# Bayesian Supersaturated Designs for Factor Screening (Version 2.2)

This document describes the implementation of a MATLABTM class to generate supersaturated designs for factor screening using Bayesian principles. The class implements the approach due to Jones *et al* [iv]. The primary advantages of the approach over alternate methods, say based on balanced incomplete block designs, are:

1. The number of experimental runs may be of arbitrary size, maybe an odd number and is not restricted to pure binary powers or multiples of four. However, it should be said that we prefer balanced designs, as it is rare to favour one factor level over another. Consequently, here we restrict ourselves to balanced designs only.
2. Prior physical knowledge relating to the statistical significance of factors may be incorporated directly.
3. Categorical variables or blocking factors at more than 2-levels can be incorporated straightforwardly. Nonetheless, as our primary application here is factor screening, in this text we will restrict the development to factors at 2-levels only.
4. The design generator is the computationally efficient *columnwise-pairwise algorithm* due to Li and Wu [x]. Unlike many optimal design approaches, this method does not require the generation of a candidate set.

Contents

[Bayesian Supersaturated Designs for Factor Screening (Version 2.2) 1](#_Toc29030899)

[1 Summary 2](#_Toc29030900)

[2 Preamble 3](#_Toc29030901)

[2.1 An Alternative Interpretation: Reformulation as a Regularised Least Squares Problem 6](#_Toc29030902)

[3 Implementation of the Columnwise-Pairwise Design Generator 8](#_Toc29030903)

[3.1 Design Evaluation Measures and their Interpretation 11](#_Toc29030904)

[3.1.1 The Criterion 11](#_Toc29030905)

[3.1.2 Maximum Absolute Correlation 12](#_Toc29030906)

[3.1.3 Average Absolute Correlation 13](#_Toc29030907)

[3.1.4 The “” Diagnostic 13](#_Toc29030908)

[4 MATLAB Package ***SSDdesign*** 13](#_Toc29030909)

[4.1 Installing the Package 15](#_Toc29030910)

[4.2 ***BayesianSSD*** Class Properties Defining the Public Interface 16](#_Toc29030911)

[4.3 ***BayesianSSD*** Class Methods 19](#_Toc29030912)

[4.3.1 The Class Constructor Method: ***BayesianSSD***. 20](#_Toc29030913)

[4.3.2 The ***desginGenerator*** Method. 22](#_Toc29030914)

[4.3.3 The ***importScaledDesign*** Method 25](#_Toc29030915)

[4.4 Example Design Using the ***makeSSD*** Class 26](#_Toc29030916)

[4.4.1 ***makeSSD*** Class Properties 27](#_Toc29030917)

[4.4.2 ***makeSSD*** Class Methods 28](#_Toc29030918)

[4.4.3 The ***generateDesign*** Method 29](#_Toc29030919)

[4.4.4 The ***makeHistograms*** Method 32](#_Toc29030920)

[5 Comparison with Other Methods 32](#_Toc29030921)

[6 Future Enhancements 38](#_Toc29030922)

[6.1 Alternative Optimal Design Algorithms (Medium Effort) 38](#_Toc29030923)

[6.2 Supporting Parallel Processing (Low Effort) 39](#_Toc29030924)

[6.3 Implement a GUI (Medium Effort) 39](#_Toc29030925)

[6.4 Multi-Objective Optimisation (Large Effort) 40](#_Toc29030926)

[7 References 40](#_Toc29030927)

# Summary

The article is organised as follows: in section 2 we introduce the principal concepts underpinning the code. This includes the definition of an and a brief review of relevant Bayesian concepts, including the definition of appropriate priors, conditional and posterior densities. At the end of section 2 we state the Bayesian optimal design measure. In section 2.1 we derive an equivalent frequentist view of the design measure. This is based on noting the similarity of the Bayesian information matrix with that of an appropriate ridge regression estimator.

In section 3, we detail the optimal design algorithm used to generate SSDs. This is the ColumnWise-PairWise (CWPW) method due to Li and Wu [x]. Unlike many previous approaches, this is a candidate list free method and preserves design balance, which we consider beneficial. The algorithm has been coded as part of a MATLAB package, ***SSDdesign***, which forms the subject of section 4. The class architecture and public interface definition are found here. As the CWPW method is not guaranteed to return exact D-optimal designs, this motivates a *trial concept*, where the algorithm is run from many random starting points and the best design selected. This is the role played by the ***makeSSD*** class described in section 4.4. This section also includes a worked example illustrating the design generation process in its entirety.

In section 5, we compare results from our algorithm with those published previously in the literature. The performance of the algorithm is broadly equivalent to that published in previous work, with our designs exhibiting very similar properties to those quoted by Li and Wu and are superior to those by Nguyen [xv] based on balanced incomplete block designs. The paper by Jones et al presents improved criterion compared to Li and Wu and Nguyen, however they employ the optimal design algorithm due to Meyer and Nachtsheim [[[1]](#endnote-2)], which may not preserve design balance.

Finally, in section 6 we discuss possible future work, including enhancements to the existing code and including additional algorithms. In addition, we give some indication of the effort involved to implement these.

# Preamble

When the number of factors is very large, or in cases where the experimental runs are very expensive, a supersaturated design is appropriate for factor screening. A has runs and factors, such that , so there are insufficient degrees of freedom to allow estimation of all of the main effects. As the corresponding regression matrix for a is not of full row rank; *i.e.* is rank deficient. The appropriate regression model corresponding to a is the main effects model:

(1)

Where, **,**  are the predictor variables and are the model parameters to be estimated from the data. Alternatively, using an obvious matrix notation, we have:

(2)

Where , , and **.** In the screening scenario, the number of statistically significant, or active, factors are unknown at the outset. Consequently, the exact form of the model is uncertain, and this complicates the application of classical optimal design approaches; which assume the exact form of the model is known a priori. This uncertainty relating to the number of active factors, has led some researchers [[[2]](#endnote-3), [[3]](#endnote-4)] to consider the *average efficiency of a design*, where the average is taken over a set of potential models.

Let denote the weight (or prior probability) assigned to the model, the associated design and a measure of the efficiency or optimality for model . Under these circumstances, the appropriate model robust design criterion is:

(3)

Bayesian methods are ideal for accounting for uncertainty. For a more complete discussion with respect to SSD, we refer you to the work of Jones *et al* [[[4]](#endnote-5)]. In this text, we provide only a rudimentary treatment. Following Jones, for SSD, for a linear model with independently, identically distributed Gaussian errors we assume, without loss of generality:

* The *prior* *distribution* is assumed to be .
* The conditional distribution of the data given the is assumed to be .
* The posterior distribution for given is , where and .

Regardless, the Bayesian optimal design, , satisfies [[[5]](#endnote-6)]:

(4)

DuMouchel and Jones [[[6]](#endnote-7)] incorporated model uncertainty into criterion (4) through their choice of prior, . They divided the set of model terms into two sets: *primary* and *potential*. Primary terms are assumed to be *active*. Potential or screening terms are factors that may or may not be significant. For example, in a large screening experiment, the primary terms might correspond to the intercept and main effects. Similarly, the remaining potential terms might correspond to factors to be screened, where strong supporting physical evidence is lacking.

*Primary terms are assumed to be active*, and since no direction is assumed for their effects, DuMouchel and Jones utilise an *uninformative* or *diffuse* prior with arbitrary prior mean and prior variance tending to infinity. The potential terms are assumed to be distributed as *; i.e.* possess a prior mean of zero and variance . *This reflects the prior belief that these parameters are not likely to be particularly explanatory*, even if they turn out to be active.

