The External Representation of Block Designs

Peter Dobcsányi and Hatem A. Nassrat

December 10, 2008

Version: 3.0 β

This document is based on *The External Representation of Block Designs v1.1* by Peter J. Cameron, Peter Dobcsányi, John P. Morgan, and Leonard H. Soicher [8].

Please send comments, questions, bug reports to extrep at designtheory dot org .

Contents

1 Introduction

This document should be of interest to those working in combinatorial or statistical design theory, as well as those interested in the development of standard electronic formats for mathematical objects.

The *External Representation of Designs* is to be used to store designs and their combinatorial, group theoretical and statistical properties in a standard platform-independent manner (*external* means external to any software). This will allow for the straightforward exchange of designs and their properties between various computer systems, including databases and web servers, and combinatorial, group theoretical and statistical packages. The external representation will also be used for outside submissions to our design database.

We have concentrated our initial development effort in the area of block designs, and in this document we present our standard for the *External Representation of Block Designs*. We shall give a full explanation and provide examples. We have tried to make the document readable by non-experts, since we don't expect everyone to be an expert in all the areas covered.

1.1 A Simple Example

We start with a simple example. It is a list of designs, in our external representation, containing a single design, known as the *Fano plane*.

```
{
  "type" : "block_design",
  "id" : "t2-v7-b7-r3-k3-L1-0",
  "v" : 7,
  "b" : 7,
  "blocks" : [
    [0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6],
    [2, 4, 5]
  ]
}
```

The design is described in JSON. Later in this document we discuss why we have chosen JSON for the specification of block designs. For now, if we look at the example above code, we will see listed within the list structure under the key `designs` the specification of the named design: it has $v = 7$ points, $b = 7$ blocks, and the seven blocks are listed (the first one being $[0, 1, 2]$). The design also includes a string identifier. This identifier is a remiminent of the previous Ext Rep [8]. This attribute will be discussed in the section (8.1) discussing essential properties of block designs.

In JSON there are two main structure and a set of basic data types. The main structures being the `list` structure and the `object` structure. If you are familiar with scripting languages like python, and JavaScript the JSON list structure is identical in syntax. Where elements of a list are separated by commas. The object structure denotes a mapping data structure where a key maps to a value. Both list elements and object values can be any JSON data type or structure where as object keys need to be a basic data type. These data structures are used to build the external representation of block designs described in this document. Design properties are represented by

keys within the various JSON objects within one design, where in some cases for compression they have been grouped into a simple list structure when appropriate.

This document contains our specification of an "external representation" of block designs, together with an explanation of the terms used, and some justification for doing it in the way we have chosen.

2 What is a Block Design?

Block designs are viewed in different ways by combinatorialists and statisticians. To a statistician, a block design is a set of "plots" or "experimental units" which carries a partition into "blocks", and a function from this set to the set of "treatments". A combinatorialist regards the set of treatments as basic (and usually calls them "points"), and identifies each block with the multiset of treatments occurring on plots in that block; thus, a block design is a set of points together with a multiset of multisets of points.

A multiset is essentially the same thing as a sorted list which may contain repeated items. In this documentation, and in the external representation of block designs, we represent a multiset as a list in square brackets [ ],

For the purpose of this specification, we have chosen to use the representation as a multiset of multisets.

Here is a small example. Suppose that we have six plots, numbered $1, 2, 3, 4, 5, 6$, with blocks $[1, 2, 3]$ and $[4, 5, 6]$. Suppose that treatment A is applied to plots $1, 2, 4, 5$, and treatment B to plots 3 and 6. Then we represent the block design as having point set $[A, B]$ and blocks $[[A, A, B], [A, A, B]]$. (Since the lists are sorted, we would represent the design in same way even if, say, treatment B was applied to plots 1 and 5.) The names of the plots have disappeared, but the plots can be recovered as incident point-block pairs or "flags". We always represent block designs in this way.

In this example, blocks have the awkward property that they are multisets (rather than sets) of points. While this does occur in practice, we have decided to exclude such designs for the time being, for various reasons. A block design is called *binary* if no treatment occurs more than once in a block, that is, if the blocks are represented by sets (rather than general multisets) of points. *All block designs in this document will be binary.*

Here is an example of a binary design. It is the Fano plane from the Introduction, viewed in a slightly different way. There are 21 plots, partitioned into seven blocks of three; there are seven treatments, numbered from 0 to 6, as shown in the following table (whose columns represent the blocks):

0	0	0	1	1	2	2
1	3	5	3	4	3	4
2	4	6	5	6	6	5

Further details can be found in the items on block designs in the Encyclopaedia of Design Theory, or in our survey paper [3].

3 The Concept of External Representation

The concept of External Representation of designs can be best understood through its role in the operation of the Design Theory Resource Server (DTRS). DTRS has many faces, it will be an ever growing database of designs, application server, web server for design related online documents, software repository etc. The main purpose of the external representation is to provide a platform independent method for information exchange about designs. With other words, the external representation acts as a *communication protocol* specialized for "talking about designs". This protocol is used in communication between various components of DTRS and its users. Here the concept "users" covers both human and software agents. Some examples for such communicating agents: database back-end for storing designs, middle layers between the database and the web and/or application servers, a researcher uploading some particular collection of designs, a user searching for designs having given properties, a statistical application program directly accessing the DTRS database etc. While these agents are free to use any internal representation of designs, they must use the standard external representation when they communicate with each other.

The external representation is used in three main areas:

1. An external representation is a formalism to encode various classes of designs as mathematical objects together with their most important properties. Many of these properties are complex mathematical objects in their own right.
2. The external representation can define invariants of a given list of designs. The use of such invariants provides a method for formulating complex queries about designs. A query will be expressed in terms of list invariants and the reply to this query will be a list of designs satisfying these invariants.
3. The external representation will be used as a specification tool to determine the content and, to some extent, the structure of the DTRS design database. Note, however, the database's internal representation can (and probably will) be quite different from this format.

Based on the above functionalities, we have determined the main technical requirements for an external representation as it follows.

- It can express the particular mathematical structures.
- It represents a hierarchical structure: a rooted, labelled tree.
- It is hardware/software platform independent and text based.
- It can be easily parsed.

To satisfy the above requirements the authors of [8] discussed multiple options. They looked into Lisp S-expressions and XML as possibilities for their external representation. And decided to use XML to implement their external representations versions 1.1 [8] and 2.0 [7]. The main driving force behind their decision was due to the practical considerations, since there are a wide variety of tools available to parse and print XML formats. However, when working with XML one quickly

sees that it takes time for the human eye to find information within an XML document. This paper proposes the external representation for block designs version 3.0 implemented in JavaScript Object Notation (JSON) [10]. This languages has many advantages over its predecessor XML in terms of readability and reduced syntax. It does not carry the verbose quality of XML where tags that denote subtrees require repetition. It is also more readable since it is easy to see the native tree structure using the JSON data structures. It also satisfies the requirements for transfer of our mathematical content due to its native support for numbers, booleans and strings when needed. The example snippet displayed earlier along with more to come expose these qualities. Since its creation in 2006 JSON has grown very popular in the data exchange realm due to its simplicity and many parsers/dumpers have been written in many programming languages [1] which emphasizes its practicality as a communication protocol.

In this documentation we focus on encoding block designs and their properties, while considering human readability. This focus deals mainly with the first element in the quoted list of areas that the external representation of block designs deals with. The other two functionalities and other types of designs are subjects of further research and development.

In order to document the external representation trees implemented in JSON we use a novel schema language reminiscent of the Relax NG [2] schema language for XML. The complete schema can be seen in Appendix A of this document. Snippets will be placed within the sections of this document to explain the related external representation structures and the JSON structures they produce.

4 Schema Language

When discussing a data representation a schema language is essential in order to explain how the data looks and how it may be formed. Although examples may help explain how the data should be presented, a formal schema is required to guide the users and serve as a reference when writing complying documents. This schema is quite simple and is meant to resemble the language of choice for the external representation (JSON) as to simplify the understanding process.

The schematic description of a JSON structure is the structure itself. In addition it is augmented with some of the standard BNF symbols, specifically the ones listed in the following table:

()	Grouping
?	Optional
|	An OR expression
*	Repition of zero or more
+	Repition of one or more

These symbols are used slightly differently in our schema than their use in standard BNF. In our schema the binary operators `?` `|` `*` `+` are placed prior to the entity they act upon. This choice was made to add readability to the schema, especially when using large sub-structures in the schema.

Since all of our internal tree node identifiers, represented by "object keys" in JSON structures, are strings that do not contain any white space or similar special characters, we have removed the

quotation marks needed in JSON from the schema language. Similarly some quotation marks were removed from string literals in the value segments of the structure when they need not be there. This makes the schema languages easier to see and read. Otherwise any character appearing in the schema is expected to be in a conforming "Ext Rep" v3.0 (short for external representation) document. Needless to say, that the BNF symbols serve as syntax drivers and the produced expansion is what the conforming document should match. The following example will help explain the rest of the features of our simple schema language:

```
<map> = {
    ( <preimage> | <preimage_cardinality> | blank ) ,
    image   : ( <number> | not_applicable )
};

<preimage> = preimage : [
      $integer *( , $integer )
    | [ $integer *( , $integer ) ] *( , [ $integer *( , $integer ) ] )
    | entire_domain
];

<preimage_cardinality> = preimage_cardinality : $integer ;

<number> = $integer | $float | <rational> ;

<rational> = { Q : [ $integer, $integer ] } ;
```

From the above example we can see how basic data types are represented in this schema language. The basic data types in JSON are number, string, and boolean [10]. In our schema we split number into either integer, floating point or rational ($\frac{a}{b}$). Each of these types are represented in the schema when preceded with a $ sign. The angle brackets (<>) surrounding the identifiers denote a place holder. They can match a definition, i.e. when they are being defined, and they would precede an equal (=) sign. These place holders may also occur within a definition, signifying that they should be replaced with what was defined upon them. It is a rule for this schema language that in order to use a place holder it has to be previously defined in the file. Documents which conform to our schema are said to be "Extrep v3 Compliant".

5 Indexing and Functions

We describe here some conventions referring to the indexing of objects, and the representation of functions.

5.1 Indexing and Ordering

We adopt the convention that, if a block design has v points, then the points are the integers $0, 1, \ldots, v-1$. This is a combination of two assumptions: the points are ordered; and the index set starts at 0 (rather than 1).

There are several choices of ordering of sets (or multisets) of points. We have chosen to order in the following way:

- first compare the length of the two lists; the shorter comes first.
- for lists of the same length, we order lexicographically. (Recall that the lists are sorted.)

So for example, here are a few sets in order:

$$[2], [0, 1], [0, 2], [1, 2], [1, 2], [0, 1, 3], [1, 2, 3], [0, 1, 2, 3]$$

For the purpose of defining functions on the collection of blocks, we now index these blocks from 0 to $b - 1$, where b is the number of blocks of the design. If the above list contains all the blocks of a certain design D, then we can refer to block 5 of D, which will be the set $[0, 1, 3]$ in this case.

The same principle can be extended to lists of lists. Assuming that the "inner" lists are already ordered, we first compare the length of the two lists, and if they are equal, we order the lists "lexicographically" (with the order previously defined between list elements). This process can be continued recursively to any level of nesting.

However, we do not require that this ordering is adhered to throughout the tree. Nevertheless, the following objects must be ordered:

- `cycle_type`
- `blocks`

The following objects may be required to be ordered (they have a boolean attribute `ordered`):

- `function_on_indices`
- `function_on_ksubsets_of_indices`
- `canonical_variances`
- `canonical_efficiency_factors`

Functions on indices, and on k-subsets of indices, are described next. For cycle types, see the section 8.4 on Automorphisms.

5.2 Functions and Index Flags

A function f with finite domain can be given by listing all $(x, f(x))$ pairs. Note that this list when spelled out in print can be a very large one, in particular, if the x-s are complex objects on their own. To help on this problem we can do several things:

- Instead of using x-s themself we use only indices referring to them. `function_on_indices` is defined to do this.

  The underlying principle is that if the external representation explicitly contains the related objects in a well defined (canonical) order then, in general, we use indexing as a way to refer to these objects. Nesting, in this sense, is not allowed.

- Frequently the domain of our functions is the set of k-subsets of some of our objects. `function_on_ksubsets_o` is defined for this situation.

- Regarding the $(x, f(x))$ pair, we allow several kinds of "contractions" (see `map` below):

  - If different x-s map to the same image, then instead of listing all these pairs we say $(\{x, x_1, x_2, \ldots\}, f(x))$. If the function f has just one image $f(x)$ we may say (entire_domain, $f(x)$).
  - Sometimes the user is not interested in the preimage $\{x, x_1, x_2, \ldots\}$ of $f(x)$, but only in its cardinality, so we allow $(|\{x, x_1, x_2, \ldots\}|, f(x))$.
  - Finally, we even allow leaving the preimage part of the pair blank, just giving the list of function values $(, f(x))$.

