Sampling Plans

Engineering design problems requiring the construction of a cheap-to-evaluate ‘surrogate’ model \( \hat{f} \) that emulates the expensive response of some black box \( f \) come in a variety of forms, but they can generally be distilled down to the following template.

Here \( f(x) \) is some continuous quality, cost or performance metric of a product or process defined by a \( k \)-vector of design variables \( x \in D \subset \mathbb{R}^k \). In what follows we shall refer to \( D \) as the design space or design domain. Beyond the assumption of continuity, the only insight we can gain into \( f \) is through discrete observations or samples \( \{x^{(i)} \rightarrow y^{(i)} = f(x^{(i)}) | i = 1, \ldots, n \} \). These are expensive to obtain and therefore must be used sparingly. The task is to use this sparse set of samples to construct an approximation \( \hat{f} \), which can then be used to make a cheap performance prediction for any design \( x \in D \).

Much of this book is made up of recipes for constructing \( \hat{f} \), given a set of samples. Excepting a few pathological cases, the mathematical formulations of these modelling approaches are well-posed, regardless of how the sampling plan \( X = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)} \} \) determines the spatial arrangement of the observations we have built them upon. Some models do require a minimum number \( n \) of data points but, once we have passed this threshold, we can use them to build an unequivocally defined surrogate.

However, a well-posed model does not necessarily generalize well, that is it may still be poor at predicting unseen data, and this feature does depend on the sampling plan \( X \). For example, measuring the performance of a design at the extreme values of its parameters may leave a great deal of interesting behaviour undiscovered, say, in the centre of the design space. Equally, spraying points liberally in certain parts of the inside of the domain, forcing the surrogate model to make far-reaching extrapolations elsewhere, may lead us to (false) global conclusions based on patchy, local knowledge of the objective landscape.

Of course, we do not always have a choice in the matter. We may be using data obtained by someone else for some other purpose or the available observations may come from a variety of external sources and we may not be able to add to them. The latter situation often occurs in conceptual design, where we wish to fit a model to performance data relating to existing, similar products. If the reader is only ever concerned with this type of modelling problem, he or she may skip the remainder of this chapter. However, if you have the possibility of
1.1 The ‘Curse of Dimensionality’ and How to Avoid It

It is intuitively obvious that the higher the number of design variables in a modelling problem, the more objective function measuring locations we need if we are to build a reasonably accurate predictor. What is more striking is just how many more: if a certain level of prediction accuracy is achieved by sampling a one-variable space in \( n \) locations, to achieve the same sample density in a \( k \)-dimensional space, \( n^k \) observations are required. To get a better feel for why this is often referred to as the curse of dimensionality, consider the following example.

Let us imagine that we would like to model the cost of a car tyre and we have a complex computational tyre design software that, given a set of geometrical variables, can, through a range of simulations, design a tyre and plan a manufacturing process for it, the latter model resulting in a cost estimate. For the sake of this example, let us assume that the analysis and design process takes one hour of computation per design. If we need a model of wheel diameter versus cost and have a computational budget of, say, ten hours, we can thus compute ten cost values at diameter values ranging from the smallest to the largest car in the manufacturer’s range. Ten simulations should give us a reasonably accurate predictor, even considering that the response can be highly nonlinear (for example due to different types of tools being needed for different sizes, nonlinearity of performance requirements, etc.). What happens, however, if we decide to refine the model by including other variables, say, tread width, groove spacing, sidewall height, flexing area thickness, shoulder thickness, bead seat diameter and liner thickness? We now have eight parameters, which, assuming the same sampling density as on the wheel diameter, means that our computational budget requirement jumps to \( 10^8 \) runs. This will take almost 11 416 years!

There are two important conclusions here. Firstly, evaluating the objective function for every possible combination of every possible design variable value can become a very expensive undertaking. Statisticians refer to this type of scenario as a full factorial experiment.

The second conclusion we can draw is that the number of design variables has a massive impact on the number of experiments required. It is therefore imperative that we minimize this at the outset. The question is, how can we tell which variables can be left out of a design study, that is, which variables do not have a significant effect on the objective function? More to the point, how can we answer the above question with a minimum number of runs of the (usually expensive) simulation? We will discuss this shortly, but first we need to make a few general points about physical and computational experiments, the two sources that may be used to obtain the objective function.

1.2 Physical versus Computational Experiments

The results of physical experiments are almost always subject to experimental error. These departures from the ‘true’ result come from three main sources:
• **human error**, that is error introduced simply by the experimenter making a mistake;
• **systematic error**, due to a flaw in the philosophy of the experiment that adds a consistent bias to the result;
• **random error**, which is due to measurement inaccuracies inherent to the instruments being used.

The key concept that differentiates between the last two items in this list is *repeatability*. If there is a systematic component in the experimental error, this will have the same value each time we repeat the experiment. The random error, however, will be different every time and, given enough experiments, it will take both positive and negative values.

Computational experiments are also subject to experimental error, resulting from:

• **human error**, ‘bugs’ in the analysis code, incorrectly entered boundary conditions in the solution of a partial differential equation, etc., and
• **systematic error**. For example, an inviscid mathematical model of the viscous flow around a body (an approximation sometimes made for computing time saving purposes) will consistently underestimate the drag forces acting on the body. Another example is the error caused by the inherently finite resolution of the numerical modelling process (e.g. errors caused by insufficient mesh resolution in a finite element solve). While this type of error can lead to underestimates or overestimates, it will do so in *exactly the same way* if we repeat the experiment.

The difference, therefore, compared to physical experiments is that computational experiments are not affected by random error – they are *deterministic*.

We dwell on this seemingly academic point here for three good reasons. Firstly, it is germane to a question of terminology. Physical experimentalists often use the term ‘noise’, referring to the *random error* that corrupts their experiments. Somewhat confusingly, though, ‘noise’ often crops up in the computational experiments literature as well, referring to *systematic error* (hardly ever stating this explicitly, but it must do, as computers do not make random errors!). This is not an especially pernicious usage, as long as both author and reader understand what it refers to. To that end, we shall, throughout this book, differentiate between the two meanings by putting a pair of inverted commas around ‘noise’ when it refers to the systematic errors of computer experiments and leaving them out when we are talking about the random noise of physical experiments or about both types.

Beyond the semantics, it will also be important for the reader to be aware of the differences between the various types of error when, later on, we tackle Gaussian process based approximation techniques. The reason is that the statistical apparatus behind these methods requires a fictional ‘physicalization’ of computer experiments: we will view the outputs (results) of computer experiments, known to be *deterministic* values, as *realizations of a stochastic process*. This is merely to facilitate the mathematical process and one of the purposes of this section is to dispel, well in advance, any confusion this artifice may cause.