For this prior distribution, the information matrix of the parameters is:

(5)

Hence, in practice the Bayesian optimality criterion, (4), becomes:

(6)

Note, although for , , for an appropriate choice of .

With respect to Lithium-Ion battery state of health testing, we are very likely to know from physical or chemical reasoning that some factors will be active. In the current context, we would consider these factors to be primary. For example, we might reasonably expect battery temperature to affect ; *i.e.* battery temperature would be a primary factor. In contrast, a new anode or cathode material compositions are factors we would like to screen; *i.e.* are potential factors. Under these circumstances it is necessary that .

Of course, if prior knowledge relating to the behaviour of the system, we may assume that all factors are *potential* in nature. In this scenario:

(7)

The information matrix is reminiscent of that found in *ridge regression* [[[7]](#endnote-8)], where the ridge parameters are applied to the potential terms only. Ridge regression can also be thought of more generally as *zero-order regularisation*. This insight permits an equivalent *Frequentist* view of the problem and is discussed in the next section.

## An Alternative Interpretation: Reformulation as a Regularised Least Squares Problem

We assume at the outset there are primary factors and factors. Further, we assume that . However, as required for a . The Bayesian prior can be interpreted as a zero-order regularisation function, where the regularisation penalty is applied only to the -potential factors. With this in mind, the model is first partitioned as follows:

(8)

Where, , , and . Regardless, the appropriate regularised cost function is:

(9)

Differentiating (9) *w.r.t.* , using standard rules on matrix derivatives, and setting the result to 0, then after some algebra yields:

(10)

Similarly, differentiating (9) *w.r.t.* and setting the result to 0, yields after some manipulation:

(11)

Equations (10) and (11) can be combined into a single matrix relation:

(12)

Using a result for partitioned matrices provided by Searle [[[8]](#endnote-9)], the coefficient matrix can be written:

(13)

Applying the rules for the determinant of products of block diagonal matrices, the determinant of the coefficient matrix is:

(14)

Maximisation of (14) is equivalent to maximising the corresponding Bayesian criterion (6). Initially, this may not seem advantageous, especially given the apparent complexity of the second determinant on the right-hand side of (14); but recall that generally is small. Consequently, is of low dimension. We can simplify matters further by applying the following factorisation to :

(15)

Where , and has orthonormal columns, and is an upper triangular matrix. Based on Householder transformations [[[9]](#endnote-10)], this factorisation is efficient to compute. It can be shown through direct multiplication, applying standard rules on matrix inverses and noting that is idempotent, as :

(16)

This avoids the necessity to explicitly calculate the inverse of . Maximisation of (16) is equivalent to maximising the Bayesian criterion (6). We now focus on the nature of the design generation algorithm intended to maximise either (6) or (16). This is the subject of the next section.

# Implementation of the Columnwise-Pairwise Design Generator

Here, we maximise the Bayesian criterion using the *Columnwise-Pairwise* algorithm due to Li and Wu [[[10]](#endnote-11)]. Unlike many other algorithms, this algorithm is candidate set free. This is a significant advantage when the number of factors is large. For example, an unconstrained candidate set for 20 factors, each at 2-levels, has over 16 million vertices. Clearly, avoiding the necessity to store such large numbers of potential conditions is advantageous. Further, the Li and Wu algorithm is computationally efficient and preserves *design* *balance*. As a design property, balance is desirable unless we favour one factor level over another, which rarely happens (perhaps in the presence of a *control*, *e.g.* Dunnett’s test [[[11]](#endnote-12)]). Further as Li and Wu point out:

* Design balance ensures that each column is uncorrelated with the intercept term in the model.
* Design balance yields substantial computational savings in both the column exchange and optimisation procedures.

Consequently, with factor screening applications in mind, we restrict ourselves in this work exclusively to balanced designs. Naturally, for a balanced design must be a multiple of two.

To begin, consider a supersaturated design with all factors at 2-levels. Define . Then , where is the row of the design matrix . The fundamental concept behind Li and Wu’s algorithm is that after excluding a *bad* column, we acquire a replacement by *performing a self-adjustment of the bad column*. Following Li and Wu, we first define:

* *Definition 1*: For a vector , is called a first-order adjustment vector of if is acquired by reversing one positive and one negative element only.
* *Definition 2*: For a vector , the first-order adjustment set of is .

Likewise, an -order adjustment vector of can be defined if pairs of ordinates, undergo a sign reversal and vice versa; *i.e.* . The -order adjustment set is denoted by . Thus, choosing a new vector from amounts to making an -order adjustment of , . The brute-force approach would be the so-called columnwise k-exchange algorithm, where the replacement column would be selected from all possible -order adjustments. This would require evaluations. Consequently, Li and Wu restrict replacements from only, requiring only evaluations. For example, let . If column replacement is based on an -order adjustment of , then there are 924 possibilities. If the adjustment is made on , there are only 36 possibilities.

Before stating the algorithms employed, we introduce some necessary notation. Let denote the Bayesian criterion for deleting the column of the scaled design matrix. That is:

(17)

Where denotes the regression matrix with the column removed and denotes the prior information matrix with the diagonal element removed. Similarly, define as:

(18)

Where , *i.e.* a member of the first order adjustment set for the column. With these quantities defined, the first algorithm is:

1. Begin by computing random, balanced designs, each with columns. Select the design with the largest Bayesian criterion as the initial design.
2. At each iteration, compute for each column, , in the design matrix. Let denote .
3. For column , identified from step 2, compute the corresponding first-order adjustment set . For each possible , compute (18).
4. Let denote the first-order adjustment vector for which (18) is maximised. Exchange for .
5. Repeat steps 2 through 4 until no improvement in (6) can be found.

Algorithm 1 is equivalent to the columnwise Van Schalkwyk algorithm [[[12]](#endnote-13)]. We may also exchange more than one column at a time to produce a *k-exchange* columnwise, pairwise algorithm. This leads to the second algorithm:

1. Begin by computing random, balanced designs, each with columns. Select the design with the largest Bayesian criterion as the initial design.
2. At each iteration, compute for each column, , in the design matrix. Sort the in descending order, from largest to the smallest, such that , hence the *k*-columns to exchange are the first .
3. For
   1. For column , identified from step 2, compute the corresponding first-order adjustment set . For each possible , compute (18).
   2. Let denote the first-order adjustment vector for which (18) is maximised. Exchange for .
4. Repeat steps 2 and 3 until no improvement in (6) can be found.

## Design Evaluation Measures and their Interpretation

The Bayesian criterion, equation (6), is just one measure of the design quality. Although not optimised, several other criteria are calculated and available to the analyst to assess the efficacy of the design. In this section we define and briefly review these quantities.

### The Criterion

This criterion was originally proposed by Booth and Cox [[[13]](#endnote-14)]. Nguyen [xv] provides the following rationale behind the measure. This utilises the singular value decomposition [[[14]](#endnote-15)] to decompose the regression matrix as:

(19)

Where and are orthogonal matrices and is diagonal. The diagonal elements of , , are known as the singular values of . From (19), it is immediately apparent that and share the same set of nonzero singular values, and since , where is the corresponding eigenvalue of and the rank of . Moreover, . Minimising is equivalent to minimising and is the same as making the as equal as possible, with . This approximates the A-optimality criterion in optimal design, which requires the minimisation of , or alternatively the D-optimality criterion, which attempts to maximise . This reveals the proper interpretation of , which is essentially an overall measure of the orthogonality of the design. Note is a “smaller-the-better” criterion.