In fact, the user may only be interested in the image cardinality, in which case the entire function body may be blank.

In more detail, the `function_on_indices`, is schematically (Section 4) described as:

```
<function_on_indices> = {
       domain            : ( points | blocks ) ,
       n                 : $integer ,
       ordered           : ( true | unknown ) ,
    ?( image_cardinality : $integer , )
    ?( precision         : $integer , )
    ?( title             : $string , )
        maps              : [ ?( <map> *( , <map> ) ) ]
};
```

This specifies a function on either points or blocks. `n` is the cardinality of the domain. `ordered` specifies whether the function entries are ordered (by preimages): if the function body is not blank and each preimage is given explicitly or (if there is just one function image) or as the one element list containing the string "`entire_domain`" (i.e. neither as a `preimage_cardinality` nor `blank`), then the value of ordered must be "true", otherwise it is "unknown". `precision` is required if the function values are real numbers and specifies the precision to which they have been computed. A function is given by a sequence of `map`'s, each of which is specified as follows:

```
<map> = {
    ( <preimage> | <preimage_cardinality> | blank ) ,
    image   : ( <number> | not_applicable )
};

<preimage> = preimage : [
      $integer *( , $integer )
    | [ $integer *( , $integer ) ] *( , [ $integer *( , $integer ) ] )
    | entire_domain
];

<preimage_cardinality> = preimage_cardinality : $integer ;
```

For an example of the use of `function_on_indices`, see section 8.5 on Resolutions.

The `function_on_ksubsets_of_indices` specification works in the same way when the domain consists of all sets of points or blocks of fixed size k:

```
<function_on_ksubsets_of_indices> = {
       domain_base       : ( points | blocks ) ,
       n                 : $integer ,
       k                 : $integer ,
       ordered           : ( true | unknown ) ,
    ?( image_cardinality : $integer , )
    ?( precision         : $integer , )
    ?( title             : $string , )
        maps              : [ ?( <map> *( , <map> ) ) ]
};
```

For an example of its use, see the section 8.3.1 on Point concurrences.

We use the concept of `index_flag` to store an element in a list of "fuzzy booleans":

```
<index_flag> = "$integer" : ( $boolean | unknown ) ;
```

For example, we may want to record for which values of α a design is α-resolvable; for each value of α, the answer may be "true", "false", or "unknown".

6 Permutation groups

Permutation groups appear in many areas of design theory, in particular as automorphism groups of designs.

The specification of an permutation group is:

```
<permutation_group> = permutation_group : {
         degree : $integer ,
         order  : $integer ,
```

```
        domain : points ,
        <generators>
    ?( , <permutation_group_properties> )
};

<permutation_group_properties> = permutation_group_properties : {
    <permutation_group_properties_member> *( , <permutation_group_properties_member> )
};

<generators> = generators : [ ? ( <permutation> *( , <permutation> ) ) ];

<permutation> = [ $integer *( , $integer ) ];
```

There are four compulsory properties:

`degree`
: An attribute giving the number n of points on which the permutations are defined (the permutation group will then act on the indices $\{0, \ldots, n-1\}$).

`order`
: An attribute giving the number of permutations in the group.

`domain`
: An attribute specifying the domain indexed by the points $0, \ldots, n-1$.

`generators`
: A list of permutations which generate the group. A permutation is represented by the ordered list of its values (the images of the points $0, \ldots, n-1$ under the permutation).

For example, the permutation group which is the automorphism group of our Fano plane can be given as:

```
  "permutation_group" : {
    "degree" : 7,
    "order" : 168,
    "domain" : "points",
    "generators" : [
      [1, 0, 2, 3, 5, 4, 6], [0, 2, 1, 3, 4, 6, 5], [0, 3, 4, 1, 2, 5, 6],
      [0, 1, 2, 5, 6, 3, 4], [0, 1, 2, 4, 3, 6, 5]
    ]
  }
```

There are also various properties which can optionally be specified:

`primitive`
: True if the group acts primitively on points. A permutation group is *primitive* if it preserves no non-trivial equivalence relation. By convention, we assume that a primitive group is transitive (that is, any point can be carried to any other by some group element). (So the trivial group acting on two points is not primitive.)

`generously_transitive`, `multiplicity_free`, `stratifiable`
: Each orbit of the group acting on the set of ordered pairs of points can be represented by a matrix of zeros and ones of order n (which can be thought of as the characteristic function of the orbit). These *basis matrices* span the *centraliser algebra* of the group (the algebra of all matrices commuting with the group elements). Now the group is *generously transitive* if all the basis matrices are symmetric; it is *multiplicity-free* if the basis matrices commute; and it is *stratifiable* if the symmetrised basis matrices commute. Each concept implies its successor in the order given.

  A transitive permutation group is generously transitive iff any two points can be interchanged by some element of the group; it is multiplicity-free iff no irreducible constituent of the permutation character occurs with multiplicity greater than 1; and it is stratifiable iff the orbits of the group on unordered pairs form an association scheme. All these properties are false if the group is not transitive.

`no_orbits`
: The number of orbits on points. The group is transitive exactly when there is just one orbit on points.

`degree_transitivity`
: The maximum number s such that the group is s-transitive on points (that is, any s-tuple of distinct points can be carried to any other by some group element).

`rank`
: The number of orbits of the group on the set of ordered pairs of points. Note that this is defined for any permutation group; if the group is transitive, it is equal to the number of orbits of the stabiliser of a point.

`cycle_type_representatives`
: see below

The *cycle type* of a permutation is the multiset of its cycle lengths (when it is written as a product of disjoint cycles). The element `cycle_type_representative` consists of a cycle type and an element of the group having that cycle type, and optionally the number of elements of the group having that cycle type. `cycle_type_representatives` is a list of these `cycle_type_representative` elements, one for each cycle type represented by an element of the group.

For the example above, there are five cycle types, $[7], [1,2,4], [1,3,3], [1,1,1,2,2]$, and $[1,1,1,1,1,1,1]$ (the last being the identity). The cycle type representative for the second type is:


```
{
  "permutation" : [0, 3, 4, 5, 6, 1, 2],
  "cycle_type" : [1, 3, 3],
  "no_having_cycle_type" : 56
}
```

7 Numerical Data Types

Some of the numerical data in the external representation are the result of possibly inexact computations. Basically, there are three sources of this inaccuracy:

- The inaccuracy of the finite floating point representation.
- Arithmetical errors during computation.
- Cutting short an otherwise infinite approximation process.

The end result is that, in general, numbers in the external representation can be considered correct only within certain limits. We say they are "precise" up to some significant figures (see the details below).

The external representation version 1.1 provides the following *numerical data types*:

- Arbitrary precision *integers* (schematically $integer).
- Arbitrary precision *rationals* (schematically `<rational>`) written in $\{q : [a, b]\}$ format where a (the numerator) and b (the denominator) are both integers.
- Floating point *decimals* (schematically `$float`) up to some given precision specified as the number of significant digits.

A conforming software implementation must provide the corresponding internal representations.

Here are the rules for representing numerical data in the external representation:

- If a number is the result of an inexact computation then it must be represented using the *decimal data type*.
- The decimal representation of an inexact number must always contain the decimal point regardless the number would round up to an integer.
- Exact numbers must be represented either using the *integer* or the *rational data type*.

The precision of decimal numbers is indicated by an optional attribute `precision` of particular elements. The entries from our schema which can have this attribute are: `<list_of_designs>`, `<block_design>`, `<function_on_...>`,
`<statistical_properties>`. The rationale for having many elements with the optional precision attribute is to provide flexible scoping rules and avoid unnecessary repetition.

The `precision` attribute gives the *number of significant figures* of all decimal numbers in the tree whose root contains the attribute. This precision can be overridden by giving different precision in one or more subtrees. In general, a precision of a decimal number is the precision given in the root of the smallest subtree containing the number and with a root having a specified precision attribute. If an external representation document contains any data which is the result result of inexact computation, precision must be specified.

8 Block Designs

Recall our blanket assumption that all block designs are *binary*: this means that no treatment occurs more than once in a block, so that the blocks are sets rather than general multisets. However, it can happen that the same set occurs more than once in the list of blocks; that is, the list of blocks may be a multiset. In this case we say that the design has *repeated blocks*.

8.1 Essential Properties

The specification of `block_design` is as follows:

```
<block_design> = {
         type      : block_design ,
         id        : ( $string | $integer ) ,
         v         : $integer ,
    ?(   b         : $integer , )
    ?(   precision : $integer , )
         <blocks>
    ?( , <point_labels> )
    ?( , <indicators> )
    ?( , <combinatorial_properties> )
    ?( , <block_design_automorphism_group> )
    ?( , <resolutions> )
    ?( , <statistical_properties> )
    ?( , <alternative_representations> )
    ?( , <info> )
};
```

The first four components of the specification are:

`id`
: An identifier for the design, unique within the given document.

`v`
: An attribute giving the number of points.

`b`
: An attribute giving the number of blocks (optional).

`blocks`
: The list of blocks (as described above). The list must be ordered:

```
<blocks> = blocks : [  <block> *( , <block> ) ];

<block> =  [ $integer *( , $integer ) ];
```

Here is the design from the example in the Introduction, including only the components above:

```
{
  "type" : "block_design",
  "id" : "t2-v7-b7-r3-k3-L1-0",
  "v" : 7,
  "b" : 7,
  "blocks" : [
    [0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6],
    [2, 4, 5]
  ]
}
```

All these components, except the attribute `b`, are essential. The subsequent elements are optional. The first optional element is `point_labels`. If, for example, the design has been built from a set of points in a projective geometry, the point labels might be the coordinates of the points. More important for applications, the point labels could be the actual treatments associated to the points in the experimental plan (after randomisation). The point labels, if present, should form a list of length v.

The identifier must be unique within a given Ext Rep document. However, it cannot be guaranteed that this identifier is globally unique unless matched against a database containing all combinatorial designs. A design with some arbitrary "id" would be sent to the database for approval, once allowed in the database, a globally unique identifier would be generated.

8.2 Indicators

Indicators are boolean variables which record certain properties which a block design may have. We have included the following indicators:

`repeated_blocks`
: True if the same set occurs more than once in the list of blocks.

`resolvable`
: True if the design has a *resolution*, which is a partition of the blocks into subsets called *parallel classes* or *resolution classes*, each of which forms a partition of the point set.

`affine_resolvable`
: True if the design is *affine resolvable*, which means that the design is resolvable and any two blocks not in the same parallel class of a resolution meet in a constant number μ of points. If the design is affine resolvable then we optionally give this constant μ (unless the design consists of a single parallel class, in which case μ is not defined).

`equireplicate`
: True if each point lies in a fixed number r of blocks. If so, then we also optionally give the replication number r.

`constant_blocksize`
: True if each block contains a fixed number k of points. If so, then we optionally also give the block size k.

`t_design`
: True if the block design is a t-design for some $t > 1$. This means that the design has constant block size and that any t points are contained in a positive constant number λ of blocks. If so, then we optionally give the maximum value of t for which this holds.

`connected`
: True if the incidence graph of the block design is a connected graph. (The *incidence graph* or *Levi graph* of a block design is the bipartite graph whose vertices are the points and blocks of the design, a point and block being adjacent if the point is contained in the block.) We optionally give the number of connected components of the incidence graph.

`pairwise_balanced`
: True if $v > 1$ and the number of blocks containing two distinct points is a positive constant λ. If so, then we optionally give this λ.

`variance_balanced`
: True if $v > 1$ and the intra-block information matrix has $v - 1$ identical, nonzero eigenvalues. Equivalently, the $v - 1$ canonical variances are all equal (and finite). For definitions of terms used here, see section 8.6 on Statistical Properties.

`efficiency_balanced`
: True if $v > 1$ and the $v-1$ statistical canonical efficiency factors are identical and nonzero. For equireplicate designs, this is equivalent to variance_balanced, but not genenerally otherwise. Also see the section 8.6 Statistical Properties.

`cyclic`
: True if the design has an automorphism which permutes all the points in a single cycle.

`one_rotational`
: True if the design has an automorphism which fixes one point and permutes the other $v - 1$ points in a single cycle.

In the last two cases, an automorphism with the stated properties can be found under `cycle_type_representatives` described in section 8.4 on Automorphisms.

The several different sorts of balance are explained in the Encyclopaedia. For a (binary) design with constant block size, variance balance reduces to pairwise balance. For a equireplicate (binary) design with constant block size, efficiency balance reduces to pairwise balance.