The final reason for insisting on the differences between the two types of experiments is to explain why their respective experimental design techniques are so different. A vast literature has been written on devising screening strategies and sampling plans for physical experiments, which are aimed, among other things, at mitigating the effects of the random error that affects the responses. In principle, this is done by replicating experiments – a pointless exercise, of course, when a deterministic computer code provides the data upon which the approximation will be built.
1.3 Designing Preliminary Experiments (Screening)

We saw earlier just how important it is to minimize the number of design variables \( x_1, x_2, \ldots, x_k \) before we attempt to model the objective function \( f \). But how do we achieve this screening, as we shall call this process in what follows, without compromising the relevance of the analysis?

If \( f \) is at least once differentiable over the design domain \( D \) with respect to each \( x \), \( \partial f/\partial x_i |_{x} \) is a useful criterion for establishing a taxonomy of design variables. Namely:

- if \( \partial f/\partial x_i |_{x} = 0, \forall x \in D \), the variable \( x_i \) can safely be neglected,
- if \( \partial f/\partial x_i |_{x} = g(x_i), \forall x \in D \), where \( g(x_i) \neq \text{constant} \), \( f \) is nonlinear in \( x_i \),
- if \( \partial f/\partial x_i |_{x} = g(x_i), \forall x \in D \), where \( g(x_i, x_j, \ldots) \neq \text{constant} \), \( f \) is nonlinear in \( x_i \) and involved in interactions with \( x_j, \ldots \).

The above classification is merely a statement of terminology, as in practice we have no way of measuring \( \partial f/\partial x_i |_{x} \) across the entire design space. Even a reasonable estimate is a tall order considering that the budget available for the screening study is generally very limited. Incidentally, there is no hard and fast rule as to what percentage of the available time should be spent on screening the variables, as this is largely problem-dependent. If we expect many variables to be inactive, a thorough screening study has the potential of enhancing the accuracy of the subsequent model considerably (due to its reduced dimensionality). If, however, there is an (engineering) reason to believe that most variables have a considerable impact on the objective, it is advisable to focus efforts on the modelling itself.

A great deal has been written about sampling plans and modelling methods specifically aimed at input variable screening (Jones et al. (1998)). Their working principles vary according to the assumptions they make about the objective function and the variables. Here we concentrate on an algorithm described by Morris (1991) because the only assumption it makes is that the objective function is deterministic (a feature shared by most computational models).

1.3.1 Estimating the Distribution of Elementary Effects

In order to simplify the presentation of what follows we make, without loss of generality, the assumption that the design space \( D = [0, 1]^k \); that is we normalize all variable into the unit cube. We shall adhere to this convention for the rest of the book and we strongly urge the reader to do likewise in implementing any algorithms described here, as, in addition to yielding clearer mathematics in some cases, this step safeguards against scaling issues.

Before proceeding with the description of the Morris algorithm we need to define an important statistical concept. Let us restrict our design space \( D \) to a \( k \)-dimensional, \( p \)-level full factorial grid, that is \( x_i \in \{0, 1/(p-1), 2/(p-1), \ldots, 1\} \), for \( i = 1, \ldots, k \). For a given baseline value \( x \in D \), let \( d_i(x) \) denote the elementary effect of \( x_i \), where

\[
d_i(x) = \frac{y(x_1, x_2, \ldots, x_{i-1}, x_i + \Delta, x_{i+1}, \ldots, x_k) - y(x)}{\Delta},
\]

(1.1)

where \( \Delta = \xi/(p-1), \xi \in \mathbb{N}^* \) and \( x \in D \) such that its components \( x_i \leq 1 - \Delta \).
Morris’s method aims to estimate the parameters of the distribution of elementary effects associated with each variable, the principle being that a large measure of central tendency indicates a variable with an important influence on the objective function across the design space and a large measure of spread indicates a variable involved in interactions and/or in terms of which \( f \) is nonlinear. In practice, we estimate the sample mean and the sample standard deviation of a set of \( d_i(x) \) values calculated in different parts of the design space.

Clearly, it is desirable to generate the preliminary sampling plan \( X \) in such a way that each evaluation of the objective function \( f \) will participate in the calculation of two elementary effects (instead of just one, if we were to pick, naively, a random spread of baseline \( x \)'s and then to add \( \Delta \) to one of their variables). Also, the sampling plan should give us a certain number (say, \( r \)) elementary effects for each variable, independently drawn with replacement. The reader interested in a thorough discussion of how to obtain such an \( X \) is invited to read Morris’s original paper (Morris, 1991) – here we limit ourselves to a description of the process itself.

Let \( B \) denote a \( k + 1 \times k \) sampling matrix of 0s and 1s with the property that for every column \( i = 1, 2, \ldots, k \) there are two rows of \( B \) that differ only in their \( i \)th entries (we shall give an example of such a matrix in the \textit{MATLAB}® implementation of the method). We then compute a random orientation of \( B \), denoted by \( B^* \):

\[
B^* = (I_{k+1,1}x^* + (\Delta/2)[(2B - I_{k+1,k})D^* + I_{k+1,k}])P^*,
\]

where \( D^* \) is a \( k \)-dimensional diagonal matrix, where each element on the diagonal is either +1 or −1 with equal probability, \( I \) is a matrix of 1s, \( x^* \) is a randomly chosen point in our discretized, \( p \)-level design space (limited around the edges by \( \Delta \), as discussed previously) and \( P^* \) is a \( k \times k \) random permutation matrix in which each column contains one element equal to 1 and all others equal to 0 and no two columns have 1s in the same position. Here is a \textit{MATLAB} implementation of Equation (1.2):

```matlab
function Bstar=randorient (k, p, xi)

% Generates a random orientation for a screening matrix

% Inputs:
% k – number of design variables
% p – number of discrete levels along each dimension
% xi – elementary effect step length factor

% Output:
% Bstar – random orientation matrix

% Step length
Delta=xi/(p-1);
m=k+1;

% A truncated p – level grid in one dimension
xs=(0:1/(p-1):1-Delta);
xs1=length(xs);

(continued)```
To obtain $r$ elementary effects for each variable, the screening plan is built from $r$ random orientations:

$$X = \begin{bmatrix} B_1^* \\ B_2^* \\ \vdots \\ B_r^* \end{bmatrix},$$

(1.3)

or in MATLAB:

```matlab
function X=screeningplan(k, p, xi, r)
% Generates a Morris screening plan with a specified number of
% elementary effects for each variable.
%
% Inputs:
% k – number of design variables
% p – number of discrete levels along each dimension
% xi – elementary effect step length factor
% r – number of random orientations (=number of elementary
% effects per variable).
%
% Output:
% X – screening plan built within a $[0,1]^k$ box

X=[];
for i=1:r
    X=[X; randorient(k,p,xi)];
end
```
We then compute the value of $f$ for each row of $X$; in what follows we shall store these objective function values in the $r(k + 1) \times 1$ column vector $t$. Taking one random orientation at a time, the adjacent rows of the screening plan and the corresponding function values from $t$ can be inserted into Equation (1.1) to obtain $k$ elementary effects (one for each variable).

Once a sample of $r$ elementary effects has been collected for each variable, the sample means and sample standard deviations of these can be computed and represented on the same chart for comparison purposes. Here is how `screeningplot.m` accomplishes this:

```matlab
function screeningplot(X, Objhandle, Range, xi, p, Labels)
    k=size(X,2);
    r=size(X,1)/k-1;
    for i=1:size(X,1)
        X(i,:)=Range(1,:)+X(i,:).*(Range(2,:)-Range(1,:));
        t(i)=feval(Objhandle,X(i,:));
    end
    for i=1:r
        for j=(i-1)*k+1:(i-1)*k+k
            F(find(X(j,:)-X(j+1,:)^2)=0),i)=(t(j+1)-t(j))/(xi/(p-1));
        end
    end
    for i=1:k
        ssd(i)=std(F(i,:));
        sm(i)=mean(F(i,:));
    end
    figure, hold on
    for i=1:k
        text(sm(i),ssd(i),Labels(i),'FontSize',25)
    end
    axis([min(sm) max(sm) min(ssd) max(ssd)]); xlabel(’Sample_means’) ylabel(’Sample_standard_deviations’)
```
Before illustrating the process by means of an engineering design example, it is worth mentioning two scenarios, where the deployment of the algorithm described above requires special care.

Firstly, if the dimensionality $k$ of the space is relatively low and we can afford a large $r$, one should keep in mind the increased probability of the same design cropping up twice in $X$. If the responses at the sample points are deterministic, there is, of course, no point in repeating the evaluation. This issue does not occur especially often, as large numbers of elementary effects are generally needed when screening spaces with high dimensionalities.

Secondly, numerical simulation codes sometimes fail to return a sensible (or, indeed, any) result, due to meshing errors, the failure of a partial differential equation solution process to converge, etc. From a screening point of view this is significant because an entire random orientation $B^*$ is compromised if the objective function computation fails for one of the points therein.

### A ten-variable weight function

Let us consider the following analytical expression (implemented in `liftsurfww.m`) used as a conceptual level estimate\(^1\) of the weight of a light aircraft wing:

$$W = 0.036 S_w^{0.758} W_{fw}^{0.0035} \left( \frac{A}{\cos^2 \Lambda} \right)^{0.6} q^{0.006} A^{0.04} \left( \frac{100 tc}{\cos \Lambda} \right)^{-0.3} (N_z W_{dg})^{0.49} + S_w W_p. \tag{1.4}$$

Table 1.1 contains a nomenclature of the symbols used in Equation (1.4), as well as a baseline set of values, roughly representative of a Cessna C172 Skyhawk aircraft and a somewhat arbitrarily chosen range for each variable. These baseline values and the ranges were used to generate a filled contour plot of the weight function (see Figure 1.1) by varying the inputs pairwise and keeping the remaining variables at the baseline value.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Baseline</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_w$</td>
<td>Wing area ($ft^2$)</td>
<td>174</td>
<td>150</td>
<td>200</td>
</tr>
<tr>
<td>$W_{fw}$</td>
<td>Weight of fuel in the wing (lb)</td>
<td>252</td>
<td>220</td>
<td>300</td>
</tr>
<tr>
<td>$A$</td>
<td>Aspect ratio</td>
<td>7.52</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>$\Lambda$</td>
<td>Quarter-chord sweep (deg)</td>
<td>0</td>
<td>–10</td>
<td>10</td>
</tr>
<tr>
<td>$q$</td>
<td>Dynamic pressure at cruise (lb/ft$^2$)</td>
<td>34</td>
<td>16</td>
<td>45</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Taper ratio</td>
<td>0.672</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>$tc$</td>
<td>Aerofoil thickness to chord ratio</td>
<td>0.12</td>
<td>0.08</td>
<td>0.18</td>
</tr>
<tr>
<td>$N_z$</td>
<td>Ultimate load factor</td>
<td>3.8</td>
<td>2.5</td>
<td>6</td>
</tr>
<tr>
<td>$W_{dg}$</td>
<td>Flight design gross weight (lb)</td>
<td>2000</td>
<td>1700</td>
<td>2500</td>
</tr>
<tr>
<td>$W_p$</td>
<td>Paint weight (lb/ft$^2$)</td>
<td>0.064</td>
<td>0.025</td>
<td>0.08</td>
</tr>
</tbody>
</table>

\(^1\) Such equations are generally derived by fitting curves to existing aircraft data. This particular formula has been adapted from Raymer’s excellent aircraft conceptual design text (Raymer, 2006).
So what does the plot reveal from the point of view of variable activity? As expected, for example, the weight per unit surface area of the paint ($W_p$) does not make much of an impact on the shape of the surface, whereas the load factor $N_z$ (which determines the magnitude of the maximum aerodynamic load on the wing) is clearly very active and it is involved in interactions with other variables. A classic example is the interaction with the aspect ratio $A$: the red area in the top right-hand corner of the weight versus $A$ and $N_z$ indicates a heavy wing for high aspect ratios and large $g$-forces (this is the reason why highly manoeuvrable fighter jets cannot have very efficient, glider-like wings).

Of interest to us here, however, is how much of all this would we have guessed simply from a cheap screening study, without an understanding of the engineering significance of the variables involved (which is quite often the case in engineering design) and without the ability to compute such a tile plot (which is almost always the case in engineering design – after all, if the objective $f$ was so cheap to compute, we would not be thinking about surrogate modelling anyway).

So what does Figure 1.2, depicting the results of an $r = 5$ screening study, reveal? The first observation we can make is that there is a clearly defined group of variables clustered around the origin – recall that a small measure of central tendency is a feature of inputs with little impact on the objective function. Indeed, we find the weight of the paint here, as
expected, as well as the dynamic pressure (meaning that it does not make a big difference in wing weight where we are in our chosen range of dynamic pressures – with the cruising speed specified, this can be viewed in terms of a cruising altitude range). The same reasoning applies (and is confirmed by Plate I) to the taper ratio and the quarter-chord sweep.²

Although still close to the zero mean, the variable with the largest central tendency within this group is the fuel weight $W_{fw}$. Its sample of elementary effects has a very low standard deviation and a mean slightly larger than the rest of the group, indicating that it is more important than them but is not involved in interactions. The plot indicates $A$ and $tc$ having similar importance, but having a nonlinear/interactive effect (as seen from their high standard deviation values).

Finally and unsurprisingly, a large (absolute) central tendency and large measure of spread point to $W_{dg}$, $S_w$ and $N_z$ having the most significant impact on wing weight. Of course, aircraft designers know that the overall weight of the aircraft and the wing area must be kept to a minimum (the latter usually dictated by constraints such as required stall speed, landing distance, turn rate, etc.) and that a requirement for high $N_z$ will translate into a need for sturdy, heavy wings. In fact, this is precisely why we have used such a well-understood function here to illustrate the workings of the screening algorithm. Screening will be done in anger, however, when no such prior knowledge is available and the identification of the important variables can merely rely on the objective function as a black box.

The script `wing.m` will run the example discussed above and will produce Plate I and a scatter plot similar to Figure 1.2 (`MATLAB` will generate a slightly different screening plan each time, as this comprises random orientations of the sampling matrix $B$).

---

2 Large variations in the sweep angle would make a significant difference. Here, however, we are looking at a small range of values (−10 to +10 degrees) typical of light, low speed aircraft.
We shall return briefly to the issue of establishing the order of importance (or activity) of the inputs of the objective function in the section about Kriging models. For now, let us look at the next step of the modelling process. With the active variables identified (either through engineering judgement or through a systematic screening study) we can now design the main sampling plan in the space defined by these variables. This will form the basis of the data that the model of the objective will be built upon.3

1.4 Designing a Sampling Plan

1.4.1 Stratification

A feature shared by all of the approximation models discussed in this book is that they are more accurate in the vicinity of the points where we have evaluated the objective function. In later chapters we will delve into the laws that quantify our decaying trust in the model as we move away from a known, sampled point, but for the purposes of the present discussion we shall merely draw the intuitive conclusion that a uniform level of model accuracy throughout the design space requires a uniform spread of points. A sampling plan possessing this feature is said to be space-filling.

The most straightforward way of sampling a design space in a uniform fashion is by means of a rectangular grid of points. This is the full factorial sampling technique referred to in the section about the curse of dimensionality.

Here is the simplified version of a MATLAB function that will sample the unit hypercube at all levels in all dimensions, with the k-vector q containing the number of points required along each dimension. The variable Edges specifies whether we want the points to be equally spaced from edge to edge (Edges=1) or we want them to be in the centres of \( n = q_1 \times q_2 \times \cdots \times q_k \) bins filling the unit hypercube (for any other value of Edges).

\[
\text{function } X = \text{fullfactorial}(q, \text{Edges})
\]

% Generates a full factorial sampling plan in the unit cube

% Inputs:
% q – k – vector containing the number of points along each
dimension
% Edges – if Edges=1 the points will be equally spaced from