If the design is balanced, then the sum of each column of the scaled design matrix, , must be zero. Consequently, the elements of , which implies the sum of the off-diagonal elements of equal - the sum of the diagonal elements of . Using this result, Nguyen goes on to show that the a lower bound for , when is divisible by , is:

(20)

### Maximum Absolute Correlation

Another reasonable criterion by with to interpret the design quality is the maximum absolute correlation among columns, . Ideally, this would be zero – implying an orthogonal design, but this is not possible because is row rank deficient. None of the algorithms presented here preclude creating duplicate columns in the design, because each column is updated independently. A design with is useless, as one or more column pairs are completely confounded. Consequently, we cannot discern the significance each factor involved separately; only as a pair.

In addition, if duplication occurs between any two factor columns it is possible for the Bayesian criterion to continue to increase, yet the design is useless since at least two factors are completely confounded. However, as the algorithm proceeds, confounding between columns may be eliminated. Therefore, upon convergence it is essential to check ***that the maximum absolute correlation among columns is less than unity***. ***If not, the design is not useful***.

### Average Absolute Correlation

The average absolute correlation, , also attempts to provide a measure of the orthogonality of the design. If the design were orthogonal, would be zero. Again, this is a “smaller-the-better” measure.

### The “c” Diagnostic

As we discussed in section 3.1.1, Nguyen [[[15]](#endnote-16)] showed that minimizing the criterion essentially attempts to set the singular values of the scaled design matrix equal, while keeping their sum of squares constant. His designs achieve this whenever . With these thoughts in mind, *Jones et al* [iv] suggested the criterion:

(21)

Where is now the largest singular value of . Note and achieves the upper bounds for Nguyen’s designs with . This is a “larger-the-better” criterion.

# MATLAB Package ***SSDdesign***

A MATLAB package, ***SSDdesign*** depicted in Figure 1, has been developed to implement a family of SSD design generators. A central concept in *object-oriented design*, is the *abstract* *class*. An abstract class is a class which cannot be initiated. Abstract classes are considered as incomplete classes and as such they cannot be called directly (instantiated). Abstract classes apply in situations where you wish to design an application, in which you intend some standard or common behaviour that is to be shared with all child classes. Typically, the parent declares abstract methods or properties that are defined explicitly in the child. In this way, the abstract parent defines a common interface, implying that groups of classes which inherit from the parent behave consistently to a user. In contrast, the child implements a custom application of the abstract methods or properties, specific to a given application.

Consider Figure 1. Class SSD lays at the heart of the class hierarchy. The purpose of this class is to define all class data, or *properties*, common to all algorithms. For example, the number of factors, the factor names, factor abbreviations and so on, are examples or properties required for all SSD design algorithms. These *concrete* properties are available to all child classes, and also their children, who inherit from SSD.

In addition, SSD declares two abstract methods, ***designGenerator***and ***isBetter*** and one abstract property, ***Measure***. The SSD class must be abstract, as the child classes necessarily implement different specific elements to generate the design. It is the nature of these specific items, e.g. the design measure, which differentiates the various SSD algorithms implemented. The concrete ***designGenerator*** method is implemented in the CWPW class. Further, the ***isBetter*** method must be unique and abstracts the concepts “*bigger the better*” and “*smaller the better*” design measures, facilitating comparisons among designs. In this way, the method calls are as similar as possible, yet custom with respect to each algorithm.

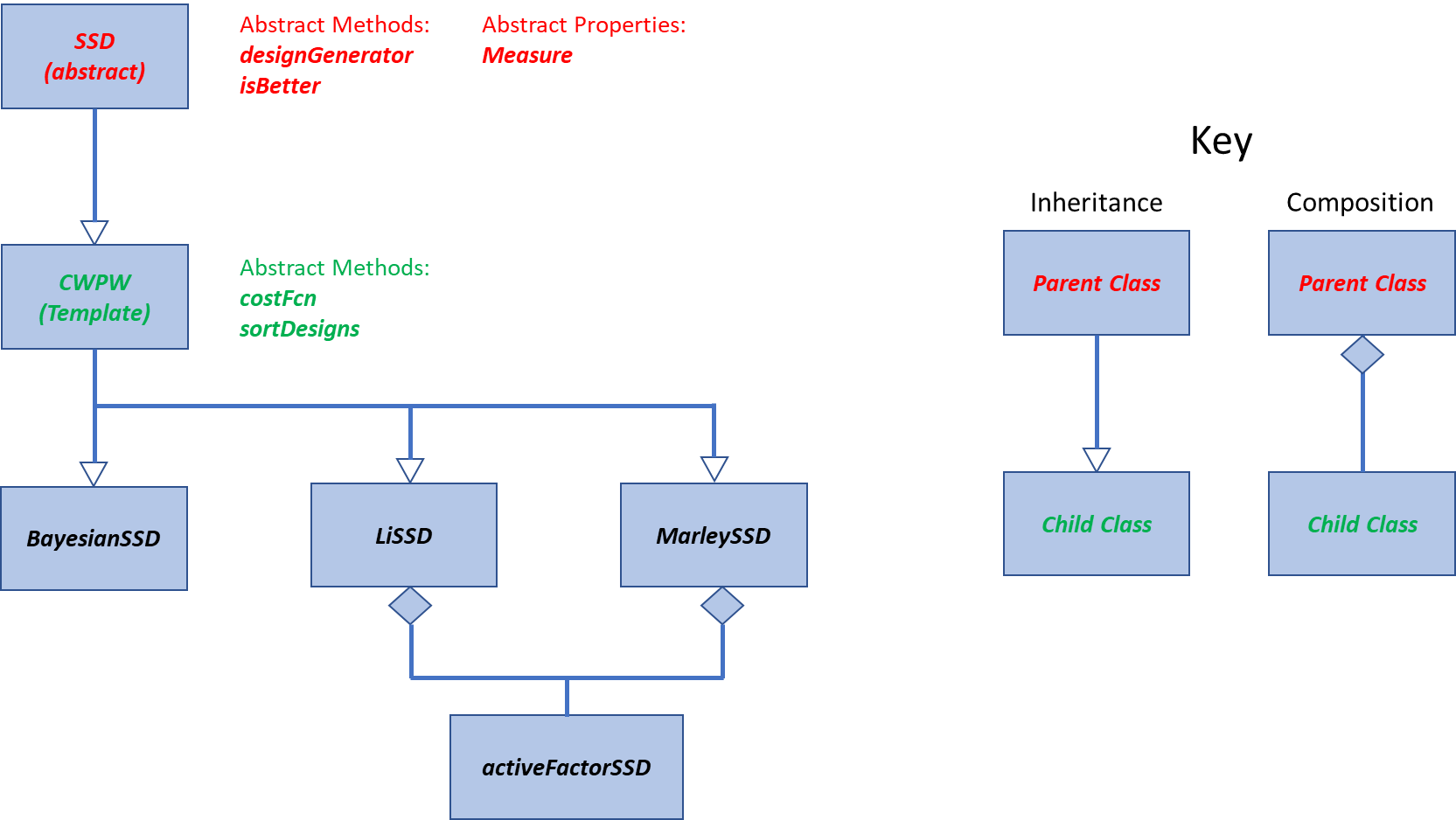


Figure 1: SSDdesign Package Class Architecture.

Note all algorithms employ the *columnwise*, *pairwise* (***CWPW***) optimal design algorithm as a generator. Here, we make use of an object-oriented design pattern called a “***template***”. The concept behind the ***template*** implementation is to define an algorithm, using abstract operations, which are subsequently overridden by the custom child class to provide specific, concrete behaviour. A ***template*** permits subclasses to redefine certain steps of an algorithm without letting them change its structure. This maximises the reuse of code.