The schematic description for indicators is:

```
<indicators> = indicators : { <indicator> *( , <indicator> ) } ;

<indicator> =
      repeated_blocks     : $boolean
    | resolvable          : $boolean
    | affine_resolvable   : ( $boolean | { mu           : $integer } )
    | equireplicate       : ( $boolean | { r            : $integer } )
    | constant_blocksize  : ( $boolean | { k            : $integer } )
```

```
    | t_design            : ( $boolean | { maximum_t     : $integer } )
    | connected           : ( $boolean | { no_components : $integer } )
    | pairwise_balanced   : ( $boolean | { lambda        : $integer } )
    | variance_balanced   : $boolean
    | efficiency_balanced : $boolean
    | cyclic              : $boolean
    | one_rotational      : $boolean
;
```

The indicators for our example are:

```
  "indicators" : {
    "repeated_blocks" : false,
    "resolvable" : false,
    "affine_resolvable" : false,
    "equireplicate" : {
      "r" : 3
    },
    "constant_blocksize" : {
      "k" : 3
    },
    "t_design" : {
      "maximum_t" : 2
    },
    "connected" : {
      "no_components" : 1
    },
    "pairwise_balanced" : {
      "lambda" : 1
    },
    "variance_balanced" : true,
    "efficiency_balanced" : true,
    "cyclic" : true,
    "one_rotational" : false
  }
```

8.3 Combinatorial Properties

Combinatorial properties are those which can be computed exactly from the list of blocks of the design. We include the following:

```
<combinatorial_properties> =
    combinatorial_properties : {
        <point_concurrences> ,
        <block_concurrences> ,
        <t_design_properties> ,
        <alpha_resolvable> ,
        <t_wise_balanced>
};
```

```
<point_concurrences> = point_concurrences : [
    <function_on_ksubsets_of_indices> *( , <function_on_ksubsets_of_indices> )
];

<block_concurrences> = block_concurrences : [
    <function_on_ksubsets_of_indices> *( , <function_on_ksubsets_of_indices> )
];

<t_design_properties> = t_design_properties : {
    t_design_properties_member *( , t_design_properties_member )
};

<t_design_properties_member> =
      parameters : {
          t      : $integer ,
          v      : $integer ,
          b      : $integer ,
          r      : $integer ,
          k      : $integer ,
          lambda : $integer
      }
    | square                : $boolean
    | projective_plane      : $boolean
    | affine_plane          : $boolean
    | steiner_system        : ( $boolean | { t: $integer } )
    | steiner_triple_system : $boolean
;

<alpha_resolvable> =  { <index_flag> *( , <index_flag> ) };

<index_flag> = "$integer" : ( $boolean | unknown ) ;

<t_wise_balanced> =  { <t_index_flag> *( , <t_index_flag> ) } ;

<t_index_flag> = "$integer" : ( $boolean | unknown | { lambda: $integer } ) ;
```

8.3.1 Point Concurrences

Each entry in the `point_concurrences` is a function on the t-element sets of points, for some positive integer t, giving the number of blocks containing each t-set. We use the general mechanism for `function_on_ksubsets_of_indices` with $k = t$, to do this. Note that a block design is t-wise balanced (see 8.3.5) if and only if the point concurrence function for $k = t$ takes only a single value.

For example, here is a small block design:

```
{
  "type" : "block_design",
  "id" : "v3-b5-r3-1",
  "v" : 3,
```

```
  "b" : 5,
  "blocks" : [
    [0], [2], [0, 1], [1, 2], [0, 1, 2]
  ]
}
```


and here are its t-wise point concurrences for $t = 1, 2$:


```
  "point_concurrences" : [
    {
      "domain_base" : "points",
      "n" : 3,
      "k" : 1,
      "ordered" : true,
      "title" : "replication_numbers",
      "maps" : [
        {
          "preimage" : ["entire_domain"],
          "image" : 3
        }
      ]
    },
    {
      "domain_base" : "points",
      "n" : 3,
      "k" : 2,
      "ordered" : true,
      "title" : "pairwise_point_concurrences",
      "maps" : [
        {
          "preimage" : [
            [0, 2]
          ],
          "image" : 1
        },
        {
          "preimage" : [
            [0, 1], [1, 2]
          ],
          "image" : 2
        }
      ]
    }
  ]
```


8.3.2 Block concurrences

Similarly, here we record the functions giving the numbers of points in the intersection of t-sets of blocks. The blocks are indexed from 0 to $b - 1$, and we again use the general mechanism for `function_on_ksubsets_of_indices`.

In practice, we almost always use the compressed representation of this function where we give only the preimage cardinalities (as described in section 5.2 on Functions and index flags).

For example, in the Fano plane, any block contains three points, and any two blocks meet in one point. This is recorded as follows:


```
"block_concurrences" : [
  {
    "domain_base" : "blocks",
    "n" : 7,
    "k" : 1,
    "ordered" : "unknown",
    "title" : "block_sizes",
    "maps" : [
      {
        "preimage_cardinality" : 7,
        "image" : 3
      }
    ]
  },
  {
    "domain_base" : "blocks",
    "n" : 7,
    "k" : 2,
    "ordered" : "unknown",
    "title" : "pairwise_block_intersection_sizes",
    "maps" : [
      {
        "preimage_cardinality" : 21,
        "image" : 1
      }
    ]
  }
]
```


8.3.3 *t*-design properties

(To be extended)

This is the area of greatest interest to combinatorialists.

Let t, v, k, λ be natural numbers with $t \leq k \leq v$ and $\lambda > 0$. A t-(v, k, λ) design is a block design with the properties

- there are v points;
- each block contains exactly k points;
- any t points are contained in exactly λ blocks.

A t-design is a block design which is a t-(v, k, λ) design for some v, k, λ.

If our design is a t-design for some $t > 1$, we record in the element `t_design_properties` the attributes t, v, b, r, k, λ. Here v and b have their usual meaning, r and k are the replication number and block size, and t and λ have the properties of the definition. We do not guarantee that the design is not a t'-design for some $t' > t$. (On the other hand, a t-design is also an s-design for any $s < t$.)

We also record some properties of the t-design. At present, we have the following:

`square`
: True if the numbers of points and blocks are equal.

`projective_plane`
: True if the design is a projective plane.

`affine_plane`
: True if the design is an affine plane.

`steiner_system`
: True if the design is a $t-(v, k, 1)$ design for some t, v, k. We also optioanlly record the relevant value of t (which may not be the same as the attribute called `t`).

`steiner_triple_system`
: True if the design is a $2-(v, 3, 1)$ design.

For example, the t-design properties of the Fano plane are as follows:

```
  "t_design_properties" : {
    "parameters" : {
      "t" : 2,
      "v" : 7,
      "b" : 7,
      "r" : 3,
      "k" : 3,
      "lambda" : 1
    },
    "square" : true,
    "projective_plane" : true,
    "affine_plane" : false,
    "steiner_system" : {
      "t" : 2
    },
    "steiner_triple_system" : true
  }
```

More properties will be included here. Among others, these will include different specific types of t-designs, and intersection triangles for Steiner systems.

8.3.4 α-resolvability

A resolution was defined above, but it can be described as a partition of the block multiset of the design into subdesigns, each of which is equireplicate with $r = 1$. More generally, an α-resolution is a partition of the design into subdesigns, each of which is equireplicate with $r = \alpha$.

The element `alpha_resolvable` is a mapping of `index_flags`, which record, for relevant positive values of α, whether the property is true, false or unknown. It is schematically represented as follows:

```
<alpha_resolvable> =  { <index_flag> *( , <index_flag> ) };

<index_flag> = "$integer" : ( $boolean | unknown ) ;
```

An example for a t-design, with parameters $t = 2$, $v = 9$, $k = 3$, $\lambda = 1$, displaying this attribute.

```
"alpha_resolvable" : {
  "1" : {
    "lambda" : 4
  },
  "2" : {
    "lambda" : 1
  },
  "4" : true
}
```

From the above snippet we can deduce that the above design contains atleast a resolution, since it is α-resolvable for r=1, and that is α-resolvable, for r=2 and r=4.

8.3.5 t-wise balance

A block design is *t-wise balanced* if each set of t distinct points is contained in a constant number of blocks; it does not imply constant block size. (The two properties together specify a t-design.) Unlike for t-designs, a block design may be t-wise balanced but not s-wise balanced for $s < t$. We store information about the values of t for which the design is t-wise balanced as mapping of `t_index_flags`.

Here is an example of the `t_wise_balanced` element for the Fano plane:

```
"t_wise_balanced" : {
  "1" : {
    "lambda" : 3
  },
  "2" : {
    "lambda" : 1
  }
}
```

t-wise balanced is schematically described as:

```
<t_wise_balanced> =  { <t_index_flag> *( , <t_index_flag> ) } ;

<t_index_flag> = "$integer" : ( $boolean | unknown | { lambda: $integer } ) ;
```

8.4 Automorphisms

An *automorphism* of a block design is a permutation of the set of points of the design such that, if this permutation is applied to the elements of each block, the multiset of blocks is the same as before. (In other words: the block multiset is a list of lists; if we apply the permutation to all elements of the inner lists, re-sort each inner list, and then re-sort the outer list, the result is the same as the original list.)

The collection of all automorphisms forms a *group*, that is, it is closed under composition of permutations. Thus, the automorphism group of a design is a permutation group on the set of points.

If the block design does not have repeated blocks, then each automorphism induces a permutation on the set $[0, \ldots, b-1]$ of block indices: this permutation carries i to j if the image of the i-th block under the automorphism is the j-th block. In this case, the automorphism group has an induced action on the set of block indices. If there are repeated blocks, the action on the set of block indices is undefined.

For example, the example in the Introduction has an automorphism $[1, 3, 5, 2, 0, 6, 4]$ (mapping 0 to 1, 1 to 3, etc.) Altogether this famous design has 168 automorphisms.

The specifications for automorphism groups and their properties for block designs are:

```
<block_design_automorphism_group> = automorphism_group: {
    <permutation_group>,
    <block_design_automorphism_group_properties>
};

<permutation_group> = permutation_group : {
         degree : $integer ,
         order  : $integer ,
         domain : points ,
         <generators>
    ?( , <permutation_group_properties> )
};

<block_design_automorphism_group_properties> =
    automorphism_group_properties: {
        block_primitive           : ( $boolean | not_applicable ) ,
        degree_block_transitivity : ( $integer | not_applicable ) ,
        no_block_orbits           : ( $integer | not_applicable )
};
```

Permutation groups and their properties have already been described in section 6. Some properties of the automorphism group are specific to block designs, and are (optionally) described separately under `automorphism_group_properties`. They are:

`block_primitive`
: True if the group acts primitively on blocks. (If there are repeated blocks, this is not defined, and takes the value `not_applicable`.)

`no_block_orbits`
: The number of orbits on blocks. (If there are repeated blocks, this is not defined, and takes the value `not_applicable`.)

`degree_block_transitivity`
: The maximum number s such that the group is s-transitive on blocks. (If there are repeated blocks, this is not defined, and takes the value `not_applicable`.)

8.5 Resolutions

Recall that a *resolution* of a block design is a partition of the blocks into subsets, each of which forms a partition of the point set. Such a partition of the block (multi)set can be represented as a function on the set of indices of blocks (the parts of the partition being the preimages of the elements in the range of the function). We thus store a resolution as a `function_on_indices` with `domain="blocks"`.

An *automorphism* of a resolution is a permutation of the set of points of the design such that, if this permutation is applied to the elements of each block in each resolution class, the (multi)set of resolution classes is the same as before. The collection of all automorphisms of a resolution of a design forms a subgroup of the automorphism group of the design itself, and we use the same `automorphism_group` structure for the automorphism group of a resolution as we do for the automorphism group of a block design (although the `automorphism_group_properties` for a resolution are different than those for a block design).

We specify a resolution as follows:

```
<resolution> =  {
          function_on_indices: <function_on_indices>
    ?( , <resolution_automorphism_group> )
};
```

A block design D may have more than one resolution. We say that two resolutions R and S of D are *isomorphic* if there is an element g in the automorphism group of D, such that, when g is applied to the elements of each block in each resolution class of R, the resulting resolution is equal to S. Isomorphism defines an equivalence relation on the set of resolutions of D.

