is that there is no way of knowing how many structures and elements are in a GDSII file before it is parsed, and how many of the optional properties are present in each element. The cell arrays holding structures and elements must be grown as the data from a GDSII file are read. This becomes problematic when the number of elements is very large and is another reason why the memory use of the read_gds_library function can become excessive.

The GDSII toolbox supports a non-standard type of GDS layout file, called a compound layout file with file extension .cgds. This format stores compound elements in GDS elements with multiple XY records. A compound boundary element is written to a GDS boundary element with multiple XY records, each representing a boundary. Similarly, a compound sref element is written to a GDS sref element that has an essentially unlimited number of XY records containing a matrix of coordinates instead of a single coordinate pair. Compound layout files can be processed much more efficiently with Octave or MATLAB, especially during reading of a layout file, in situations when a large number of elements with the same properties is

needed. While the .cgds format cannot be read by layout viewers such as Klayout, it is useful for storing partial layouts, or structure libraries, that can be efficiently read back when a final layout is created. See the help text of the `write_gds_library` for more details.

17. Choice of Units

The meter is the dimensional unit used in the GDSII format. All positions, polygon vertices, and so on are stored in a GDS file as 4-byte integer multiples of a smallest length called the database unit ('dbunit' in the toolbox). The database unit is defined when a library is created, the default database unit is 10^{-9} m = 1 nm. At the level of Octave (Matlab), positions are specified in a user unit ('uunit' in the toolbox) that is usually different from the database unit. The user unit is also defined during library creation, the default length of the user unit is 10^{-6} m = 1 µm, which is convenient for many layouts. When changing either the user or database unit it is important to consider the effect of rounding when a length is converted from user units to *integer* database units. For example, with the user and database units set to their default values, a length of 10.5 nm = 0.0105 µm will be stored as 11 database units. The rounding may be significant e.g. for a very high resolution e-beam lithography system, and in this case a smaller database unit should be chosen. The number of significant digits after the decimal marker in a length specified in user units is given by $\log_{10}$(uunit/dbunit).

References

[1] http://en.wikipedia.org/wiki/GDSII
[2] KLayout – http://www.klayout.de