Two aspects require abstraction: the individual design measure itself and its interpretation. The latter is essential for ranking designs. Consequently, the ***BayesianSSD***, ***MarleySSD*** and ***LiSSD*** classes all implement their own, concrete, ***costFcn*** and ***sortDesigns*** methods.

## Installing the Package

Create a directory in windows called ***SuperSaturatedDesigns***. Extract all files from ***SSD\_V1.zip*** to this directory. Use the MATLAB “***Set Path***” tool, see Figure 2, to add this directory to the path. Use the ***Add with Subfolders*** button. Click ***Save*** and close the ***Set Path*** pop-up window. The package is now installed.

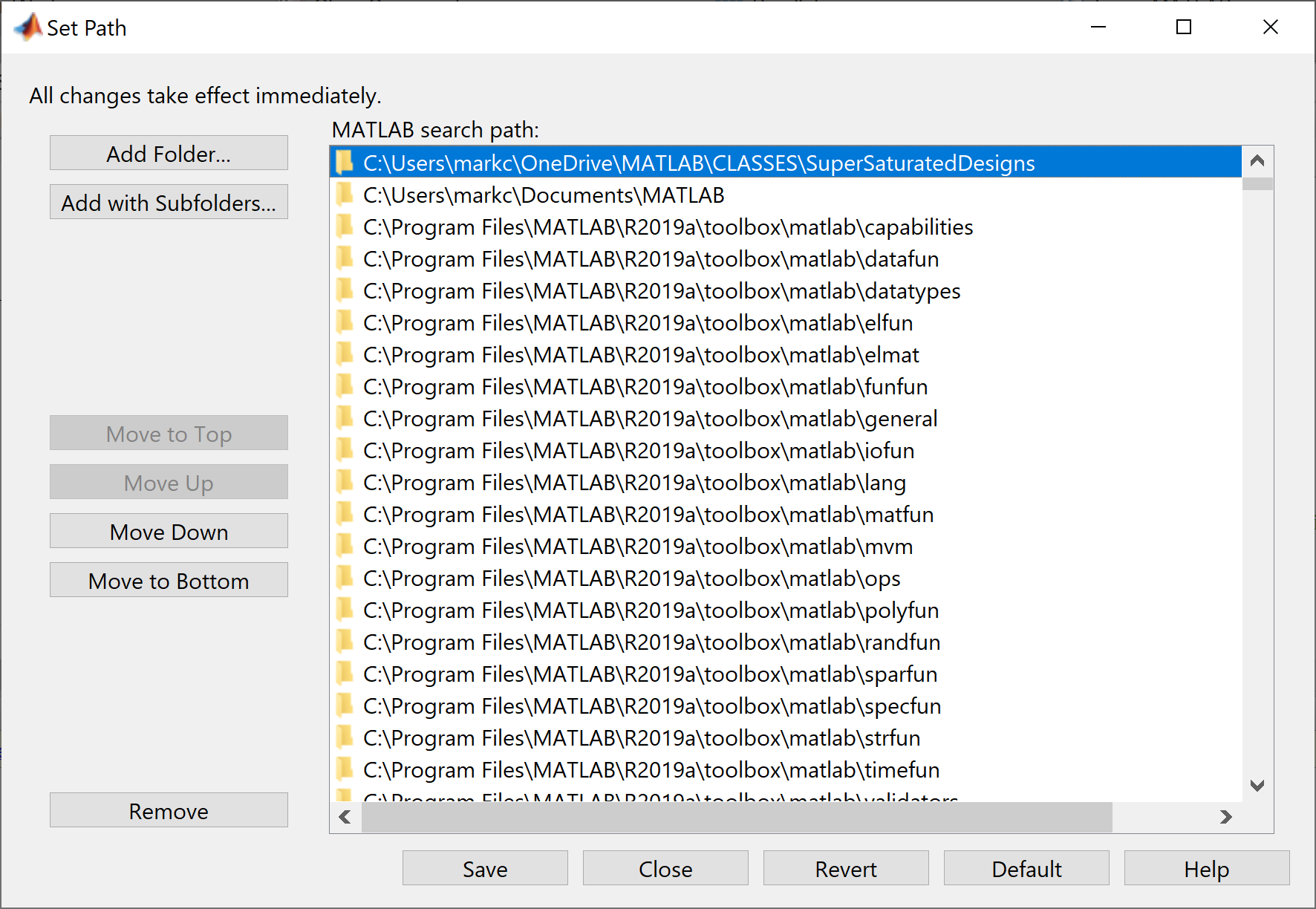


Figure 2: MATLAB "**Set Path**" pop-up window. You may also wish to move this directory to the bottom of the list, as MATLAB searches the path from top-to-bottom. To do this, click the “Move to Bottom” button and then click on “Save”.

Table 1 defines the classes contained within the package and their current status. All algorithms are fully implemented.

Table 1: SSD package files description and implementation status

|  |  |  |
| --- | --- | --- |
| **Class Name** | **Description** | **Implemented** |
| SSD | Parent class for all algorithms. Defines basic design information common to all algorithms. | Yes |
| SSDalgorithm | An enumeration class used to *label* the design generation algorithm under use. | Yes |
| BayesianSSD | The algorithm due to Jones *et al* [iv]. | Yes |
| LiSSD | The algorithm due to Li and Wu [x] | Yes |
| MarleySSD | The algorithm due to Marley [[[16]](#endnote-17)] | Yes |
| activeFactorSSD | A class to define subsets of the design of a user defined size. | Yes |
| CWPW | The ColumnWise PairWise optimal design algorithm. | Yes |
| TrialsSSD | An abstract class providing the fundamentals of the trials approach to design generation. | Yes |
| makeSSD | A container class to execute a series of design trials and select the best for any algorithmic approach implemented. | Yes |

## ***BayesianSSD*** Class Properties Defining the Public Interface

Table 2 provides a list of properties visible to the user; *i.e. public properties*. Also defined is their corresponding attribute. The “***SetAccess***” attribute, if set to ***public***, defines whether the value of a property is definable at the command line. If the “***SetAccess***” attribute is set to “***immutable***”, then the property can only be set as the object is instantiated; *i.e.* as an argument of the initial command line call creating the object. *After that, the property value cannot be chan*ged. This prevents unintended consequences downstream by preventing the user from overriding their initial choice. If ***SetAccess*** is ***protected***, this implies the value can be changed by a method of the class, or child of the class. A “***Dependent***” property is calculated every time it is accessed and is not stored when the object is saved. This reduced memory requirements.