We use the element `resolutions` to store a nonempty list of (distinct) resolutions of a resolvable design. The attributes of this entry are used to specify whether the listed resolutions are pairwise nonisomorphic and whether all isomorphism classes of resolutions are represented in the list.

```
<resolutions> = resolutions : {
    pairwise_nonisomorphic  : ( $boolean | unknown ) ,
    all_classes_represented : ( $boolean | unknown ) ,
    value                   : [ <resolution> *( , <resolution> ) ]
};
```

We now display a famous resolvable design, the affine plane of order 3, which has just one resolution.


```
{
  "external_representation_version" : "3.0",
  "design_type" : "block_design",
  "number_of_designs" : 1,
  "invariants" : {
    "relations" : [
      ["t", "=", 2],
      ["v", "=", 9],
      ["b", "=", 12],
      ["r", "=", 4],
      ["k", "=", 3],
      ["lambda", "=", 1]
    ]
  },
  "pairwise_nonisomorphic" : true,
  "complete_upto_isomorphism" : true,
  "designs" : [
    {
      "type" : "block_design",
      "id" : "t2-v9-b12-r4-k3-L1-0",
      "v" : 9,
      "b" : 12,
      "blocks" : [
        [0, 1, 2], [0, 3, 4], [0, 5, 6], [0, 7, 8], [1, 3, 5], [1, 4, 7],
        [1, 6, 8], [2, 3, 8], [2, 4, 6], [2, 5, 7], [3, 6, 7], [4, 5, 8]
      ],
      "resolutions" : {
        "pairwise_nonisomorphic" : true,
        "all_classes_represented" : true,
        "value" : [
          {
            "function_on_indices" : {
              "domain" : "blocks",
              "n" : 12,
              "ordered" : true,
              "title" : "resolution",
              "maps" : [
                {
                  "preimage" : [0, 10, 11],
                  "image" : 0
                },
                {
                  "preimage" : [1, 6, 9],
                  "image" : 1
                },
                {
                  "preimage" : [2, 5, 7],
                  "image" : 2
                },
```

```
            {
              "preimage" : [3, 4, 8],
              "image" : 3
            }
          ]
        },
        "automorphism_group" : {
          "permutation_group" : {
            "degree" : 9,
            "order" : 432,
            "domain" : "points",
            "generators" : [
              [0, 2, 1, 4, 3, 6, 5, 8, 7], [2, 1, 0, 5, 7, 3, 8, 4, 6],
              [3, 6, 7, 4, 0, 2, 8, 5, 1], [1, 0, 2, 3, 5, 4, 7, 6, 8],
              [0, 2, 1, 3, 4, 8, 7, 6, 5], [0, 3, 4, 1, 2, 5, 6, 8, 7],
              [2, 1, 0, 3, 8, 5, 7, 6, 4], [0, 2, 1, 3, 4, 8, 7, 6, 5]
            ]
          }
        }
      }
    ]
  }
}
```


8.6 Statistical Properties

For a statistician, a block design is a plan for an experiment. The v points of the block design are usually called *treatments,* a general terminology encompassing any set of v distinct experimental conditions of interest. The purpose of the experiment is to compare the treatments in terms of the magnitudes of change they induce in a response variable, call it y. These magnitudes are called *treatment effects.*

In a typical experiment (there are *many* variations on this, but we stick to the basics to start), each treatment is employed for the same number r of experimental runs. Each *run* is the application of the treatment to an individual *experimental unit* (also called *plot*) followed by the observation of the response y. An experiment to compare v treatments using r runs (or "replicates") requires a total of vr experimental units.

If the vr experimental units are *homogeneous* (for the purposes of the experiment, essentially undifferentiable) then the assignment of the v treatments, each to r units, is made completely at random. Upon completion of the experiment, differences in treatment effects are assessed via differences in the v means of the observed values y for the v treatments (each mean is the average of r observations). This simplest of experiments is said to follow a *completely randomized design* - it is *not* a block design.

The concept of a *blocked* experiment comes into play when the vr experimental units are *not* homogeneous. A *block* is a subset of the experimental units which are essentially undifferentiable from one another, but which share a common characteristic thought to affect the response and which

differentiates them from other experimental units. If we can partition our vr heterogeneous units into b such sets (blocks) of k homogeneous units each, then after completion of the experiment, when the statistical analysis of results is performed, we are able to separate the variability in response due to this systematic unit heterogeneity from that induced by differences in the treatments, thereby increasing the precision of treatment comparisons.

To make clear the essential issue here, consider a simple example. We have $v = 3$ fertilizer cocktails (the treatments) and will compare them in a preliminary greenhouse experiment employing $vr = 6$ potted tobacco plants (the experimental units). If the pots are identically prepared with a common soil source and each receiving a single plant from the same seed set and of similar size and age, then we deem the units homogeneous. Simply randomly choose two pots for the application of each cocktail. This is a completely randomized design. At the end of the experimental period (two months, say) we measure $y =$ the total biomass per pot.

Now suppose three of the plants are clearly larger than the remaining three. The statistically "good" design is also the intuitively appealing one: make separate random assignments of the three cocktails to the three larger plants, and to the three smaller plants, so that each cocktail is used once with a plant of each size. We have blocked (by size) the 6 units into two homogeneous sets of 3 units each, then randomly assigned treatments within blocks. Notice that there are $3!\times3!=36$ possible assignments here; above there were $6!=720$ possible assignments. The purpose of the blocking is to allow variabilty due to initial size to be removed from treatment comparisons in the statistical analysis. Because $k = v$ this is called a *complete* block design.

The statistical use of the term "block design" should now be clear: a block design is a plan for an experiment in which the experimental units have been partitioned into homogeneous sets, telling us which treatment each experimental unit receives. The external representation is a bit less specific: each block of a block design in external representation format tells us a set of treatments to use on a homogeneous set (block) of experimental units but without specifying the exact treatment-to-unit map within the block. The latter is usually left to random assignment, and moreover, does not affect the standard measures of "goodness" of a design (does not affect the information matrix; see below), so will not be mentioned again.

There are solid mathematical justifications for why the complete block design in the example above is deemed "good," which we develop next. This development does not require that $k = v$, nor that the block sizes are all the same, nor that each treatment is assigned to the same number of units. However, it does assume that the block sizes are known, fixed constants, as determined by the collection (of fixed size) of experimental units at hand. Given the division of units into blocks, we seek an assignment of treatments to units, i.e. a block design, that optimizes the precision of our estimates for treatment effects. From this perspective, two different designs are comparable if and only if they have the same v, b, and block sizes (more precisely, block size distribution).

Statistical estimation takes place in the context of a model for the observations y. Let y_{ij} denote the observation on unit i in block j. Of course we must decide what treatment is to be placed on that unit - this is the design decision. Denote the assigned treatment by $d[i, j]$. Then the standard statistical model for the block design (there are many variations, but here this fundamental, widely applicable block design model is the only one considered) is

$$y_{ij} = \mu + \tau_{d[i,j]} + \beta_j + e_{ij}$$

where τ is the treatment effect mentioned earlier, β_j is the effect of the block (reflecting how this homogeneous set of units differs from other sets), μ is an average response (the treatment and block effects may be thought of as deviations from this average), and e_{ij} is a random error term reflecting variability among homogeneous units, measurement error, and indeed whatever forces that play a role in making no experimental run perfectly repeatable. In this model the e_{ij}'s have independent probability distributions with common mean 0 and common (unknown) variance σ^2.

With n the total number of experimental units in a block design, the design map d (note: symbol d is used both for the map and the block design itself) from plots to treatments can be represented as an $n \times v$ incidence matrix, denoted A_d. Also let N_d be the $v \times b$ treatment/block incidence matrix, let K be the diagonal matrix of block sizes ($= kI$ for equisized blocks), and write

$$C_d = A_d' A_d - N_d K^{-1} N_d'$$

which is called the *information matrix* for design d (note: A' denotes the transpose of a matrix A). Why this name? Estimation focuses on comparing the treatment effects: every *treatment contrast* $\sum c_i \tau_i$ with $\sum c_i = 0$ is of possible interest. All contrasts are *estimable* (can be linearly and unbiasedly estimated) if and only if the block design is connected. For disconnected designs, all contrasts within the connected treatment subsets span the space of all estimable contrasts. For a given design d, we employ the *best* (minimum variance) linear unbiased estimators for contrasts. The variances of these estimators, and their covariances, though best for given d, are a function of d. In fact, if c is the vector of contrast coefficients c_i then the variance of contrast $c'\tau = \sum c_i \tau_i$ is

$$\sigma^2 c' C_d^+ c$$

where C_d^+ is the Moore-Penrose inverse of C_d (if $C_d = \sum x_{di} E_{di}$ is the spectral decomposition of C_d, then $C_d^+ = \sum_{x_{di} \neq 0} \frac{1}{x_{di}} E_{di}$). The information carried by C_d is the precision of our estimators: large information C_d corresponds to small variances as determined by C_d^+.

We wish to make variances small through choice of d. That is, we choose d so that C_d^+ is (in some sense) small. *Design optimality criteria* are real-valued functions of C_d^+ that it is desirable to minimize. Obviously a design criterion may also be thought of as a function of d itself, which we do when convenient.

With this background, let us turn now to what has been implemented for the external representation of `statistical_properties`:

```
<statistical_properties> = statistical_properties: {
          precision: $integer
    ?( ,  <canonical_variances> )
    ?( ,  <pairwise_variances> )
    ?( ,  <optimality_criteria> )
    ?( ,  <other_ordering_criteria> )
    ?( ,  <canonical_efficiency_factors> )
    ?( ,  <functions_of_efficiency_factors> )
    ?( ,  <robustness_properties> )
};
```

The elements of `statistical_properties` are quantities which can be calculated starting from the information matrix C_d.

8.6.1 Canonical variances

The $v \times v$ symmetric, nonnegative definite matrix C_d is never of full rank; its maximal rank is $v-1$, which is achieved exactly when the block design d is connected. Denote the $v-1$ ordered, largest eigenvalues of C_d by

$$x_{d1} \leq x_{d2} \leq \cdots \leq x_{d,v-1}$$

Design d is connected if and only if $x_{d1} > 0$. The corresponding nonzero eigenvalues of C_d^+ are the inverses of the nonzero x_{di}'s ; for a connected design these are

$$z_{d1} \geq z_{d2} \geq \cdots \geq z_{d,v-1}$$

The z_{di} are called the *canonical variances*. They are the variances, aside from the constant σ^2, of a set of contrasts whose vectors of coefficients are any orthonormal set of eigenvectors of C_d orthogonal to the all-ones vector (called *standard contrasts*). We define a full set of $v-1$ canonical variances even for disconnected designs, in which case some of the z_{di} are taken as infinity. An infinite canonical variance corresponds to a contrast which is not estimable.

Many of the commonly used design optimality criteria are based on the canonical variances. Because of their importance they have merited an element, `canonical_variances`, in the external representation. Infinite values are recorded there as "not_applicable" and, as already explained, correspond to zero values of x_{di}'s.

8.6.2 Pairwise variances

In statistical practice, some experiments focus on comparing the effect of each treatment to each other treatment; these are the *elementary contrasts* $\tau_i - \tau_{i'}$. The variances $v_{dii'}$ of the elementary contrasts for a connected design d, aside from the constant σ^2, are

$$v_{dii'} = c^+_{dii} + c^+_{di'i'} - 2c^+_{dii'}$$

for $1 \leq i < i' \leq v$, where $c^+_{dii'}$ is the general element of C_d^+ . Several optimality criteria are based on the $v(v-1)/2$ numbers $v_{dii'}$, called *pairwise variances*. Moreover, partial balance properties are reflected in the $v_{dii'}$. For these reasons, `pairwise_variances` is also an element in the external representation. For disconnected designs some elementary contrasts are not estimable; in the external representation, the corresponding values $v_{dii'}$ are recorded as "not_applicable."

8.6.3 Optimality criteria

We are now in a position to define the design `optimality_criteria` that have been implemented.

`phi_0`

$\Phi_0 = \sum log(z_{di})$

This is the log of the product of the canonical variances, called the D-criterion (for "determinant"). The product is proportional to the volume of the confidence ellipsoid for joint estimation of the standard contrasts.

`phi_1`

$\Phi_1 = \sum z_{di}/(v-1)$

This is the arithmetic mean of the canonical variances, called the A-criterion (for "average"). It is also proportional to the average of the $v(v-1)/2$ pairwise variances $v_{dii'}$.

`phi_2`

$\Phi_2 = \sum z_{di}^2/(v-1)$

This is the mean of the squared canonical variances. For any fixed value of Φ_1 this is minimized when the z_{di} are as close as possible in the square error sense. Thus it is a measure of *balance* of the design. A design is said to be *variance balanced* when all normalized treatment contrasts are estimated with the same variance. This occurs if and only if all the z_{di} are equal, which gives the smallest conceivable (and often unattainable) value for Φ_2 for fixed Φ_1 . Among binary, equiblocksize designs, only balanced incomplete block designs achieve equality of the z_{di}.

`maximum_pairwise_variances`

The largest pairwise variance ($\max(v_{dii'})$), called the MV-criterion (for "maximum variance"). This is a minimax criterion: minimize the maximum loss (as measured by variance) for estimating the elementary contrasts.

`E_criteria`

$z_{d1} + z_{d2} + \ldots + z_{di}$

The sum of the i largest canonical variances, called the E_i criterion. E_1 is usually called "the" E-criterion; minimization of E_1 is minimization of the worst variance over all possible normalized treatment contrasts. E_1 is the counterpart of `maximum_pairwise_variances` for the set of all contrasts. More generally, minimization of E_i is minimization of the sum of the i worst variances over all possible sets of i normalized orthogonal treatment contrasts. Thus the E_i are a family of minimax criteria. E_{v-1} is equivalent to Φ_1. A design which minimizes all of the E_i for $i = 1, \ldots, v-1$ is *Schur-optimal* (it minimizes all Schur-convex functions of the canonical variances).