depth to edge (default), otherwise they will be in
the centres of \( n = q(1) \times q(2) \times \cdots \times q(k) \) bins filling
the unit cube.

(continued)

3 There may be some cases where the runs performed for screening purposes may be recycled at the actual model fitting step, in particular when the objective is very expensive. For example, if some of the variables turn out not to have any impact at all, these runs can form part of the reduced dimensionality sampling plan. Of course, life is rarely as black and white as this and the ignored variables will have had some effect. In this case, a judgement will have to be made regarding the trade-off between losing some expensive runs versus introducing some additional noise into the model fitting data.
% Output:
%   X – full factorial sampling plan

if nargin < 2, Edges=1; end

if min(q) < 2
    error('You must have at least two points per dimension.');
end

% Total number of points in the sampling plan
n=prod(q);

% Number of dimensions
k=length(q);

% Pre-allocate memory for the sampling plan
X=zeros(n,k);

% Additional phantom element
q(k+1)=1;

for j=1:k
    if Edges==1
        one_d_slice =(0:1/(q(j)-1):1);
    else
        one_d_slice =(1/q(j)/2:1/q(j):1);
    end
    column=[];
    while length(column) < n
        for l=1:q(j)
            column=[column; ones(prod(q(j+1:k)),1)*···
                      one_d_slice(l)]
        end
    end
    X(:,j)=column;
end

For example, `fullfactorial([3 4 5],1)` will produce the three-dimensional sampling plan shown in Figure 1.3. Clearly, such a design will satisfy the uniformity criterion, but it has two major flaws.

Firstly, it is only defined for designs of certain sizes, those that can be written as products of the numbers of levels for each dimension, that is \( n = q_1 \times q_2 \times \cdots \times q_k \).\(^4\) Secondly, when projected on to the axes, sets of points will overlap and it can be argued that the sampling

\(^4\) Random sampling plans provide an alternative, which clearly can be created for any number of design points and they do make sense in some cases. For example, if we sample the design variables according to some distribution, we can assess how the objective values are distributed (Santner et al., 2003), a scenario typical of robustness studies. In most cases, however, the spread of a random plan can be quite uneven and therefore not space-filling (especially for small values of \( n \)).
Sampling Plans

Figure 1.3. Example of a three-dimensional full factorial sampling plan.

of any individual variable could be improved by making sure that these projections are as uniform as possible.

This can be done by splitting the range of that variable into a relatively large number of equal sized bins and generating equal sized random subsamples within these bins. This approach is known as stratified random sampling. A natural development of this idea is to generate a sampling plan that is stratified on all of its dimensions. The technique commonly used to achieve this is known as Latin hypercube sampling.

1.4.2 Latin Squares and Random Latin Hypercubes

As we have seen, stratification of sampling plans aims to generate points whose projections onto the variable axes are uniform. Before we look at generic techniques for building such plans, it is worth considering the case of discrete valued variables in two dimensions. Such uniform projection plans can be generated quite readily: if \( n \) designs are required, an \( n \times n \) square is built by filling every column and every line with a permutation of \( \{1, 2, \ldots, n\} \), that is every number must only appear once in every column and every line. For example, for \( n = 4 \), a Latin square (for this is what such plans are usually known as) might look like this:

\[
\begin{array}{cccc}
2 & 1 & 3 & 4 \\
3 & 2 & 4 & 1 \\
1 & 4 & 2 & 3 \\
4 & 3 & 1 & 2 \\
\end{array}
\]
We have highlighted the 1’s to illustrate the point about the uniform projection idea but, of course, we could have chosen 2, 3 or 4 just as well. Also, this is just one, arbitrarily chosen four-point Latin square – we could equally have picked any of the other 575 possible arrangements. Incidentally, the number of distinct Latin squares increases rather sharply with \( n \); for example, there are 108 776 032 459 082 956 800 Latin squares of order eight! (It is left as an exercise for the reader to check this.)

Building a *Latin hypercube*, that is the multidimensional extension of this, can be done in a similar way, by splitting the design space into equal sized hypercubes (bins) and placing points in the bins (one in each), making sure that from each occupied bin we could exit the design space along any direction parallel with any of the axes without encountering any other occupied bins. This is illustrated for three dimensions in Figure 1.4.

We achieve this using the following technique. If \( \mathbf{X} \) denotes the \( n \times k \) matrix in which we wish to build our sampling plan of \( n \) points in \( k \) dimensions (each row represents a point) we begin by filling up \( \mathbf{X} \) with random permutations of \( \{1, 2, \ldots, n\} \) in each column and we normalize our plan into the \([0, 1]^k\) box. The following code performs this in MATLAB.

---

**Figure 1.4.** Three-variable, ten-point Latin hypercube sampling plan shown in three dimensions (top left), along with its two-dimensional projections. All ten points are visible on each of the latter, while each row and column of bins contains exactly one point.
function X=rlh(n, k, Edges)
% Generates a random Latin hypercube within the \([0,1]^k\) hypercube.
% Inputs:
% n – desired number of points
% k – number of design variables (dimensions)
% Edges – if Edges=1 the extreme bins will have their centres
% on the edges of the domain, otherwise the bins will
% be entirely contained within the domain (default
% setting).
% Output:
% X – Latin hypercube sampling plan of \(n\) points in \(k\)
% dimensions.
if nargin<3
    Edges=0;
end
% Pre – allocate memory
X=zeros(n,k);
for i=1:k
    X(:,i)=randperm(n)’;
end
if Edges==1
    X=(X-1)/(n-1);
else
    X=(X-0.5)/n;
end

The above recipe will thus yield a randomized sampling plan, whose projections on to
the axes are uniformly spread (multidimensional stratification). This, however, does not
guarantee that the plan will be space-filling. After all, placing all of the points on the main
diagonal of the design space will fulfil the multidimensional stratification criterion, but,
intuitively, will not fill the available space uniformly. We therefore need some measure of
uniformity that will allow us to distinguish between ‘good’ and ‘bad’ Latin hypercubes, even
in cases that are not as clear-cut as the diagonal example given above.

1.4.3 Space-filling Latin Hypercubes
One of the most widely-used measures to evaluate the uniformity (‘space-fillingness’) of a
sampling plan is the maximin metric introduced by Johnson et al. (1990). The criterion based
on this may be defined as follows.

Let \(d_1, d_2, \ldots, d_m\) be the list of the unique values of distances between all possible pairs
of points in a sampling plan \(\mathbf{X}\), sorted in ascending order. Further, let \(J_1, J_2, \ldots, J_m\) be
defined such that \(J_j\) is the number of pairs of points in \(\mathbf{X}\) separated by the distance \(d_j\).
Definition 1.1. We will call $X$ a maximin plan among all available plans if it maximizes $d_1$ and, among plans for which this is true, minimizes $J_1$.

Clearly, this definition could be applied to any set of sampling plans, but, since we would like to keep the appealing stratification properties of Latin hypercubes, we restrict our scope to that class of designs. Nonetheless, even across this narrower domain, Definition 1.1 might still yield several maximin designs. Therefore we shall use the more complete ‘tie-breaker’ definition of Morris and Mitchell (1995).

Definition 1.2. We call $X$ the maximin plan among all available plans if it maximizes $d_1$, among plans for which this is true, minimizes $J_1$, among plans for which this is true, maximizes $d_2$, among plans for which this is true, minimizes $J_2$, . . . , minimizes $J_m$.\footnote{To be completely rigorous, Definition 1.1 has been shown by Johnson et al. (1990) to be equivalent to the so-called D-optimality criterion used in linear regression, whereas Definition 1.2 is simply intuitively appealing and practically more useful. As we are considering sampling plans that do not make any assumptions relating to model structure, we shall use the latter.}

Before proceeding further, we need to clarify what we mean by ‘distance’ in the above definitions. The metric most widely used is the $p$-norm of the space:

$$d_p(x^{(i_1)}, x^{(i_2)}) = \left( \sum_{j=1}^{k} |x_j^{(i_1)} - x_j^{(i_2)}|^p \right)^{1/p}. \quad (1.5)$$

For $p = 1$ this is the rectangular distance (sometimes also referred to as the Manhattan norm in reference to the district’s grid-like layout) and $p = 2$ yields the Euclidean norm. There is little evidence in the literature of one being more suitable than the other for sampling plan evaluation if no assumptions are made regarding the structure of the model to be fitted, though it must be noted that the rectangular distance is considerably cheaper to compute. This can be quite significant, especially if large sampling plans are to be evaluated.

And now, onto the practicalities of working with Definition 1.2 in a computational context. First, we need to build the vectors $d_1, d_2, \ldots, d_m$ and $J_1, J_2, \ldots, J_m$. Here is a MATLAB implementation of a function that accomplishes this task:

```matlab
function [J, distinct_d] = jd(X, p)
    % Computes the distances between all pairs of points in a sampling
    % plan X using the p–norm, sorts them in ascending order and
    % removes multiple occurrences.
    %
    % Inputs:
    %   X – sampling plan being evaluated
    %   p – distance norm (p=1 rectangular – default, p=2 Euclidean)
    %
    % Outputs:
    %   J – multiplicity array (that is, the number of pairs
    %        separated by each distance value).
    %   distinct_d – list of distinct distance values
```

(continued)
if ~exist('p','var')
    p=1;
end

% Number of points in the sampling plan
n=size(X,1);

% Compute the distances between all pairs of points
d=zeros(1,n*(n-1)/2);

for i=1:n-1
    for j=i+1:n
        % Distance metric: p–norm
        d((i-1)*n-(i-1)*i/2+j-i)=norm(X(i,:)-X(j,:),p);
    end
end

% Remove multiple occurrences
distinct_d=unique(d);

% Pre-allocate memory for J
J=zeros(size(distinct_d));

% Generate multiplicity array
for i=1:length(distinct_d)
    % J(i) will contain the number of pairs separated
    % by the distance distinct_d(i)
    J(i)=sum(ismember(d,distinct_d(i)));
end

A very time-consuming part of this calculation is the creation of the vector that contains the distances between all possible pairs of points. This becomes especially significant for large sampling plans (for example, in the case of a 1000-point plan almost half a million calculations are required). Therefore pre-allocation of the memory is essential, which leaves us with the somewhat roundabout way of computing the indices of d (as opposed to appending each new element to d, which would require the use of expensive dynamic memory allocation).

We now need to implement Definition 1.2 itself. Since finding the most space-filling design will require pairwise comparisons, we ‘divide and conquer’ the problem by simplifying it to the task of choosing the better of two sampling plans. The function \texttt{mm(X1,X2,p)} accomplishes this, returning the index of the more space-filling of the two designs and 0 if they are equal (the $p$-norm is used to compute the distances):