Table 2: List of Bayesian SSD Public Properties

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | Definition | Default Value | Attributes |
| **CreationDate** | Date the object was created | N/A | ***SetAccess*** = ***immutable***  ***GetAccess = public*** |
| **UserName** | Windows account id | N/A | ***SetAccess*** = ***immutable***  ***GetAccess = true*** |
| **ComputerName** | Computer Name | N/A | ***SetAccess*** = ***immutable***  ***GetAccess = public*** |
| **Tau** | Prior distribution variance parameter |  | ***SetAccess*** = ***public***  ***GetAccess*** = ***public*** |
| **Algorithm** | SSD design generator algorithm | “Jones” | ***Constant*** = ***true***  ***GetAccess*** = ***public*** |
| **M** | Total number of factors | N/A | ***SetAccess*** = ***immutable***  ***GetAccess = public*** |
| **NumInitDesigns** | Number of initial random designs to generate. |  | ***SetAccess*** = ***public***  ***GetAccess*** = ***public*** |
| **K** | Prior distribution information matrix | N/A | ***SetAccess*** = ***false***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **F** | Posterior distribution information matrix | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **Measure** | Determinant of the posterior distribution information matrix | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **V** | Posterior distribution variance-covariance matrix | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **ActiveFacs** | (1xM) vector of logical parameters. Set an element to true to indicate the corresponding factor is primary, and false to indicate it is a potential factor. | **false(1, obj.M)** | ***SetAccess*** = ***immutable***  ***GetAccess*** = ***protected*** |
| **NumScreenFacs** | Number of potential factors | **obj.M** | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **NumActiveFacs** | Number of primary factors. | 0 | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **Active** | Abbreviations for user defined ***primary*** factors. | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **Potential** | Abbreviations for user defined ***potential*** factors. | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **Name** | String array of long factor names. | “A”, “B”, … ,”Z”, “AB”, …, ”AZ”, “BA”… | ***SetAccess*** = ***protected***  ***GetAccess = public*** |
| **Abbreviation** | String array of factor abbreviations | “A”, “B”, … ,”Z”, “AA”, …, ”AZ”, “BA”… | ***SetAccess*** = ***protected***  ***GetAccess = public*** |
| **Lo** | Array of factor settings defining low setting; i.e. the (-1) setting in the design. Use -1 for categorical variables. | -1 | ***SetAccess*** = ***protected***  ***GetAccess = public*** |
| **Hi** | Array of factor settings defining high setting; i.e. the (+1) setting in the design. Use +1 for categorical variables. | +1 | ***SetAccess*** = ***protected***  ***GetAccess = public*** |
| **N** | Design size | **ceil(obj.M/2)** | ***SetAccess*** = ***protected***  ***GetAccess = public*** |
| **D** | Design matrix in natural units | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **X** | Model regression matrix | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **Es2** | criterion | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **MaxAbsCorr** | Maximum absolute correlation among factors | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **Kexch** | Number of columns exchanged during optimisation | 5 | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public*** |
| **MeanAbsCorr** | Average absolute correlation between columns | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **No** | Number of nonorthogonal column combinations | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **NumOrthCol** |  | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **C** | The “*c*” diagnostic | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |

## ***BayesianSSD*** Class Methods

The class has been designed to be as simple as possible to use and as such defines only 3 public methods. These are sufficient to both create designs and also compare them against existing alternatives from other sources, *e.g.* academic papers. The public methods are:

* The class constructor method, ***BayesianSSD***, creates an instance of the class and is discussed in section 4.3.1.
* Similarly, the ***designGenerator*** method creates a design using the Bayesian criterion, equation (6), as a cost function and the ***CWPW*** ***template***. Note, the initial design is randomly generated each time and so rerunning this method will give different results each time it is executed. This method is discussed in detail in section 4.3.2.
* Finally, the ***importScaledDesign*** method permits the user to define an object containing a design from an external source such as an academic paper. This is discussed in section 4.3.3.

### The Class Constructor Method: ***BayesianSSD***.

The class constructor instantiates, or creates, a class object. The command line syntax is:

**obj = BayesianSSD( M, ‘PARAM1’, VALUE1, …, ‘PARAM?’, VALUE? );**

Where **M** is the number of factors, ‘**PARAM#**’ is a MATLAB string defining the nature of the following, and corresponding **VALUE#**. The supported (‘**PARAM#**’, ‘**VALUE#**’) pairs are defined in Table 3.

Table 3: Valid (Parameter, Value) argument pairs for the BayesianSSD class constructor. Note all parameter strings are case insensitive. (Parameter, Value) pairs may be entered in any order.

|  |  |  |
| --- | --- | --- |
| **Parameter String** | **Value** | **Default Value** |
| Factor | String array of descriptive factor names; e.g. [“Battery Temperature”, “Electrolyte Type”, …] | “A”, “B”, …, “Z”, “AA”, …, “AZ”, “BA”, … |
| Abbreviation | String array of factor abbreviations. | “A”, “B”, …, “Z”, “AA”, …, “AZ”, “BA”, … |
| Lo | Array of factor settings defining low setting; i.e. the (-1) setting in the design. Use -1 for categorical variables. | **-ones(1, obj.M)** |
| Hi | Array of factor settings defining high setting; i.e. the (+1) setting in the design. Use +1 for categorical variables. | **ones(1, obj.M)** |
| Active | (1xM) vector of logical parameters. Set an element to true (or 1) to indicate the corresponding factor is primary, and false to indicate it is a potential factor. | **false(1, obj.M)** |

For example, imagine we wish to define an SSD for 10-factors, 3 of which are assumed to be primary factors and 7 potential. For the purposes of demonstration, we utilise “Active1”, …, “Active3” to denote the primary factors and “Screening1”, …, “Screening7” to denote the screening factors. Similarly, we use “A1”, …, “A3” for the primary factor abbreviations and “S1”, …, “S7” for the potential terms. An object representing these choices would be created by typing the following at the command line.

**Names = ["Screening1", "Active1", "Screening2", "Screening3", "Active2", …**

**"Screening4", "Screening5", "Active3", "Screening6", "Screening7"];**

**Abb = ["S1", "A1", "S2", "S3", "A2", "S4", "S5", "A3", "S6", "S7"];**

**obj = BayesianSSD( 10, “Abbreviation”, Abb, “Factor”, Names, “Active”, [0, 1, 0, 0, 1, 0, 0, 1, 0, 0])**

These commands yield the output:

*obj =* ***BayesianSSD*** *with properties:*

*Tau: 1*

*Algorithm: Jones*

*K: [11×11 double]*

*NumScreenFacs: 7*

*NumActiveFacs: 3*

*Active: ["A1" "A2" "A3"]*

*Potential: ["S1" "S2" "S3" "S4" "S5" "S6" "S7"]*

*M: 10*

*CreationDate: '09-Dec-2019'*

*UserName: 'markc'*

*ComputerName: 'DESKTOP-8LT60VB'*

*Lo: [10×1 double]*

*Hi: [10×1 double]*

*Name: ["A" "B" "C" "D" "E" "F" "G" "H" "I" "J"]*

*Abbreviation: ["S1" "A1" "S2" "S3" "A2" "S4" "S5" "A3" "S6" "S7"]*

*N: []*

*X: []*

*MaxAbsCorr: 0*

It is important to realise it is the contents of the **(“Active”, “Value”)** pair which define the factors as either “primary” or “potential”, not the content of the corresponding **Factor** or **Abbreviation** value pairs.

Note, in this example the constructor syntax applies the default values for the “Lo” and “Hi” properties, which are -1 and +1 respectively. These arguments define the mapping from the scaled design space, , to the application unit space, . All design calculations are carried out in the scaled units. Obviously, the test plan needs (or design) must be in the application or natural unit space.

### The ***desginGenerator*** Method.

The design generator method implements the columnwise, pairwise algorithm defined in section 3. This algorithm maximises the Bayesian criterion . This should be set at the command line prior to executing the ***desginGenerator*** method. For example, to set , type the following at the command line:

**obj.Tau = 2.0;**

The selected value of has little influence on the design generated. Consequently, following Jones et al, we recommend . However, if comparing designs using the Bayesian criterion, realise only comparisons between designs with the same -value are meaningful. For example, consider the design:

Table 4: Example SSD in 10-factors and 6-Runs



With , . If is now set to , then for the same design the Bayesian criterion is now . Similarly, the public property **NumInitDesigns** can be set at this juncture. From the perspective of generating designs with near optimal measures more frequently, this property should be set to a larger than the default of 1000; especially if the number of factors is relatively large. However, the price paid is the increased computational expense. For example, to set the number of initial number of random designs to be compared to 2500, type the following at the command line:

**obj.NumInitDesigns = 2500**

Regardless, to create an optimal design based on the Bayesian measure, the necessary command syntax is:

**obj = obj.designGenerator( N, Kexch, MaxIter, DispFlg );**

Where, **N** is the desired design size. This must be an even number. If an odd number is entered it is automatically rounded up or down, as appropriate, to the nearest even number. The **Kexch** argument denotes the number of columns to be exchanged. The default is 1. Argument **MaxIter** defines the maximum number of iterations allowed of the ***CWPW*** optimal design algorithm. If **DispFlg** is set to ***true***, then the CWPW algorithm outputs intermediate results. To suppress this behaviour set **DispFlg** to ***false***.