8.6.4 Other ordering criteria

In addition to the optimality criteria just listed, we also implement several ordering criteria for block designs (optimality criteria are ordering criteria that meet conditions described fully in a later subsection).

`no_distinct_canonical_variances`
: The number of distinct z_{di}. For balanced incomplete block designs this value is 1. A balance criterion; the fewer variances a design produces, the easier are the results to understand.

`max_min_ratio_canonical_variances`
: The ratio of largest to smallest canonical variance ($z_{d1}/z_{d,v-1}$), called the *canonical variance ratio.* Again, the value for a balanced incomplete block design is 1. Values close to one correspond to variances that are quite similar.

`no_distinct_pairwise_variances`
: The number of distinct $v_{dii'}$. Analogous to `no_distinct_canonical_variances`, but for pairwise variances rather than canonical variances.

`element max_min_ratio_pairwise_variances`
: The ratio of largest to smallest pairwise variance ($\max(v_{dii'})/\min(v_{dii'})$), called the *pairwise variance ratio.* Analogous to `max_min_ratio_canonical_variances`, but for pairwise variances rather than canonical variances.

`trace_of_square`
: $\sum z_{di}^{-2} = \sum x_{di}^2$.
The trace of the square of C_d. This is called the S-criterion. Typically invoked as part of an (M,S)-optimality argument (minimize S subject to maximizing the trace of C_d). No direct statistical interpretation, though usually leads to reasonably "good" designs.

It was mentioned above that a complete block design (each block size is v and each treatment is assigned to one unit in each block) is a "good" design. Now we state why. Over all possible assignments of v treatments to b blocks of size v, a complete block design minimizes *all* of the criteria defined above (save for $\mathrm{tr}(C_d^2)$, which it minimizes subject to the mean of the unsquared components). The same statement holds for a balanced incomplete block design for constant block size less than v (whenever a BIBD exists). Otherwise, the optimal block design problem can be quite tricky, with such uniform optimality hard to come by.

An `optimality_value` for any of the optimality criteria above has three elements: its numerical `value` and two associated numbers `absolute_efficiency` and `calculated_efficiency` (for `other_ordering_criteria`, the same concepts are implemented under the names `absolute_comparison` and `calculated_comparison` so are not separately discussed here - see the later subsection on design orderings). Given any two designs, d_1 and d_2 say, they can be compared on any of the listed optimality criteria. The *relative efficiency* of design d_2 with respect to criterion Φ , compared to design d_1, is $\Phi(d_1)/\Phi(d_2)$. If d_1 is in fact an *optimal* design as measured by Φ (d_1 minimizes $\Phi(d)$ over all d), then the relative efficiency of any d compared to d_1 is the `absolute_efficiency` of d. Both of these efficiencies are between 0 and 1, with smaller criterion values corresponding to larger efficiencies; the absolute efficiency of an optimal design is 1.

The concept of absolute efficiency depends on what is meant by the phrase "all d". It has already been explained that comparisons are for designs with the same v, b, and block sizes. In the external representation, an `absolute_efficiency` is for the class of *all binary designs* with the same v, b, and block size distribution, called the *reference universe.* When the minimum criterion value over the reference universe is not known, `absolute_efficiency` takes the value "unknown." For a

disconnected design `absolute_efficiency` takes the value "0" regardless of whether the optimal value is known or not. It happens, only rarely, that a smaller value of a criterion can be found for a nonbinary design with the same v, b, and block sizes, in which case the `absolute_efficiency` of the nonbinary design will be greater than 1. Nonbinary designs are not at present considered in the external representation. Relative efficiencies when the best value over the reference universe is not known, or within a subclass of the reference universe, can be calculated on a case-by-case basis; in external representation terminology, this is a `calculated_efficiency`. For instance, one may wish to compare only resolvable designs. `calculated_efficiency` takes the value "0" for all disconnected designs.

8.6.5 Efficiency factors

There is another set of values, the `canonical_efficiency_factors`, that are used to evaluate a design but which has not yet been discussed. Let r_i be the number of units receiving treatment i (this is the general diagonal element of $A_d'A_d$) and let R be the diagonal matrix with the $\sqrt{r_i}$ along the diagonal. The *canonical efficiency factors*

$$e_{d1} \leq e_{d2} \leq \cdots \leq e_{d,v-1}$$

for design d are the $v-1$ largest eigenvalues of $F_d = R^{-1}C_dR^{-1}$. The remaining eigenvalue of F_d is 0.

In the incomplete block design, the variance of the estimator of $x'\tau$ is equal to $x'C_d^- x\sigma^2_{\text{IBD}}$, while the variance in a completely randomized design with the same replication is $x'R^{-2}x\sigma^2_{\text{CRD}}$, where the two values of σ^2 are the variances per plot in the incomplete block design and the completely randomized design respectively. Therefore the relative efficiency of the IBD to this CRD is

$$\frac{x'R^{-2}x}{x'C_d^- x} \times \frac{\sigma^2_{\text{CRD}}}{\sigma^2_{\text{IBD}}}$$

The first part of this, which depends on the design but not on the values of the plot variances, is called the *efficiency factor* for the contrast $x'\tau$. Put $R^{-1}x = u$. Then the efficiency factor for $x'\tau$ can be expressed as

$$\frac{u'u}{u'F_d^- u},$$

which, if u is an eigenvector of F with eigenvalue ε, is equal to ε.

Since F_d is symmetric, it can orthogonally diagonalized. The contrast $x'\tau$ is called a *basic contrast* if $x = Ru$ for an eigenvector u of F_d which is not a multiple of Ru_0, where u_0 is the all-1 vector. Thus the canonical efficiency factors are the efficiency factors of the basic contrasts. The basic contrasts span the space of all treatment contrasts; moreover, if u_1 is orthogonal to u_2 then the estimators of $(Ru_1)'\tau$ and $(Ru_2)'\tau$ are uncorrelated (and independent if the errors are normally distributed).

Each efficiency factor lies between 0 and 1; at the extremes are contrasts that cannot be estimated (efficiency factor = 0) and contrasts that are estimated just as well as in an unblocked design with the same σ^2 (efficiency factor = 1). Thus $1 - e_{di}$ is the proportion of information lost to

blocking when estimating a corresponding basic contrast (or any contrast in its eigenspace); e_{di} is the proportion of information retained. Design d is disconnected if and only if $e_{d1} = 0$.

The comparison to a completely randomized design *with the same replication numbers* is the key concept here. Efficiency factors evaluate design d over the universe of all designs with the same replications $r_1, \ldots, r_v$ as d, constraining the earlier discussed reference universe of competitors with the given v and block size distribution. This constrained universe of comparison is typically justified as follows: the replication numbers have been purposefully chosen (and thus fixed) to reflect relative interest in the treatments, or the replication numbers are forced by the availablity of the material (for example, scarce amounts of seed of new varieties but plenty of the control varieties), so the task is to determine a best (in whatever sense) design within those constraints. The idealized best (in every sense) is the completely randomized design (no blocking) *so long as this does not increase the variance per plot*. Though experimental material at hand has forced blocking, the unobtainable CRD can still be used as a fixed standard for comparison.

Variances of contrasts estimated with a CRD exactly mirror the selected sample sizes. If the replication numbers are intended to reflect relative interest in treatments, then a reasonable design goal is to find d for which variances of all contrast estimators enjoy the same relative magnitudes as in the CRD. This is exactly the property of *efficiency balance*: design d is *efficiency balanced* if its canonical efficiency factors are all equal: $e_{d1} = e_{d2} = \ldots = e_{d,v-1}$.

For equal block sizes k ($< v$), the only equireplicate, binary, efficiency balanced designs are the BIBDs. Unfortunately, an unequally replicated design cannot be efficiency balanced if the block sizes are constant and it is binary. Thus in many instances the best hope is to approximate the relative interest intended by the choice of sample sizes. Approximating efficiency balance (seeking small dispersion in the efficiency factors) will then be a design goal, typically in conjunction with seeking a high overall efficiency factor as measured through one or more summary functions of the canonical efficiency factors. The harmonic mean of the canonical efficiency factors (see below) is often called "the" efficiency factor of a design; if the value is 0.87, for instance, then use of blocks has resulted in an overall 13% loss of information.

For an equireplicate design (all r_i are equal—to r say) the canonical efficiency factors are just $1/r$ times the inverses of the canonical variances; some statisticians consider them a more interpretable alternative to the canonical variances in this case. If all the efficiency factors are 1, the design is *fully efficient*, a property achieved in the equiblocksize case (with $k \leq v$) only by complete block designs. Consequently, efficiency factors for equireplicate designs can also be interpreted as summarizing the loss of information when using incomplete blocks (block sizes smaller than v) rather than complete blocks.

The external representation contains the following commonly used `summaries_of_efficiency_factors`. In terms of these measures, an optimal design is one which *maximizes* the value. Each summary measure induces a design ordering which is identical to that for one of the `optimality_criteria` above, based on the canonical variances, *provided* the set of competing designs is restricted to be equireplicate. More generally, these measures should only be used to compare designs with the same replication numbers.

`harmonic_mean`
: $(v-1)/\sum(1/e_{di})$

This is the harmonic mean of the efficiency factors. Equivalent to (produces the same design ordering as) Φ_1 in the equireplicate case.

`geometric_mean`

$exp(\sum log(e_{di})/(v-1))$

This is the geometric mean of the efficiency factors. Equivalent to (produces the same design ordering as) Φ_0 in the equireplicate case.

`minimum`

The smallest efficiency factor (e_{d1}). Equivalent to E_1 in the equireplicate case.

The Introduction gives an example of a block design which is called the Fano plane. It is a BIBD for 7 treatments in 7 blocks of size 3. As with any BIBD, it is `pairwise_balanced`, `variance_balanced`, and `efficiency_balanced`, and it is optimal with respect to all of the `optimality_criteria` over its entire reference universe. Here are all of the `statistical_properties`, that have been discussed so far, for this example:


```
  "statistical_properties" : {
    "precision" : 9,
    "canonical_variances" : {
      "no_distinct" : 1,
      "ordered" : true,
      "value" : [
        {
          "multiplicity" : 6,
          "canonical_variance" : 0.428571429
        }
      ]
    },
    "pairwise_variances" : {
      "domain_base" : "points",
      "n" : 7,
      "k" : 2,
      "ordered" : true,
      "maps" : [
        {
          "preimage" : ["entire_domain"],
          "image" : 0.857142857
        }
      ]
    },
    "optimality_criteria" : {
      "phi_0" : {
        "value" : -5.08378716,
        "absolute_efficiency" : 1,
        "calculated_efficiency" : 1
      },
      "phi_1" : {
        "value" : 0.428571429,
        "absolute_efficiency" : 1,
```

```
      "calculated_efficiency" : 1
    },
    "phi_2" : {
      "value" : 0.183673469,
      "absolute_efficiency" : 1,
      "calculated_efficiency" : 1
    },
    "maximum_pairwise_variances" : {
      "value" : 0.857142857,
      "absolute_efficiency" : 1,
      "calculated_efficiency" : 1
    },
    "E_criteria" : {
      "1" : {
        "value" : 0.428571429,
        "absolute_efficiency" : 1,
        "calculated_efficiency" : 1
      },
      "2" : {
        "value" : 0.857142857,
        "absolute_efficiency" : 1,
        "calculated_efficiency" : 1
      },
      "3" : {
        "value" : 1.28571429,
        "absolute_efficiency" : 1,
        "calculated_efficiency" : 1
      },
      "4" : {
        "value" : 1.71428571,
        "absolute_efficiency" : 1,
        "calculated_efficiency" : 1
      },
      "5" : {
        "value" : 2.14285714,
        "absolute_efficiency" : 1,
        "calculated_efficiency" : 1
      },
      "6" : {
        "value" : 2.57142857,
        "absolute_efficiency" : 1,
        "calculated_efficiency" : 1
      }
    }
  },
  "other_ordering_criteria" : {
    "trace_of_square_of_C" : {
      "value" : 32.6666667,
      "absolute_comparison" : 1,
      "calculated_comparison" : 1
    },
```

```
      "max_min_ratio_canonical_variances" : {
        "value" : 1.0,
        "absolute_comparison" : 1,
        "calculated_comparison" : 1
      },
      "max_min_ratio_pairwise_variances" : {
        "value" : 1.0,
        "absolute_comparison" : 1,
        "calculated_comparison" : 1
      },
      "no_distinct_canonical_variances" : {
        "value" : 1,
        "absolute_comparison" : 1,
        "calculated_comparison" : 1
      },
      "no_distinct_pairwise_variances" : {
        "value" : 1,
        "absolute_comparison" : 1,
        "calculated_comparison" : 1
      }
    },
    "canonical_efficiency_factors" : {
      "no_distinct" : 1,
      "ordered" : true,
      "value" : [
        {
          "multiplicity" : 6,
          "canonical_efficiency_factor" : 0.777777778
        }
      ]
    },
    "functions_of_efficiency_factors" : {
      "harmonic_mean" : 0.777777778,
      "geometric_mean" : 0.777777778,
      "minimum" : 0.777777778
    }
  }
```


8.6.6 Robustness properties

Experiments do not always run successfully on all experimental units. In the fertilizer/tobacco example above, if midway through the growth period one of the pots is accidentally broken, then one experimental unit has been "lost." One is effectively left with a *different* block design, with different properties than the one initiated.