```matlab
function Mmplan=mm(X1,X2,p)
% Given two sampling plans chooses the one with the better
% space–filling properties (as per the Morris–Mitchell criterion).

% Inputs:

(continued)
% X1, X2 – the two sampling plans
% p – the distance metric to be used (p=1 rectangular –
%         default, p=2 Euclidean)
%
% Outputs:
%       Mmplan – if Mmplan=0, identical plans or equally space–
%                 filling, if Mmplan=1, X1 is more space–filling,
%                 if Mmplan=2, X2 is more space–filling.

if ~exist(‘p’,’var’)
    p=1;
end

if sortrows(X1)==sortrows(X2)
    % If the two matrices contain the same points
    Mmplan=0;
else
    % Calculate the distance and multiplicity arrays
    [J1,d1]=jd(X1,p); m1=length(d1);
    [J2,d2]=jd(X2,p); m2=length(d2);

    % Blend the distance and multiplicity arrays together for
    % comparison according to Definition 1.2B. Note the different
    % signs – we are maximizing the d’s and minimizing the J’s.
    V1(1:2:2*m1-1)=d1;
    V1(2:2:2*m1)=-J1;
    V2(1:2:2*m2-1)=d2;
    V2(2:2:2*m2)=-J2;

    % The longer vector can be trimmed down to the length of the
    % shorter one
    m =min(m1,m2);
    V1=V1(1:m); V2=V2(1:m);

    % Generate vector c such that c(i)=1 if V1(i)>V2(i), c(i)=2 if
    % V1(i)<V2(i) and c(i)=0 otherwise
    c=(V1>V2)+2*(V1<V2);

    % If the plans are not identical but have the same space–filling
    % properties
    if sum(c) ==0
        Mmplan=0;
    else
        % The more space–filling design (mmplan)
        % is the first non–zero element of c
        i=1;
        while c(i)==0
            i=i+1;
        end
        Mmplan =c(i);
    end
end
As we stated above, searching across a space of potential sampling plans can be accomplished by pairwise comparisons. We could, therefore, in theory, write an optimization algorithm with $m m$ as the comparative objective. However, there is some experimental evidence (Morris and Mitchell, 1995) that the resulting landscape will be quite deceptive from an optimization point of view and therefore difficult to search reliably. The reason is that the comparison process will stop as soon as we find a nonzero element in the comparison array $c$ and therefore the remaining values in $d_1, d_2, \ldots, d_m$ and $J_1, J_2, \ldots, J_m$ will be lost. These, however, could provide the optimization process with potentially useful ‘slope’ information on the global landscape.

Morris and Mitchell (1995) define the following scalar-valued criterion function used to rank competing sampling plans. This, while based on the logic of Definition 1.2, includes the vectors $d_1, d_2, \ldots, d_m$ and $J_1, J_2, \ldots, J_m$ in their entirety:

$$
\Phi_q(X) = \left( \sum_{j=1}^{m} J_j d_j^{-q} \right)^{1/q}.
$$

(1.6)

The smaller the value of $\Phi_q$, the better the space-filling properties of $X$ will be. Here is Equation (1.6) in MATLAB-speak:

```matlab
function Phiq=mmphi(X, q, p)
% Calculates the sampling plan quality criterion of Morris and Mitchell.
% Inputs:
%    X - sampling plan
%    q - exponent used in the calculation of the metric
%    p - the distance metric to be used (p=1 rectangular - (default, p=2 Euclidean)
% Output:
%    Phiq - sampling plan 'space-fillingness' metric
% Assume defaults if arguments list incomplete
if ~exist('p','var')
    p=1;
end
if ~exist('q','var')
    q=2;
end
% Calculate the distances between all pairs of points (using the p-norm) and build multiplicity array J
% [J,d]=jd(X,p);
% The sampling plan quality criterion
Phiq=sum(J.*(d.^(1-q))^(1/q));
```
This equation distills the cumbersome definition of the maximin criterion into a rather neat and compact form, but it raises the question of how to choose the value of \( q \). Large \( q \)'s will ensure that each term in the sum dominates all subsequent terms. Thus, because the distances \( d_j \) are arranged in ascending order, \( \Phi_q \) will rank sampling plans in a way that matches the original definition of the criterion quite closely and therefore we are back to the original problem. Lower values of \( q \) yield a \( \Phi_q \) landscape that, while it may not match the definition exactly, is more amenable to optimization.

To illustrate the relationship between Equation (1.6) and the maximin criterion of Definition 1.2, let us consider sets of 50 random Latin hypercubes of different sizes and dimensionalities. Let us then compute the ranking of each plan within its set according to Definition 1.2, as well as according to \( \Phi_q \) (using \( p = 1 \) in each case) for a range of values of \( q \).

Figure 1.5 depicts the results of this small investigation. It is unwise to draw far-reaching conclusions from only a few arbitrarily chosen experiments and we neither attempt this here, nor is it really necessary for practical purposes. Nonetheless, the correlation plots suggest that the larger the sampling plan, the smaller the \( q \) required to produce a ranking based on \( \Phi_q \) that matches that of Definition 1.2 almost exactly. Taking the case of the set of 50 100-point 15-variable hypercubes, the bottom right-hand tile of Figure 1.5 indicates that the \( \Phi_{250} \)-based ranking only differs from that of the definition in three places and even these plans are only mis-ranked by one place. At the other end of the scale, it can be seen that for \( q = 1 \), there is virtually no correlation, except for the smallest sampling plans considered.

Should the reader wish to conduct their own investigation for different families of sampling plans, here are the tools required to do it. The rankings according to \( mm \) and \( mmphi \) using a simple bubble sort algorithm are implemented in \texttt{mmsort.m} and \texttt{phisort.m} respectively:

```matlab
function Index=mmsort(X3D,p)
% Ranks sampling plans according to the Morris–Mitchell criterion definition. Note: similar to phisort, which uses the numerical quality criterion \( \Phi_q \) as a basis for the ranking.
%
% Inputs:
% X3D – three–dimensional array containing the sampling plans to be ranked.
% p – the distance metric to be used (p=1 rectangular – default, p=2 Euclidean)
%
% Output:
% Index – index array containing the ranking

if ~exist('p','var')
    p=1;
end

% Pre–allocate memory
Index=(1:size(X3D,3));
```

(continued)
% Bubble-sort
swap_flag=1;

while swap_flag==1
swap_flag=0;
i=1;
while i<= length(Index)-1
if mm(X3D(:,:,Index(i)),X3D(:,:,Index(i+1)),p)==2
buffer=Index(i);
Index(i)=Index(i+1);
Index(i+1)=buffer;
swap_flag=1;
end
i=i+1;
end
end

phisort.m only differs in having \( q \) as the third argument, as well as the comparison line being: if \( \text{mmphi}(X3D(:,:,\text{index}(i)),q,p) > \text{mmphi}(X3D(:,:,\text{index}(i+1)),q,p) \).

So how does one find the best Latin hypercube for a given application? Morris and Mitchell (1995) recommend minimizing \( \Phi_q \) for \( q = 1, 2, 5, 10, 20, 50 \) and 100 (Figure 1.5 confirms these to be reasonable values) and then choosing the best of the resulting plans according to the actual maximin definition. This is one more possible use for mmsort.m; one can create a matrix \( X3D \) with each two-dimensional slice being the best sampling plan found according the various \( \Phi_q \)’s and mmsort\( (X3D,1) \) will then rank them according to Definition 1.2 using the rectangular distance metric. The only remaining question then is, how to find these optimized \( \Phi_q \) designs? We discuss this next.