Continuing the example from section 4.3.1, to create a design in 6-runs, allowing up to 100 iterations of the optimal design algorithm, type the following at the command line:

**obj = obj.designGenerator( 6, 1, 100 );**

Note, in this example, the default setting of **DispFlg** = ***true*** is applied. The output from this is:

=====================================================

CWPW Exchange Algorithm

=====================================================

Iteration 1, Column Exchanged = 2, New Design Measure = 1.8686

Iteration 2, Column Exchanged = 6, New Design Measure = 1.8686

Iteration 3, Column Exchanged = 6, New Design Measure = 1.8686

In this instance the algorithm converges very quickly, probably because the initial random design is near the optimum, and the resulting design is:

Table 5: Example Optimal Bayesian SSD in 10-Factors and 6-Runs



The value of the Bayesian criterion for this design is . The class also provides some other design measures by which to evaluate the efficacy of the design. For example, the well-know criterion is 4.0 and the maximum absolute correlation between columns is 0.333.

Remember that the initial design is randomly generated. Consequently, if the ***designGenerator*** method is re-run, the results will be different. This behaviour is intentional and reflects the fact that the ***CWPW*** algorithm is not guaranteed to return the so-called exact D-optimal design. We will return to this point in section 4.4.

### The ***importScaledDesign*** Method

It can sometimes be useful to compare your current design with alternatives from the academic literature. With this in mind, we have supplied the ***importScaledDesign*** method. The command syntax is:

**obj = obj.importScaledDesign( ScaledDesign );**

Where **ScaledDesign** is an SSD in scaled units. Several checks are carried out on the **ScaledDesign** argument, and it is only accepted if the design passes all of them. In the event of a failure a suitable error message is displayed. The appropriate tests are:

* The number of factors must not change.
* The maximum number of runs must not exceed ; *i.e.* must be at least one less than the number of factors.
* Must be comprised of (-1) and (+1) elements only.
* Must be balanced.

## Example Design Using the ***makeSSD*** Class

As discussed in section 4.3.2, the initial design is intentionally random and, since the CWPW algorithm is not guaranteed to yield the exact D-optimal design, different designs will be produced each time the ***designGenerator*** method is executed. To mitigate this, it makes obvious sense to run the method several, perhaps, many times and choose the design associated with the largest value of the design measure. This is the concept behind the ***makeSSD*** class. We use the term *trials* to refer to the collection of designs generated by a ***makeSSD*** object.

The ***makeSSD*** class makes use of another object-oriented class association, namely *composition*. With composition, the *child class is stored as a property of the parent class*. In this way, the parent has full access to the child, but the child does not inherit the properties and methods of the parent. The main reason to use *composition* is that it allows you to reuse the child code without modelling an “*is-a”* association as you do by using [inheritance](https://stackify.com/oop-concept-inheritance/). That allows for the creation of cleaner application programmable interfaces and implies stronger *encapsulation.*

The class architecture is illustrated in Figure 3. Note that class ***makeSSD*** inherits from the abstract class ***trialSSD***. The abstract method ***isBest*** is implemented in ***makeSSD*** and abstracts “larger is better” or “smaller is better” design measures.

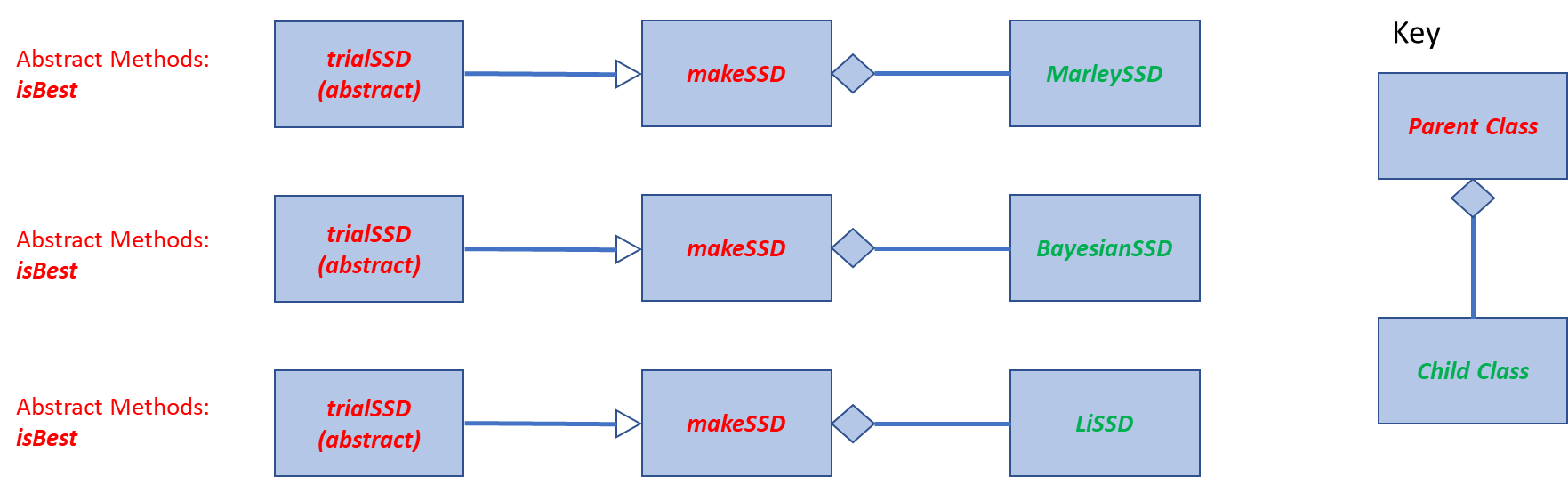


Figure 3: makeSSD class architecture, which employs composition. Consequently, any child SSD algorithm object can be fully encapsulated within the parent.

The public properties and methods defining the user interface are discussed in the following subsections. Section 4.4.1 defines the public properties, which may be displayed at the command line. Similarly, sections 4.4.2 through 4.4.4 describe the command syntax for public methods, which comprise the user interface.

### ***makeSSD*** Class Properties

A full list of public properties is provided in Table 6. The best design encountered during the trials is stored in property ***BestSSD***. This can be extracted and used like any ***BayesianSSD*** class object.