The concept of *robustness* of a block design is here considered as its ability to maintain desirable statistical properties under loss of individual plots or entire blocks. Such a loss is catastrophic if the design becomes disconnected. Less than catastrophic but of genuine concern are losses in the information provided by the design, as measured by various optimality criteria. The two elements of `robustness_properties` accommodate these two perspectives.

The element `robust_connected` makes the statement *The design is connected under all possible ways in which* `number_lost` *of* `category_lost` *can be removed.* If the reported value of `number_lost` is known to be the largest integer for which this statement is true then `is_max` takes the value "true" and otherwise takes the value "unknown" (the value "false" is not allowed).

The element `robust_efficiencies` reports A, E, D, and MV efficiencies for a given number (`number_lost`) of plots or blocks (`category_lost`) removed from the design. The efficiencies can be calculated from two different perspectives. If `loss_measure` ="average" then the criterion value used is the average of all its values over all possible deletions of the type and number prescribed. If `loss_measure` ="worst" then the criterion value used is the maximum of all its values over all possible deletions of the type and number prescribed.

Balance measures have not been incorporated under `robust_efficiencies`. This is because designed balance is typically severely affected by plot/block loss and in ways that need have no relation to treatment structure.

The calculations associated with the values reported here can be quite expensive.

8.6.7 Computational details

As has already been explained, the elements of `statistical_properties` are quantities which can be calculated starting from the information matrix C_d. There are three fundamental calculations: the canonical variances, the pairwise variances, and the canonical efficiency factors.

The canonical variances are the inverses of the eigenvalues of C_d, eigenvalues of zero corresponding to canonical variances of ∞. Thus we need the roots of the polynomial $|C_d - xI| = 0$. As C_d is a rational matrix, this polynomial admits a factorization into irreducible factors over the rational field. Thus, in theory, the multiplicities of the canonical variances can be determined exactly, even if some of the values themselves are irrational. If the eigenvalues of C_d are numerically extracted directly without factoring the characteristic polynomial, then the problem of inexact counts of those eigenvalues can arise.

Pairwise variances are defined above in terms of the Moore-Penrose inverse C_d^+ of C_d: $v_{dii'} = c^+_{dii} + c^+_{di'i'} - 2c^+_{dii'}$. In fact, *any* generalized inverse C_d^- of C_d can be used, from which $v_{dii'} = c^-_{dii} + c^-_{di'i'} - 2c^-_{dii'}$. Let J be an all-ones matrix. If d is connected, then $C_d + aJ$ is invertible for any $a \neq 0$ and $C_d^- = (C_d + aJ)^{-1}$ is a generalized inverse of C_d (the same operation can be carried out for the connected components of C_d if d is disconnected). Thus pairwise variances can be calculated by inversion of a rational, nonsingular matrix.

Efficiency factors are defined as eigenvalues of the matrix $F_d = R^{-1}C_dR^{-1}$, which can certainly be irrational. Extracting the roots of $N_dK^{-1}N_d'$ with respect to R^2, that is, solving the equation $|N_dK^{-1}N_d' - \mu R^2| = 0$, produces values μ_{di} for $i = 1, \ldots, v$ satisfying $\mu_{dv} = 0$ and otherwise $\mu_{di} = 1 - e_{di}$. Thus efficiency factors can be found by extracting roots of a symmetric, rational matrix, involving the same computational issues as for the canonical variances.

The number of infinite canonical variances equals the number of connected components of d less 1 (this being zero for any connected designs). Numerical extraction of eigenvalues of C_d can potentially produce, at a given level of precision, values indistinguishable from zero that are in actuality positive, consequently producing an erroneous number of infinite canonical variances. This approximation error is prohibited by cross-checking against the `connected` indicator.

8.6.8 Design orderings based on the information matrix

The external representation implements optimality criteria and other ordering criteria as aids in judging statistical properties of of members of a class of block designs. Definitions and motivating principles for these two classes of criteria are given here.

Denote by $\mathcal{C}$ the class of information matrices for the class of designs $\mathcal{D}$ under consideration, that is,

$$\mathcal{C} = \{C_d : d \in \mathcal{D}\}.$$

If g map elements of $\mathcal{C}$ to a subset of the reals plus ∞, then g provides an ordering on d:

$$d_1 \geq_g d_2 \iff g(C_{d1}) \leq g(C_{d2})$$

Usually $\mathcal{D}$ is our reference universe, but need not be so. In any case $\mathcal{D}$ is finite and $g(C_d) = \infty$ if and only if d is disconnected.

While it is trivial to define ordering functions g, what does it mean for a function $g : \mathcal{C} \to \mathcal{R}$ to be an *optimality criterion*? Any ordering of information matrices could be allowed, but not all orderings reflect a reasonable statistical concept of optimality. We work here towards appropriate definitions.

The first fundamental consideration is that of relative interest in the v members of the treatment set. Let $\mathcal{P}$ be the class of $v \times v$ permutation matrices. If treatments are of equal interest, then order g should satisfy the symmetry condition

$$g(C_d) = g(PC_dP') \text{ for every } P \in \mathcal{P}.$$

Only g satisfying this condition are considered here.

Another fundamental principle arises from the nonnegative definite ordering on information matrices:

$$C_{d_1} \geq_{nnd} C_{d_2} \iff C_{d_1} - C_{d_2} \text{ is nonegative definite}$$

Now $C_{d_1} \geq_{nnd} C_{d_2} \iff C^+_{d_2} \geq_{nnd} C^+_{d_1}$ so this ordering says $\text{var}_{d_1}(\widehat{l'\tau}) \leq \text{var}_{d_2}(\widehat{l'\tau})$ for *every* contrast $l'\tau$. A reasonable restriction to place on an optimality criterion g is that it respect the nonnegative definite ordering:

$$C^+_{d_2} \geq_{nnd} C^+_{d_1} \Rightarrow g(C_{d_1}) \leq g(C_{d_2})$$

Fact: In the reference universe of all binary block designs with v treatments and fixed block size distribution, $C^+_{d_2} \geq_{nnd} C^+_{d_1} \iff C_{d_1} = C_{d2}$

Proof: The trace $\text{tr}(C_d)$ is fixed for all d in the reference universe. Consequently $C_{d_1} \geq_{nnd} C_{d_2}$ says that $C_{d_1} - C_{d_2}$ is a nonnegative definite matrix with zero trace, that is, it is the zero matrix.

Thus the nonnegative definite ordering does not distinguish among ordering functions g for the reference universe. While the external representation does not currently include nonbinary designs, we take as part of our definition that an optimality criterion g must respect the nonnegative definite ordering; effectively, it must be able to make this fundamental distinction in the larger class of *all*

designs with the same v and block size distribution. A criterion that cannot do this has little (if any) capacity to detect inflated variances.

Typically one wishes to consider not arbitrary functions on the matrices C_d, but functions of some characteristic(s) of those matrices. Of particular interest are the lists of canonical variances and pairwise variances. A criterion which is a function of a list of values should respect orderings of lists, as follows. A list L_d of s real values calculated from C_d may be thought of as the uniform probability distribution $p(l) = \frac{1}{s}$ for each $l \in L_d$. Probability distributions may be stochastically ordered: the distribution of X is stochastically larger than that of Y, written $X \geq_s Y$, if $\Pr(X \leq a) \leq \Pr(Y \leq a)$ for every a. Thus define L_{d_2} to be stochastically larger than L_{d_1}, written $L_{d_2} \geq_s L_{d_1}$, if $|L_{d_2} \leq a| \leq |L_{d1} \leq a|$ for every a. Criterion g respects the stochastic ordering with respect to list L if

$$L_{d_2} \geq_s L_{d_1} \Rightarrow g(C_{d_1}) \leq g(C_{d_2})$$

The nnd order on information matrices (or their M-P inverses) implies the stochastic order on both the lists of canonical variances and the lists of pairwise variances.

Fact: In the reference universe of all binary block designs with v treatments and fixed block size distribution, if L is the list of canonical variances, then $L_{d_2} \geq_s L_{d_1} \iff L_{d_2} = L_{d1}$.

Proof: This follows from fixed trace of the information matrix in the reference universe, and that element-wise inversion of nonnegative lists reverses the stochastic ordering.

Thus every ordering criterion that is a function of the list of canonical variances trivially respects the stochastic order over the binary class. This may not be so for a criterion based on the list of pairwise variances.

A weaker ordering of lists than stochastic ordering, which is of some interest and which is not trivially respected in the binary class, is the *weak majorization ordering*. Let $L_{d[i]}$ be the i^{th} largest member of list L_d. Define L_{d_2} to weakly majorize L_{d_1}, written $L_{d_2} \geq_m L_{d_1}$, if $\sum_{i=1}^{t} L_{d_2[i]} \geq \sum_{i=1}^{t} L_{d_1[i]}$ for every $t = 1, 2, \ldots, s$. If also equality of the two sums holds at $t = s$, then L_{d_2} is said simply to *majorize* L_{d_1}. Criterion g respects the weak majorization ordering with respect to list L if

$$L_{d_2} \geq_m L_{d_1} \Rightarrow g(C_{d_1}) \leq g(C_{d_2}).$$

The weak majorization ordering is respected by every function of the form $g(C_d) = \sum_{i=1}^{s} h(L_{di})$ for continuous, increasing, convex h.

For any connected design d, the inverses of the canonical variances are the eigenvalues of the information matrix C_d. Now the list of eigenvalues has constant sum for all d in the reference universe; for these lists, majorization and weak majorization are equivalent. Moreover, if two lists of eigenvalues are ordered by majorization, then the corresponding lists of canonical variances are ordered by weak majorization. Consequently, weak majorization can sometimes be determined for canonical variances over the reference universe via the corresponding eigenvalues of information matrices.

Relationships among the three ordering principles discussed are

$$\text{nnd ordering} \Rightarrow \text{stochastic ordering} \Rightarrow \text{weak majorization ordering}$$

the latter two for either the pairwise variances or the canonical variances. None of the implications can in general be reversed.

We call a symmetric ordering criterion an *optimality criterion* if (1) it preserves the nnd definite ordering of information matrices over the generalized universe of all designs for given v and block size distribution, and (2) it admits direct interpretation as a summary measure of magnitude of variances of one or more treatment contrast estimators. Each of the functions in `optimality_criteria` possesses these two properties.

Ordering criteria can fall outside this scope yet still be of interest, such as those provided in the element `other_ordering_criteria.` These functions, discussed next, typically fail on both requirements for an optimality criterion, but may preserve orderings in restricted classes.

The S-criterion ($\mathrm{tr}(C_d^2)$) is typically employed as the second step in a so-called (M, S)-optimality argument: first maximize $\mathrm{tr}(C_d)$ (that is, restrict to the binary class - our reference universe), then minimize S. Within the binary class, S preserves the weak majorization order on the canonical variances; outside of that class, it is possible to find considerably smaller values of S, though inevitably at considerable cost on one or more optimality criteria. Thus S may be viewed as an ordering criterion suitable for use in restricted classes, and/or in a subsidiary role to one or more optimality criteria in a multi-criterion design screening.

The function `max_min_ratio_canonical_variances` preserves the weak majorization order over the binary class (indeed within any fixed $\mathrm{tr}(C_d)$ class), and `max_min_ratio_pairwise_variances` preserves the majorization order over that class. Both suffer the same defects as S outside the reference universe. Each of these three criteria is a summary measure of scatter of variances, not of magnitude; minimizing over too large a class will reduce scatter at the cost of increasing magnitude.

Two additional ordering criteria implemented are the support sizes of the distributions of canonical variances and pairwise variances. These, too, can be informative as subsidiary criteria in a multi-criterion design search, but because they do not employ the values in the corresponding distributions, `no_distinct_canonical_variances` and `no_distinct_pairwise_variances` cannot be guaranteed to preserve (outside of the reference universe) any of the list orderings discussed. Like S and the variance ratios, these measures give information on scatter in a list of variances, and thus are fairly called *balance criteria*.

Included with `other_ordering_criteria` are `absolute_comparisons` and `calculated_comparisons.` These serve the same role, and are computed with the same rules, as `absolute_efficiencies` and `calculated_efficiencies` for `optimality_criteria.` Because `other_ordering_criteria` typically do not measure magnitude of variance, we do not consider it correct terminological usage to call their relative values "efficiencies."

9 Lists of Block Designs

A list of block designs is essentially what the name implies. However, the listed designs must be distinct, and we allow assertions to be made about this list; in particular, it will be possible to say

- the designs in the list are pairwise non-isomorphic;

- these are all the designs with such-and-such properties.