Optimizing \( \Phi_q \)

Having established a criterion that we can use to assess the quality of a Latin hypercube sampling plan, we now need a systematic means of optimizing this metric across the space of Latin hypercubes. This is not a trivial task – the reader will recall from the earlier discussion of Latin squares that this space is vast. Indeed, it is so vast that for many practical applications we have little hope of locating the globally optimal solution and we therefore must aim to find the best possible sampling plan within a given computing time budget.

This budget depends on the computational cost of obtaining an objective function value. The optimum division of the total computational effort between generating the sampling plan and actually computing objective function values at the sites therein is an open research question, though typically one would rarely allocate more than about 5% of the total wall time to the former task.

Parallels can be drawn with choosing how much time to spend devising a timetable for revision before an exam. A better timetable will make the revision more effective, but one doesn’t want to take too much of the revision time itself!
Figure 1.5. Scatter plots of maximin rankings versus rankings according to \( \Phi_q \), different values of \( q \) for sets of 50 random Latin hypercubes of different sizes and dimensionalities (the rectangular distance metric was used).
One of the challenges of devising a sampling plan optimizer is to make sure that the search process always stays within the space of Latin hypercubes. We have seen that the defining feature of a Latin hypercube $X$ is that each column is a permutation of the list of the possible levels of the corresponding variable. The smallest alteration we can therefore make to a Latin hypercube without spoiling its key multidimensional stratification property is to swap two of the elements within any of the columns of $X$. Here is a MATLAB implementation of ‘mutating’ a Latin hypercube in this way, generalized to random changes applied to multiple sites:

```matlab
function X=perturb(X, PertNum)

% Interchanges pairs of randomly chosen elements within randomly
% chosen columns of a sampling plan a number of times. If the plan is
% a Latin hypercube, the result of this operation will also be a
% Latin hypercube.
%
% Inputs:
% X – sampling plan
% PertNum – the number of changes (perturbations) to be made
% to X.
%
% Output:
% X – perturbed sampling plan

if ~exist(‘PertNum’,’var’)
    PertNum=1;
end

[n,k]=size(X);

for pert_count=1:PertNum
    col=floor(rand*k)+1;
    % Choosing two distinct random points
    el1=1; el2=1;
    while el1==el2
        el1=floor(rand*n)+1;
        el2=floor(rand*n)+1;
    end
    % Swap the two chosen elements
    buffer=X(el1,col);
    X(el1,col)=X(el2,col);
    X(el2,col)=buffer;

end
```

We use the term ‘mutation’ because this is a problem that lends itself to nature-inspired computation. Morris and Mitchell (1995) use a simulated annealing (SA) algorithm, the detailed pseudocode of which can be found in their paper. As an alternative here we offer a method based on evolutionary operation (EVOP).
Evolutionary operation

As introduced by Box (1957), evolutionary operation was designed to optimize chemical processes. The current parameters of the reaction would be recorded in a box at the centre of a board, with a series of ‘offspring’ boxes along the edges containing values of the parameters slightly altered with respect to the central, ‘parent’ values. Once the reaction was completed for all of these sets of variable values and the corresponding yields recorded, the contents of the central box would be replaced with that of the setup with the highest yield and this would then become the parent of a new set of peripheral boxes.

This is generally viewed as a local search procedure, though this depends on the mutation step sizes, that is on the differences between the parent box and its offspring. The longer these steps, the more global is the scope of the search.

For the purposes of the Latin hypercube search, we apply a variable scope strategy. We start with a long step length (that is a relatively large number of swaps within the columns) and, as the search progresses, we gradually home in on the current best basin of attraction by reducing the step length to a single change.

In each generation we mutate (randomly, using perturb.m) the parent a pertnum number of times. We then pick the sampling plan that yields the smallest $\Phi_p$ value (as per the Morris–Mitchell criterion, calculated using mmphi.m) among all offspring and the parent; in evolutionary computation parlance this selection philosophy is referred to as elitism. Should the reader wish to opt for a nonelitist approach (for example to encourage a more global search), the EVOP code can be modified fairly easily to exclude the parent from the selection step.

The EVOP based search for space-filling Latin hypercubes is thus a truly evolutionary process: the optimized sampling plan results from the nonrandom survival of random variations.

Putting it all together

We now have all the pieces of the optimum Latin hypercube sampling process puzzle: the random hypercube generator as a starting point for the optimization process, the ‘space-fillingness’ metric that we need to optimize, the optimization engine that performs this task and the comparison function that selects the best of the optima found for the various $q$’s. We just need to put this all into a sequence. Here is the MATLAB embodiment of the completed puzzle (with some of the frills omitted):

```matlab
function X=bestlh(n,k, Population, Iterations)
% Generates an optimized Latin hypercube by optimizing the Morris–
% Mitchell criterion for a range of exponents and plots the first two
% dimensions of the current hypercube throughout the optimization
% process. %
% Inputs:
%   n – number of points required
%   k – number of design variables
%   Population – number of individuals in the evolutionary
%   operation optimizer
%   Iterations – number of generations the evolutionary
```

(continued)
% operation optimizer is run for
% Note: high values for the two inputs above will ensure high
% quality hypercubes, but the search will take longer.
%
% Output:
% $X$ - optimized Latin hypercube

if $k<2$
    error('Latin_hypercubes_are_not_defined_for_k<2');
end

% List of qs to optimize $\Phi_q$ for
q=[1 2 5 10 20 50 100];

% Set the distance norm to rectangular for a faster search. This can
% be changed to $p=2$ if the Euclidean norm is required.
p=1;

% We start with a random Latin hypercube
XStart=rlh(n,k);

% For each q optimize $\Phi_q$
for $i=1$:length(q)
    fprintf('Now_optimizing_for_q=%d\n', q(i));
    X3D(1:n,1:k,i)=mmlhs(XStart, Population, Iterations, q(i));
end

% Sort according to the Morris–Mitchell criterion
Index=mmsort(X3D,p);
fprintf('Best_lh_found_using_q=%d\n', q[Index(1)]);

% And the Latin hypercube with the best space–filling properties is...
X=X3D(:,:,Index(1));

% Plot the projections of the points onto the first two dimensions
plot (X(:,1),X(:,2),'ro');drawnow;

title(strcat('Morris–Mitchell_optimum_plan_found_using_q=','...
num2str(q(Index(1)))));
xlabel('x_1'); ylabel('x_2');

It is worth noting that we need not necessarily have sorted all the candidate plans in
ascending order – after all, it is the best one we are really interested in. Nonetheless, the
added computational complexity is minimal (the vector is only ever going to contain as
many elements as there are candidate $q$ values and only an index array is sorted, not the
actual repository of plans) and this gives the reader the opportunity to compare, if desired,
the plans different choices of $q$ lead to.
1.4.4 Space-filling Subsets

We have, so far, looked at the problem of minimizing $\Phi_q$ over the space of all Latin hypercubes of a certain size $n$ and dimensionality $k$. Another question might be: how do we find the best space-filling plan across a more restricted space, say, that of $n_s$ element subsets $X_s$ of an $n$ element plan $X$? This is not merely an academic exercise, as this problem will arise later on when we discuss improving the quality of a predictor by running a space-filling subset of its training data through higher fidelity analysis.