Table 6: makeSSD Class Public Properties

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | Definition | Default Value | Attributes |
| **NumTrials** | Number of design trials | 1000 | ***SetAccess*** = ***protected***  ***GetAccess = public*** |
| **DesignProperties** | Table of design properties for each trial. | N/A | ***SetAccess*** = ***protected***  ***GetAccess = public*** |
| **BestSSD** | Best design object from the trial set. This is a ***BayesianSSD*** object. | N/A | ***SetAccess*** = ***protected***  ***GetAccess = public*** |
| **M** | Total number of factors | N/A | ***SetAccess*** = ***private***  ***GetAccess = public*** |
| **Measure** | Determinant of the posterior distribution information matrix for the object stored in the **BestSSD** property. | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **N** | Design size | **ceil(obj.M/2)** | ***SetAccess*** = ***protected***  ***GetAccess = public*** |
| **Es2** | criterion for the object for the object stored in the **BestSSD** property. | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **MaxAbsCorr** | Maximum absolute correlation among factors for the object stored in the **BestSSD** property. | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **MeanAbsCorr** | Average absolute correlation between columns for the object stored in the **BestSSD** property. | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **No** | Number of nonorthogonal column combinations for the object stored in the **BestSSD** property. | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **NumOrthCol** |  | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |
| **C** | The -diagnostic | N/A | ***SetAccess*** = ***protected***  ***GetAccess*** = ***public***  ***Dependent*** = ***true*** |

### ***makeSSD*** Class Methods

As always, the class constructor instantiates, or creates, a class object. The following commands create an instance of the ***makeSSD*** class containing a ***BayesianSSD*** object with 16 factors, 3 of which are assumed active:

**SSDobject = BayesianSSD( 16, “Active”, [ true(1,3), false(1,13) ] );**

**SSDobj.NumInitDesigns = 5000;**

**obj = makeSSD( SSDobject )**

Executing these commands returns:

obj =

**makeSSD** with properties:

NumTrials: 1000

DesignProperties: [0×0 table]

BestSSD: [1×1 BayesianSSD]

M: 16

Algorithm: Jones

N: []

Measure: NaN

MaxAbsCorr: 0

MeanAbsCorr: NaN

No: 0

NumOrthCol: 136

### The ***generateDesign*** Method

This method conducts the design trials and selects the design corresponding to the most optimal design measure. The default number of trials is 1000. The necessary command line syntax is:

**obj = obj.generateDesign( NumTrials, “Param1”, Value1,…, 'Param#', Value# );**

Where **NumTrials** is the number of design trials to be undertaken and must be in the range . Similarly, “**Param?**” is a string variable defining the nature of the corresponding value argument, “**Value?**”. Valid **( “Param?”, “Value?” )** combinations are defined in Table 7.

Table 7: Valid (Parameter, Value) argument pairs for the **makeSSD** class, **generateDesign** method. Note all parameter strings are case insensitive. (Parameter, Value) pairs may be entered in any order.

|  |  |  |
| --- | --- | --- |
| **Parameter String** | **Value** | **Default Value** |
| N | The design size. Same for all trials. | **ceil( obj.M/2 )** |
| Kexch | Number of columns to exchange per iteration of the optimal design algorithm. | 5 |
| MaxIter | Maximum number of iterations for the optimal design algorithm per trial. | 100 |

For example, to execute 1500 trials, each with a design size of 12, exchanging 9 columns at each iteration of the optimal algorithm, with each trial allowed a maximum 500 iterations, the corresponding command is:

**obj = obj.generateDesign( 1500, "N", 12, "Kexch", 9, "MaxIter", 500 ) ;**

The output from this is:

====================================================

Making Trial Designs

====================================================

Iteration # 1, Best Design Updated. Measure = 4.03907

Iteration # 2, Best Design Updated. Measure = 4.10686

Iteration # 3, Best Design NOT Updated.

Iteration # 4, Best Design Updated. Measure = 4.11805

Iteration # 5, Best Design NOT Updated.

Iteration # 6, Best Design Updated. Measure = 4.12684

Iteration # 7, Best Design NOT Updated.

Iteration # 8, Best Design NOT Updated.

Iteration # 9, Best Design Updated. Measure = 4.17181

Iteration # 10, Best Design NOT Updated.

Iteration # 11, Best Design NOT Updated.

Iteration #1500, Best Design NOT Updated.

The entire process requires approximately 2.0 minutes to complete on my Dell Inspiron 7548, 64-bit, 2.4 GHz machine with 16 Giga Bytes of RAM.

To best design, in terms of the Bayesian measure, is stored in the **BestSSD** property. This can be assigned to any variable, **BestDesign** say, at the command line using: **BestDesign** **= obj.BestSSD**. Regardless, Table 8, illustrates the returned best design for the example cited. Upon completion, the code generates histograms of the primary design measures. These are generated by an automated call to the ***makeHistograms*** method, which plots histograms generated from the trial data for all the primary design measure. This is the subject of the next section.

Table 8: Example best design encountered during the trials. = 5.2, which corresponds to the “best” value given in [x].



### The ***makeHistograms*** Method

The ***makeHistograms*** method creates histograms of all the primary design measures: , , , and . Each is displayed in a separate figure. The command syntax is:

**obj.makeHistograms( NumBins )**

Where **NumBins** is the number of histogram bins. The number of default bins for the histogram is 21.

# Comparison with Other Methods

In this section, we begin by comparing the results from our code with those from a case study in the paper by Li and Wu [x], in 16 factors with 12 runs. Here, we use the ***makeSSD*** class to conduct 500 trials. For each trial, the number of initial random designs is set to 1000. Likewise, the prior variance is set to throughout. All factors are considered potential in nature and initially 5-columns are exchanged at each iteration. This is in line with Li and Wu’s recommendation, who consider a 5-column exchange a good compromise between design efficiency and computational speed. The total time required for the trials was just under 90 seconds.

Li and Wu’s design is shown in Table 9. Likewise, Table 10 presents the corresponding design returned by our code optimising the Bayesian criterion. The corresponding design measures for each are presented in Table 11.

Table 9: Li and Wu's SSD in 16-factors and 12-runs



Table 10: Results from the **makeSSD** class with 500 trials



Li and Wu claim that their criterion of 5.20 was the lowest known at that time. Utilising the Bayesian measure, with the ***makeSSD*** class, we obtain the same value. Equally, the Bayes measure, maximum absolute correlation between columns, , the mean absolute correlation, and the number of nonorthogonal columns, , are all identical – despite the designs being quite different. This suggests that multiple designs may possess the same value of .

Table 11: Design Evaluation Measures for Li and Wu's design and for the trials method using the **makeSSD** class. Values for the **makeSSD** class are in parentheses.

|  |  |
| --- | --- |
| **Design Evaluation Measure** | **Value** |
|  | 5.20 (5.20) |
|  | 4.97 (4.97) |
|  | 0.667 (0.667) |
|  | 0.103 (0.103) |
|  | 36 (36) |
| *c* | 0.968 (0.968) |

Figure 4 presents the histogram for the criterion from the trials. The optimal value is returned for 12 out of 500 trials, or just 2.4 percent of the time. This confirms the efficacy of the trials approach to generating an optimal SSD.



Figure 4: Histogram for the E(s2) Criterion for the Population of Trials. The optimal value of 5.2 is achieved just 12 out of 500 trials.



Figure 5: Histogram for the Criterion for the Population of Trials. The optimal value of 5.2 is achieved just 12 out of 500 trials. Designs exhibiting the optimal Bayes criterion also possess the optimal E(s2) measure.

Figure 5 presents the corresponding histogram for the Bayesian criterion, again for the trials data. Note those designs possessing the optimal criterion, also possess the maximum reported value of . This suggests that generating designs by maximising the Bayesian criterion is essentially equivalent to generating designs by minimising .

We repeated this case study with 3-primary and 13-potential factors. The design evaluation measures for this case are shown in Table 12. All the criterion, except , are degraded.