Here is the schema definition for the *list_of_designs* entity which is the root element of any valid external representation document.

```
<list_of_designs> = {
    <header> ,
    <designs>
};

<header> =
         external_representation_version : "3.0"
         design_type                     : ( block_design | latin_square | mixed )
         number_of_designs               : ( $integer | unknown ) )
    ?( , <invariants> )
    ?( , pairwise_nonisomorphic        : ( $boolean | unknown ) )
    ?( , complete_upto_isomorphism     : ( $boolean | unknown ) )
    ?( , number_of_isomorphism_classes : ( $integer | unknown ) )
    ?( , precision                     : $integer )
    ?( , <info> )
;

<designs> = designs : [ <design> *( , <design> ) ] ;
```

There are two compulsory entries:

`external_representation_version`
: It will be used by applications to check compliance of documents and of themselves. It must contain a fixed string representing the current protocol version of external representation schema.

  In the future, the minor version number will be incremented when backward compatible minor changes have been made. That means that older documents satisfying previous protocols with the same major version number remain valid under the new protocol. The corresponding requirement for implementations is that an implementation in compliance with a given protocol version should be able to deal with any document of the same major and a lower protocol version.

  The major version will be incremented between not entirely compatible versions or when significant new structures have been introduced.

`designs`
: This is the entry under which each design is listed.

The optional `list_definition` component will be used to define *list_invariants* and to formulate *queries* to the database. These concepts are the subjects of future development.

10 Implementation Policies

(Under development)

The external representation for block designs gives the implementor a great deal of choice about what to include when specifying a block design and its properties. Here we record our policies about what (and what not) to include in certain cases:

`How far to go with point_concurrences?`
: If the given block design is not a t-design (with $t \geq 2$), then include the k-wise point concurrences only for $k = 1$ and (unless there is just one point) $k = 2$. In both cases, the full preimage should be given (which may be `entire_domain`). This policy gives the replication number for each point and the pairwise point concurrences. If the given block design D is a t-design (with $t = 2$) then include the k-wise point concurrences for $k = 1, 2, \ldots, max(t)$ for which D is a t-design. Again, full preimages should be given and they are all, of course, `entire_domain`.

`How far to go with block_concurrences?`
: Include the k-wise block concurrences for $k = 1$ and (unless there is just one block) $k = 2$. In both cases, preimages should be collapsed to preimage cardinalities. This policy gives the sizes of the blocks, the number of blocks of each size, the sizes of the pairwise intersections of blocks, and the number of pairs of blocks giving each intersection size.

`How far to go with t_wise_balanced?`
: This is analogous to `point_concurrences`. If the given block design D is not a t-design (with $t = 2$), then normally include whether or not D is t-wise balanced only for $t = 1$ and $t = 2$. Otherwise, include this information for $t = 1, 2, \ldots, max(t)$ for which D is a t-design. Note that this maximum t is recorded in the `t_design` indicator.

References

[1] Javascript object notation (json). http://www.json.org.

[2] Relax ng schema language for xml. http://relaxng.org.

[3] R.A. Bailey, Peter J. Cameron, Peter Dobcsányi, John P. Morgan, and Leonard H. Soicher. Designs on the web. *Discrete Mathematics*, 306(23):p3014 – 3027, 20061201.

[4] T. Beth, D. Jungnickel, and H. Lenz. *Design Theory*, volume 1 and 2 (Second edition). Cambridge University Press, 1999.

[5] T. Calinski and S. Kageyama. Block designs: A randomization approach. In *Lecture Notes in Statistics 150*. Springer, New York, 2000.

[6] Peter J. Cameron, editor. *Encyclopaedia of DesignTheory*. 2004.

[7] Peter J. Cameron, Peter Dobcsányi, John P. Morgan, and Leonard H. Soicher. Dtrs protocol version 2.0, 2004.

[8] PJ Cameron, P. Dobcsányi, JP Morgan, and LH Soicher. *The External Representation of Block Designs*.

[9] C.J. Colbourn and J.H. Dinitz, editors. *The CRC Handbook of Combinatorial Designs*. CRC Press, 1996.

[10] Douglas Crockford. Rfc4627: Javascript object notation, 2006.

[11] K. R. Shah and B. K. Sinha. *Theory of Optimal Designs*. Springer, New York, 1989.

A Design Schema

```
#
# The External Representation of Block Designs
#
# an EBNF based schema
#

<list_of_designs> = {
    <header> ,
    <designs>
};

<header> =
         external_representation_version : "3.0"
         design_type                     : ( block_design | latin_square | mixed )
         number_of_designs               : ( $integer | unknown ) )
    ?( , <invariants> )
    ?( , pairwise_nonisomorphic        : ( $boolean | unknown ) )
    ?( , complete_upto_isomorphism     : ( $boolean | unknown ) )
    ?( , number_of_isomorphism_classes : ( $integer | unknown ) )
    ?( , precision                     : $integer )
    ?( , <info> )
;

<invariants> = invariants : {
    # relations which hold over the list of designs
    # conjunction implicitly assumed
    ?( relations : [
              [ <rel_arg>, <rel_op>, <rel_arg> ]
          *( , [ <rel_arg>, <rel_op>, <rel_arg> ] )
    ]
    # optional invariant description as needed
    ?( invariant_description : $string )
};

<rel_arg> = <design_parameter> | <scalar> ;

<design_parameter> = t | v | b | ... | resolvable | ... ;

<rel_op> = "=" | "!=" | "<" | "<=" | ">" | ">=" ;

<designs> = designs : [ <design> *( , <design> ) ] ;

<design> = <block_design> | <latin_square> ;

<block_design> = {
         type      : block_design ,
         id        : ( $string | $integer ) ,
         v         : $integer ,
    ?(   b         : $integer , )
```

```
    ?(   precision : $integer , )
         <blocks>
    ?( , <point_labels> )
    ?( , <indicators> )
    ?( , <combinatorial_properties> )
    ?( , <block_design_automorphism_group> )
    ?( , <resolutions> )
    ?( , <statistical_properties> )
    ?( , <alternative_representations> )
    ?( , <info> )
};

<blocks> = blocks : [  <block> *( , <block> ) ];

<block> =  [ $integer *( , $integer ) ];

<point_labels> = point_labels : [
      ( $integer *( , $integer ) )
    | ( $string *( , $string ) )
];

<latin_square> =  { latin_square : <to be defined later> };

<indicators> = indicators : { <indicator> *( , <indicator> ) } ;

<indicator> =
      repeated_blocks     : $boolean
    | resolvable          : $boolean
    | affine_resolvable   : ( $boolean | { mu            : $integer } )
    | equireplicate       : ( $boolean | { r             : $integer } )
    | constant_blocksize  : ( $boolean | { k             : $integer } )
    | t_design            : ( $boolean | { maximum_t     : $integer } )
    | connected           : ( $boolean | { no_components : $integer } )
    | pairwise_balanced   : ( $boolean | { lambda        : $integer } )
    | variance_balanced   : $boolean
    | efficiency_balanced : $boolean
    | cyclic              : $boolean
    | one_rotational      : $boolean
;

<combinatorial_properties> =
    combinatorial_properties : {
        <point_concurrences> ,
        <block_concurrences> ,
        <t_design_properties> ,
        <alpha_resolvable> ,
        <t_wise_balanced>
};

<point_concurrences> = point_concurrences : [
    <function_on_ksubsets_of_indices> *( , <function_on_ksubsets_of_indices> )
```

```
];

<block_concurrences> = block_concurrences : [
    <function_on_ksubsets_of_indices> *( , <function_on_ksubsets_of_indices> )
];

<t_design_properties> = t_design_properties : {
    t_design_properties_member *( , t_design_properties_member )
};

<t_design_properties_member> =
      parameters : {
          t      : $integer ,
          v      : $integer ,
          b      : $integer ,
          r      : $integer ,
          k      : $integer ,
          lambda : $integer
      }
    | square                : $boolean
    | projective_plane      : $boolean
    | affine_plane          : $boolean
    | steiner_system        : ( $boolean | { t: $integer } )
    | steiner_triple_system : $boolean
;

<alpha_resolvable> =  { <index_flag> *( , <index_flag> ) };

<index_flag> = "$integer" : ( $boolean | unknown ) ;

<t_wise_balanced> =  { <t_index_flag> *( , <t_index_flag> ) } ;

<t_index_flag> = "$integer" : ( $boolean | unknown | { lambda: $integer } ) ;

<resolutions> = resolutions : {
    pairwise_nonisomorphic  : ( $boolean | unknown ) ,
    all_classes_represented : ( $boolean | unknown ) ,
    value                   : [ <resolution> *( , <resolution> ) ]
};

<resolution> =  {
          function_on_indices: <function_on_indices>
    ?( , <resolution_automorphism_group> )
};

<resolution_automorphism_group> = resolution_automorphism_group : {
          <permutation_group>
    ?( , <resolution_automorphism_group_properties> )
};

<resolution_automorphism_group_properties> =
```

```
    resolution_automorphism_group_properties : <to be defined later> ;

<block_design_automorphism_group> = automorphism_group: {
    <permutation_group>,
    <block_design_automorphism_group_properties>
};

<permutation_group> = permutation_group : {
         degree : $integer ,
         order  : $integer ,
         domain : points ,
         <generators>
    ?( , <permutation_group_properties> )
};

<permutation_group_properties> = permutation_group_properties : {
    <permutation_group_properties_member> *( , <permutation_group_properties_member> )
};

<permutation_group_properties_member> =
      primitive            : $boolean
    | generously_transitive : $boolean
    | multiplicity_free     : $boolean
    | stratifiable          : $boolean
    | no_orbits             : $integer
    | degree_transitivity   : $integer
    | rank                  : $integer
    | <cycle_type_representatives>
;

<block_design_automorphism_group_properties> =
    automorphism_group_properties: {
        block_primitive          : ( $boolean | not_applicable ) ,
        degree_block_transitivity : ( $integer | not_applicable ) ,
        no_block_orbits          : ( $integer | not_applicable )
};

<cycle_type_representatives> = cycle_type_representatives : [
        <cycle_type_representative> *( , <cycle_type_representative> )
];

<cycle_type_representative> = {
    permutation          : <permutation>,
    cycle_type           : [ $integer *( , $integer ) ] ,
    no_having_cycle_type : $integer
};

<generators> = generators : [ ? ( <permutation> *( , <permutation> ) ) ];

<permutation> = [ $integer *( , $integer ) ];
```

```
<alternative_representations> = alternative_representations : {
    <incidence_matrix>
};

<incidence_matrix> = incidence_matrix: {
    shape   : points_by_blocks ,
    <matrix>
};

<matrix> =
       no_rows    : $integer ,
       no_columns : $integer ,
    ?( title      : $string , )
       matrix     : [ <row> *( , <row> ) ]
;

<row> = [ <number> *( , <number> ) ];

<statistical_properties> = statistical_properties: {
          precision: $integer
    ?( ,  <canonical_variances> )
    ?( ,  <pairwise_variances> )
    ?( ,  <optimality_criteria> )
    ?( ,  <other_ordering_criteria> )
    ?( ,  <canonical_efficiency_factors> )
    ?( ,  <functions_of_efficiency_factors> )
    ?( ,  <robustness_properties> )
};

<robustness_properties> = robustness_properties: {
    ?( <robustness_properties_member> *( , <robustness_properties_member> ) )
};

<robustness_properties_member> =
      robust_connected_plots     : <robust_connected_value>
    | robust_connected_blocks    : <robust_connected_value>
    | robust_efficiencies_plots  : <robust_efficiencies_value>
    | robust_efficiencies_blocks : <robust_efficiencies_value>
;

<robust_efficiencies_value> = {
    precision: $integer ,
    robustness_efficiency_values: [
        <robustness_efficiency_values> * ( , <robustness_efficiency_values> )
    ]
};

<robustness_efficiency_values> = {
         number_lost               : $integer ,
         loss_measure              : ( average | worst ) ,
    ?( , phi_0                     : <robustness_efficiency_values_value> )
```

```
    ?( , phi_1                      : <robustness_efficiency_values_value> )
    ?( , maximum_pairwise_variances : <robustness_efficiency_values_value> )
    ?( , E_1                        : <robustness_efficiency_values_value> )
};

<robustness_efficiency_values_value> = {
         self_efficiency       : <number>
    ?( , absolute_efficiency   : ( <number> | unknown ) )
    ?( , calculated_efficiency : ( <number> | unknown ) )
};

<robust_connected_value> = {
    number_lost : $integer ,
    is_max      : ( true | unknown )
};

<functions_of_efficiency_factors> = functions_of_efficiency_factors: {
    geometric_mean : <number> ,
    minimum        : <number> ,
    harmonic_mean  : <number>
};

<canonical_efficiency_factors> = canonical_efficiency_factors: {
    no_distinct: $integer | unknown | not_applicable ,
    ordered: true | unknown ,
    value: [
             { multiplicity               : ( $integer | not_applicable ) ,
               canonical_efficiency_factor : ( <number> | blank ) }
         *( , { multiplicity               : ( $integer | not_applicable ) ,
               canonical_efficiency_factor : ( <number> | blank ) } )
    ]
};

<other_ordering_criteria> = other_ordering_criteria : {
    ?( <other_ordering_criteria_member> *( , <other_ordering_criteria_member>  ) )
};

<other_ordering_criteria_member> =
      trace_of_square_of_C             : <ordering_criteria_value1>
    | max_min_ratio_canonical_variances : <ordering_criteria_value1>
    | max_min_ratio_pairwise_variances  : <ordering_criteria_value1>
    | no_distinct_canonical_variances   : <ordering_criteria_value2>
    | no_distinct_pairwise_variances    : <ordering_criteria_value2>
;

<ordering_criteria_value1> = {
         value                 : ( <number> | not_applicable )
    ?( , absolute_comparison   : ( <number> | unknown ) )
    ?( , calculated_comparison : ( <number> | unknown ) )
};
```

```
<ordering_criteria_value2> = {
         value                 : ( $integer | unknown | not_applicable )
    ?( , absolute_comparison   : ( <number> | unknown ) )
    ?( , calculated_comparison : ( <number> | unknown ) )
};

<optimality_criteria> = optimality_criteria: {
    ?( <optimality_criteria_member> *( , <optimality_criteria_member> ) )
};

<optimality_criteria_member> =
      phi_0                     : <optimality_criteria_value>
    | phi_1                     : <optimality_criteria_value>
    | phi_2                     : <optimality_criteria_value>
    | maximum_pairwise_variances : <optimality_criteria_value>
    | E_criteria                : { <E_value> *( , <E_value> ) }
;

<optimality_criteria_value> = {
         value                 : ( <number> | not_applicable )
    ?( , absolute_efficiency   : ( <number> | unknown ) )
    ?( , calculated_efficiency : ( <number> | unknown ) )
};

<E_value> =  "$integer" : {
         value                 : ( <number> | not_applicable )
    ?( , absolute_efficiency   : ( <number> | unknown ) )
    ?( , calculated_efficiency : ( <number> | unknown ) )
};

<pairwise_variances> =
    # function with domain_base=points and k=2
    pairwise_variances: <function_on_ksubsets_of_indices> ;

<canonical_variances> = canonical_variances: {
    no_distinct : ( $integer | unknown | not_applicable ) ,
    ordered     : ( true | unknown ) ,  # do we need this?
    value: [
             { multiplicity       : ( $integer | not_applicable ) ,
               canonical_variance : ( <number> | blank | not_applicable ) }
         *( , { multiplicity      : ( $integer | not_applicable ) ,
               canonical_variance : ( <number> | blank | not_applicable ) } )
    ]
};

<function_on_ksubsets_of_indices> = {
       domain_base       : ( points | blocks ) ,
       n                 : $integer ,
       k                 : $integer ,
       ordered           : ( true | unknown ) ,
    ?( image_cardinality : $integer , )
```

```
    ?( precision         : $integer , )
    ?( title             : $string , )
        maps             : [ ?( <map> *( , <map> ) ) ]
};

<function_on_indices> = {
       domain            : ( points | blocks ) ,
       n                 : $integer ,
       ordered           : ( true | unknown ) ,
    ?( image_cardinality : $integer , )
    ?( precision         : $integer , )
    ?( title             : $string , )
        maps              : [ ?( <map> *( , <map> ) ) ]
};

<map> = {
    ( <preimage> | <preimage_cardinality> | blank ) ,
    image   : ( <number> | not_applicable )
};

<preimage> = preimage : [
      $integer *( , $integer )
    # Removed: word ksubset, substituting with a seq of lists
    | [ $integer *( , $integer ) ] *( , [ $integer *( , $integer ) ] )
    | entire_domain
];

<preimage_cardinality> = preimage_cardinality : $integer ;

<info> = info : {
         software  : [ $string *( , $string ) ]
    ?( , reference : [ ?( $string *( , $string ) ) ] )
    ?( , note      : [ ?( $string *( , $string ) ) ] )
};

# scalars
<scalar> = <number> | $boolean | $null | unknown
<number> = $integer | $float | <rational> ;
<rational> = { Q : [ $integer, $integer ] } ;
```