Since selecting the $n_s$ element subset that minimizes $\Phi_q$ is an NP-complete problem and an exhaustive search would have to examine $\binom{n}{n_s} = \frac{n!}{n_s!(n-n)!}$ subsets (infeasible for all but very moderate cardinalities), an alternative strategy could be the following greedy algorithm aimed at locating at least a local optimum.

**Strategy one: greedy local search**

We start from a sample point $x^{(i)}$, $i \leq n$, as the first member of $X_s$, we then loop through the remaining candidates $x^{(j)}$, $j = 1, \ldots, i-1$ and $j = i+1, \ldots, n$, and add the point that minimizes the Morris–Mitchell criterion. This loop is then repeated (always leaving out the points we have already included), choosing each time the point that, when added to $X_s$, minimizes the optimality criterion over the points we have so far. This is akin to a local optimization of the criterion — better results can be achieved by repeating the process from all $n_c$ possible starting points, keeping the best $X_s$ overall (multistart local search).

While not exhaustive, even this approach can prove computationally prohibitive beyond plans greater than a few dozen elements. Here is an even cheaper alternative.

**Strategy two: exchange algorithm**

We start from a randomly selected subset $X_s$ and calculate the $\Phi_q$ criterion. We then exchange the first point $x_s^{(i)}$ with each of the remaining points in $X \setminus X_s$ and retain the exchange that gives the best $\Phi_q$. Here is the MATLAB implementation of this technique:

```matlab
function [Xs,Xr]=subset(X,ns)
%
% Given a sampling plan, returns a subset of a given size with
% optimized space–filling properties (as per the Morris–Mitchell
% criterion).
%
% Inputs:
% X – full sampling plan
% ns – size of the desired subset
%
% Outputs:
% Xs – subset with optimized space–filling properties
% Xr – remainder X\Xs
%
n=size(X,1);
%
% Norm and quality metric exponent – different values can be used if
% required
p=1; q=5;
(continued)
```
\begin{verbatim}
r=randperm(n);
Xs=X(r(1:ns),:);
Xr=X(r(ns+1:end),:);

for j=1:ns
    orig_crit=mmphi(Xs,q,p);
    orig_point=Xs(j,:);

    bestsub=1;
    bestsubcrit=Inf;

    for i=1:n-ns
        % We look for the best point to substitute the current one with
        % bestsub=1;
        % bestsubcrit=Inf;
        Xs(j,:)=Xr(i,:);
        crit=mmphi(Xs,q,p);
        if crit< bestsubcrit
            bestsubcrit=crit;
            bestsub=i;
        end
    end

    if bestsubcrit<orig_crit
        Xs(j,:)=Xr(bestsub,:);
    else
        Xs(j,:)=orig_point;
    end
end
\end{verbatim}

1.5 A Note on Harmonic Responses

We conclude our discussion of sampling plans with uniform projections with some ‘small print’ regarding cases when exact uniformity is actually undesirable.

If the function being sampled is expected to have a harmonic component with a wavelength comparable to an integer multiple of 1/n (the width of each bin in a Latin hypercube), the completely uniform projection properties of a Latin hypercube (or full factorial design) might lead to misleading samples (they always sample the signal in the same phase, therefore it will seem like the points could have come from a constant function). This potential (though somewhat unlikely) pitfall can be avoided by adding a small random perturbation to each point:

\begin{verbatim}
% Add a random perturbation
% if harmonic component suspected
if perturb ==
    X=X+(rand(n,k)-0.5)*(1/n)*0.25;
end
\end{verbatim}
1.6 Some Pointers for Further Reading

The importance of sampling plan design in a wide range of disciplines is reflected in the richness of the relevant literature (though not all of this body of work uses the same terminology as the present text). Our practical introduction to the subject is limited to a small subset of the plethora of techniques that have emerged and it is inevitably subject, to some extent, to the authors’ personal bias. More significantly, though, we have highlighted the approaches that, in our view, make the weakest assumptions regarding the type and the size of the problem.

We have also limited the descriptions of the theoretical backgrounds of the techniques covered to what we deemed to be necessary for their correct use. Here are a few additional resources the reader may wish to consult.

The history of Latin hypercubes began in 1979 with a *Technometrics* paper by McKay et al. (1979). A series of refinements have followed, including the application of intersite distance criteria to Latin hypercube plans. Of these we have focused here on the maximin criterion – the text by Santner et al. (2003) is a good source of information on others.

Another class of sampling plans that have generated some interest in recent decades are *orthogonal arrays*. An $n \times k$ matrix $X$ with entries from a set of $s \geq 2$ symbols is called an orthogonal array of strength $r$, size $n$ with $k$ constraints and $s$ levels if each $n \times r$ submatrix of $X$ contains all possible $1 \times r$ row vectors with the same frequency $\lambda$ (note the rather awkward restriction that the number of entries must be $n = \lambda s^r$). In a 1993 paper, Tang (1993) introduces orthogonal array based Latin hypercubes. These essentially extend the univariate stratification properties of Latin hypercubes to $r$-variate margins, again for a limited range of plan sizes.

The sampling plans discussed in this chapter are generally based on the assumption that the size $n$ of the plan is predetermined. This is not always the case, as, although we may know what the total computing time is we can budget for, it is not always obvious how many candidate designs we will be able to evaluate in that time (some designs may be harder to analyse than others).

Should we run out of time after evaluating, say, 80% of the points of our carefully constructed Morris–Mitchell optimal Latin hypercube, there will be no guarantees regarding the space-filling properties of that $0.8n$-strong subset.⁶ In such cases it often makes sense to start with a plan small enough to fit safely into the budget and select subsequent points one by one (until the time runs out) based on models fitted to the points we have thus far, that is deciding where we are sampling next based on which areas appear promising. We shall discuss such procedures in detail later on; however, if they are infeasible for some reason (for example the budget is very large and thus the repeated model fitting process may be impractical), there is an alternative: Sobol sequences (Sobol, 1994). These are sampling plans with reasonably good space-filling properties (at least for large $n$) and have the property that, for any $n$ and $k > 1$, the sequence for $n - 1$ and $k$ is a subset of the sequence for $n$ and $k$. Thus, from a ‘space-fillingness’ point of view, it does not matter when the time runs out.

---

⁶ A question arising from here is, what is the ordering of a sampling plan $X = \{x_1, x_2, \ldots, x_n\}$ that, given the ‘space-fillingness’ metric $\Phi$, optimizes $\Phi(\{x_1, x_2\})$, $\Phi(\{x_1, x_2, x_3\})$, \ldots and $\Phi(\{x_1, x_2, x_3, \ldots, x_n\})$ simultaneously?
References