Table 12: Design Evaluation Measures for Li and Wu's design and for the trials method using the **makeSSD** class with 3-primary and 13-potential factors. Values for the **makeSSD** class are in parentheses.

|  |  |
| --- | --- |
| **Design Evaluation Measure** | **Value** |
|  | 5.20 (5.73) |
|  | 4.97 (4.88) |
|  | 0.667 (0.333) |
|  | 0.103 (0.119) |
|  | 36 (43) |

The reason the value of the Bayesian criterion decreases follows from the form of . We showed previously in section 2.1 that:

Consequently:

(22)

Consider the case where ; *i.e.* all factors are potential factors. Under these circumstances, and , which implies ; where is an matrix of ones. Hence:

(23)

Let denote the column vector of . As the design is balanced, . Consequently, . Thus:

(24)

Substituting (24) into (12) and noting , yields for only:

(25)

Now consider the case where , . Let denote the matrix:

(26)

We require to be positive definite, otherwise . Let , and . Let denote the Schur complement of in , such that:

(27)

Then is positive definite (*p.d.*) *iff* and are positive definite. If is to be invertible, then . Under these circumstances is positive definite. Let denote the singular values of and denote the singular values of . Similarly, let denote the singular values of . Then, using a result due to Loyka [[[17]](#endnote-18)]:

(28)

For a square matrix, the product of the singular values is the determinant. Hence, for to be p.d., the . The are clearly maximised as the , implying that , and in turn this implies the upper bound for . This explains the relationship illustrated in Figure 6 for the trials example given in section 4.4.3, where the Bayesian measure is seen to decrease almost linearly as the number of primary factors increases.



Figure 6: Reduction in the Bayesian measure as the number of primary factors increases for the design presented in Table 8.

# Future Enhancements

Below is a list of potential future enhancements to the package, which may be worth implementing later. These items are not in priority order. In every case some attempt has been made to indicate the necessary implementation effort associated with each one.

## Alternative Optimal Design Algorithms (Medium Effort)

Jones *et al* [iv] utilise the coordinate exchange algorithm, due to Meyer and Nachtsheim [[[18]](#endnote-19)]. This too is a candidate set free method. However, the Meyer and Nachtsheim algorithm is a row exchange algorithm, which is not guaranteed to preserve design balance. There is evidence in [iv] that lack of balance may lead to improved properties, particularly , but this assumes a practitioner is willing to favour one factor over another. There are scenarios in which such thinking apply. The two most obvious are:

1. The presence of a *control*. It makes obvious sense to have more occurrences of the control setting in in the design than potential alternatives. This is because we desire more precision with respect to the average response metric at the control setting – see Dunnett’s method in [xi, [[19]](#endnote-20)].
2. If one factor level is very expensive to manufacture and the other comparatively cheap. Under these circumstances, to reduce overall costs it makes sense to have more levels of the cheaper setting than the expensive one in a design column.

However, at this juncture, we do not feel this additional flexibility is worth the cost of implementation. Consequently, for the moment, the practitioner is limited to balanced designs only.

## Supporting Parallel Processing (Low Effort)

At each iteration. the -exchange elements of the algorithm presented occurs in a prescribed order. This lack of independence among exchanges effectively rules out parallel processing implementation. However, the trials themselves are independent from one another and this element of the algorithm could take advantage of parallel processing. If a computational advantage is achieved through the use of parallel processing, then this may permit the use of more trials, which should be beneficial - especially when is very large.

## Implement a GUI (Medium Effort)

It would be relatively straightforward to develop a GUI to configure the individual objects, execute the trials, display the design measures for the best design and the corresponding histograms for the trials. This would greatly simplify usage. The MathWorks recommend programmatic layout and call back definition for GUIs; this would require some effort to implement. It is also my experience that exposing a GUI to a use base subsequently generates a large list of “if only it could do this” requests. Consequently, several iterations of the GUI may be required to meet customer requirements. If implemented the GUI could be compiled as a Windows DLL, eliminating the need for native Matlab on user machines.

## Multi-Objective Optimisation (Large Effort)

We provided several alternative design measures by which any SSD can be assessed. Many of these measures are competing, implying that one can be improved at the expense of another. Instead of considering the value of these alternatives after generating the optimal design, another approach is to employ multi-objective optimisation (MOO) techniques and attempt to improve all simultaneously. Instead of returning a single design, MOO algorithms return estimates of the Pareto or non-dominating set. Among members of the Pareto set, competing objectives can only be traded and the analyst must decide the relevant importance of these in selecting a solution. MATLAB provides a multi-object Genetic Algorithm (MOGA) which could be employed to implement such methods, however systematic techniques to visualise high dimensional objective decision spaces and select solutions is still an area of active research.

# References

1. Meyer, R. K., Nachtsheim, C. J., The Coordinate Exchange Algorithm for Constructing Exact Optimal Experimental Designs, *Technometrics*, 1995, 37: 60-69. [↑](#endnote-ref-2)
2. Cheng, C. S., Steinberg, D.M., Sun, D.X., 1999. Minimum aberration and model robustness for two-level fractional factorial designs. *J. Roy. Statist. Soc. Ser. B* 61, 85–93. [↑](#endnote-ref-3)
3. Li, W., Nachtsheim, C. J., 2000. Model-robust factorial designs. *Technometrics,* 42, 345–352. [↑](#endnote-ref-4)
4. Jones, B., Lin, D. K. J., Nachtsheim, C. J., *Journal of Statistical Planning and Inference*, 2008, 138, 86-92. [↑](#endnote-ref-5)
5. Chaloner, K., Verdinelli, I., 1995. Bayesian experimental design: a review. *Statist. Sci.,* 10, 273–304. [↑](#endnote-ref-6)
6. DuMouchel,W., Jones, B., 1994.A simple Bayesian modification of D-optimal designs to reduce dependence on an assumed model. *Technometrics*, 36, 37–47. [↑](#endnote-ref-7)
7. Montgomery, D. C., Peck, E. A., Introduction to Linear Regression Analysis, John Wiley & Sons, Second Edition, 1992. [↑](#endnote-ref-8)
8. Searle, S. R., Matrix Alg*ebra* Useful for Statistics, John Wiley & Sons, First Edition, 1982. [↑](#endnote-ref-9)
9. [Householder, A. S.](https://en.wikipedia.org/wiki/Alston_Scott_Householder) (1958), *"*Unitary Triangularization of a Nonsymmetric Matrix*".*[*Journal of the ACM*](https://en.wikipedia.org/wiki/Journal_of_the_ACM)*.*5 (4): 339–342. [↑](#endnote-ref-10)
10. Li, W. W., Wu, C. F. J., Columnwise-Pairwise Algorithms With Applications to the Construction of Supersaturated Designs, *Technometrics*, 1997, 39, 171-179. [↑](#endnote-ref-11)
11. Dunnett, C. W., (1964.), New tables for multiple comparisons with a control, Biometrics, 20:482–491. [↑](#endnote-ref-12)
12. Van Schalkwyk, D. J., On the Design of Mixture Experiments, Unpublished PhD Thesis, University of London, 1971. [↑](#endnote-ref-13)
13. Booth, K. H. V., Cox, D. R., 1962, Some Systematic Supersaturated Designs, *Technometrics* 4, 489-495. [↑](#endnote-ref-14)
14. Press, W. H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P., Numerical Recipes in C, Cambridge University Press, Second Edition, 1992. [↑](#endnote-ref-15)
15. Nguyen, N.K., 1996. An algorithm approach to constructing supersaturated designs. *Technometrics* 38, 69–73. [↑](#endnote-ref-16)
16. Marley, C. J., Screening experiments using supersaturated designs with applications to industry, Ph.D. Thesis, University of Southampton, 2010. [↑](#endnote-ref-17)
17. Loyka, S., On Singular Value Inequalities of the Sum of Two Matrices, 1995, *arXiv*. [↑](#endnote-ref-18)
18. Meyer, R. K., Nachtsheim, C. J., The Coordinate Exchange Algorithm for Constructing Exact Optimal Designs, 1995, *Technometrics*, 37, 60-69. [↑](#endnote-ref-19)
19. Montgomery, D. C., Design and Analysis of Experiments, John Wiley and Sons, Third Edition, 1991. [↑](#endnote-ref-20)