B An example

Here in its entirety is the example which we have seen in parts throughout this document.


```
{
  "external_representation_version" : "3.0",
  "design_type" : "block_design",
  "number_of_designs" : 1,
  "invariants" : {
    "relations" : [
      ["t", "=", 2],
      ["v", "=", 7],
      ["b", "=", 7],
      ["r", "=", 3],
      ["k", "=", 3],
      ["lambda", "=", 1]
    ]
  },
  "pairwise_nonisomorphic" : true,
  "complete_upto_isomorphism" : true,
  "info" : {
    "software" : [
      "[ DESIGN-1.1, GRAPE-4.2, GAPDoc-0.9999, GAP-4.4.3 ]",
      "[ bdstat-0.8/280, numarray-1.1.1, pydesign-0.5/274, python-2.4.0.candidate.1 ]",
      "['python 2.5.2.final.0', 'blockdesign 0.1.5', 'ddb2 0.2.1', 'ddb-xt2json 0.1']"
    ]
  },
  "designs" : [
    {
      "type" : "block_design",
      "id" : "t2-v7-b7-r3-k3-L1-0",
      "v" : 7,
      "b" : 7,
      "blocks" : [
        [0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6],
        [2, 4, 5]
      ],
      "indicators" : {
        "repeated_blocks" : false,
        "resolvable" : false,
        "affine_resolvable" : false,
        "equireplicate" : {
          "r" : 3
        },
        "constant_blocksize" : {
          "k" : 3
        },
        "t_design" : {
          "maximum_t" : 2
        },
        "connected" : {
```

```
    "no_components" : 1
  },
  "pairwise_balanced" : {
    "lambda" : 1
  },
  "variance_balanced" : true,
  "efficiency_balanced" : true,
  "cyclic" : true,
  "one_rotational" : false
},
"combinatorial_properties" : {
  "point_concurrences" : [
    {
      "domain_base" : "points",
      "n" : 7,
      "k" : 1,
      "ordered" : true,
      "title" : "replication_numbers",
      "maps" : [
        {
          "preimage" : ["entire_domain"],
          "image" : 3
        }
      ]
    },
    {
      "domain_base" : "points",
      "n" : 7,
      "k" : 2,
      "ordered" : true,
      "title" : "pairwise_point_concurrences",
      "maps" : [
        {
          "preimage" : ["entire_domain"],
          "image" : 1
        }
      ]
    }
  ],
  "block_concurrences" : [
    {
      "domain_base" : "blocks",
      "n" : 7,
      "k" : 1,
      "ordered" : "unknown",
      "title" : "block_sizes",
      "maps" : [
        {
          "preimage_cardinality" : 7,
          "image" : 3
        }
```

```
      ]
    },
    {
      "domain_base" : "blocks",
      "n" : 7,
      "k" : 2,
      "ordered" : "unknown",
      "title" : "pairwise_block_intersection_sizes",
      "maps" : [
        {
          "preimage_cardinality" : 21,
          "image" : 1
        }
      ]
    }
  ],
  "t_design_properties" : {
    "parameters" : {
      "t" : 2,
      "v" : 7,
      "b" : 7,
      "r" : 3,
      "k" : 3,
      "lambda" : 1
    },
    "square" : true,
    "projective_plane" : true,
    "affine_plane" : false,
    "steiner_system" : {
      "t" : 2
    },
    "steiner_triple_system" : true
  },
  "alpha_resolvable" : {
    "3" : true
  },
  "t_wise_balanced" : {
    "1" : {
      "lambda" : 3
    },
    "2" : {
      "lambda" : 1
    }
  }
},
"automorphism_group" : {
  "permutation_group" : {
    "degree" : 7,
    "order" : 168,
    "domain" : "points",
    "generators" : [
```

```
      [1, 0, 2, 3, 5, 4, 6], [0, 2, 1, 3, 4, 6, 5],
      [0, 3, 4, 1, 2, 5, 6], [0, 1, 2, 5, 6, 3, 4],
      [0, 1, 2, 4, 3, 6, 5]
    ],
    "permutation_group_properties" : {
      "primitive" : true,
      "generously_transitive" : true,
      "multiplicity_free" : true,
      "stratifiable" : true,
      "no_orbits" : 1,
      "degree_transitivity" : 2,
      "rank" : 2,
      "cycle_type_representatives" : [
        {
          "permutation" : [1, 3, 5, 2, 0, 6, 4],
          "cycle_type" : [7],
          "no_having_cycle_type" : 48
        },
        {
          "permutation" : [0, 2, 1, 5, 6, 4, 3],
          "cycle_type" : [1, 2, 4],
          "no_having_cycle_type" : 42
        },
        {
          "permutation" : [0, 3, 4, 5, 6, 1, 2],
          "cycle_type" : [1, 3, 3],
          "no_having_cycle_type" : 56
        },
        {
          "permutation" : [0, 1, 2, 4, 3, 6, 5],
          "cycle_type" : [1, 1, 1, 2, 2],
          "no_having_cycle_type" : 21
        },
        {
          "permutation" : [0, 1, 2, 3, 4, 5, 6],
          "cycle_type" : [1, 1, 1, 1, 1, 1, 1],
          "no_having_cycle_type" : 1
        }
      ]
    }
  },
  "automorphism_group_properties" : {
    "block_primitive" : true,
    "no_block_orbits" : 1,
    "degree_block_transitivity" : 2
  }
},
"statistical_properties" : {
  "precision" : 9,
  "canonical_variances" : {
    "no_distinct" : 1,
```

```
  "ordered" : true,
  "value" : [
    {
      "multiplicity" : 6,
      "canonical_variance" : 0.428571429
    }
  ]
},
"pairwise_variances" : {
  "domain_base" : "points",
  "n" : 7,
  "k" : 2,
  "ordered" : true,
  "maps" : [
    {
      "preimage" : ["entire_domain"],
      "image" : 0.857142857
    }
  ]
},
"optimality_criteria" : {
  "phi_0" : {
    "value" : -5.08378716,
    "absolute_efficiency" : 1,
    "calculated_efficiency" : 1
  },
  "phi_1" : {
    "value" : 0.428571429,
    "absolute_efficiency" : 1,
    "calculated_efficiency" : 1
  },
  "phi_2" : {
    "value" : 0.183673469,
    "absolute_efficiency" : 1,
    "calculated_efficiency" : 1
  },
  "maximum_pairwise_variances" : {
    "value" : 0.857142857,
    "absolute_efficiency" : 1,
    "calculated_efficiency" : 1
  },
  "E_criteria" : {
    "1" : {
      "value" : 0.428571429,
      "absolute_efficiency" : 1,
      "calculated_efficiency" : 1
    },
    "2" : {
      "value" : 0.857142857,
      "absolute_efficiency" : 1,
      "calculated_efficiency" : 1
```

```
    },
    "3" : {
      "value" : 1.28571429,
      "absolute_efficiency" : 1,
      "calculated_efficiency" : 1
    },
    "4" : {
      "value" : 1.71428571,
      "absolute_efficiency" : 1,
      "calculated_efficiency" : 1
    },
    "5" : {
      "value" : 2.14285714,
      "absolute_efficiency" : 1,
      "calculated_efficiency" : 1
    },
    "6" : {
      "value" : 2.57142857,
      "absolute_efficiency" : 1,
      "calculated_efficiency" : 1
    }
  }
},
"other_ordering_criteria" : {
  "trace_of_square_of_C" : {
    "value" : 32.6666667,
    "absolute_comparison" : 1,
    "calculated_comparison" : 1
  },
  "max_min_ratio_canonical_variances" : {
    "value" : 1.0,
    "absolute_comparison" : 1,
    "calculated_comparison" : 1
  },
  "max_min_ratio_pairwise_variances" : {
    "value" : 1.0,
    "absolute_comparison" : 1,
    "calculated_comparison" : 1
  },
  "no_distinct_canonical_variances" : {
    "value" : 1,
    "absolute_comparison" : 1,
    "calculated_comparison" : 1
  },
  "no_distinct_pairwise_variances" : {
    "value" : 1,
    "absolute_comparison" : 1,
    "calculated_comparison" : 1
  }
},
"canonical_efficiency_factors" : {
```

```
          "no_distinct" : 1,
          "ordered" : true,
          "value" : [
            {
              "multiplicity" : 6,
              "canonical_efficiency_factor" : 0.777777778
            }
          ]
        },
        "functions_of_efficiency_factors" : {
          "harmonic_mean" : 0.777777778,
          "geometric_mean" : 0.777777778,
          "minimum" : 0.777777778
        }
      }
    }
  ]
}
```