GENERATING SYNTHETIC UNIT-RECORD DATA FROM PUBLISHED MARGINAL TABLES

Alan Lee

Department of Statistics, University of Auckland
Private Bag 92019, Auckland, New Zealand
email: `lee@stat.auckland.ac.nz`

SUMMARY. We survey methods for generating synthetic data sets without making use of unit-record data. The methods we describe are based on creating data sets which match publically available marginal tables. We describe a set of R functions which implement the methods under study, and apply the methods to data from the 2001 Census of Population and Dwellings.

KEY WORDS: Contingency tables, marginal tables, log-linear models, maximum likelihood, iterated proportional fitting, integer programming.

Printed: May 4, 2007

Contents

1 INTRODUCTION

Recently there has been an increased interest in issues concerning confidentiality of data. Statistical agencies are often required by law or policy to protect the confidentiality of the information they collect from individuals, businesses or other organisations. Much survey and census data is categorical in nature and often statistical agencies report the results in the form of marginal cross-classification tables of counts by aggregating over different categories. In such tables of counts, the occurrence of small values is usually taken to present the possibility of a disclosure risk, since data for individuals who are unique in the population may be used in matching against other databases by an intruder or data snooper. In order to reduce the likelihood of this, statistical agencies often employ disclosure limitation methods minimize information loss while keeping small the disclosure risk from different data snoopers. Considerable effort has gone into developing disclosure limitation methods for tabular data that effectively lower disclosure risk. These techniques include cell suppression, global recoding, rounding, and various forms of perturbation.

This project starts from a slightly different perspective. As discussed above, data collected by statistical agencies is usually published in the form of marginal tables and the original unit-record data are not publically available due to confidentiality requirements. However, in some circumstances marginal tables are not sufficient, but a synthetic data set matching the original in key respects may suffice. For example, unit-record data may be required for educational purposes and as test data for software development projects. Completely fictitious data could be used, but it is desirable that the unit-record data be plausible in the sense that the marginal tables derived from the synthetic data match, either exactly or approximately, the published marginal tables.

In other words, in many cases what we require is a synthetic data set which has the same marginal totals for specified margins as the original one. This data set can be conveniently represented as a complete cross-classification of all the variables of interest. Put this way, the problem reduces to finding a complete table whose margins match specified margins from the original, confidential complete table.

In this report, we describe two complementary approaches to this problem. First, we address the problem of finding a joint table whose margins match specified published margins **exactly**. This approach uses integer programming techniques, where the marginal tables are used to form linear constraints. This approach works well for moderate size tables, but cannot handle very large problems due to the inability of current integer programming software to deal with the associated time and memory demands.

The second approach is to find a table whose margins match specified published tables **approximately**. This approach is feasible for much larger tables, and relies on the theory of hierarchical log-linear models, iterative proportional fitting and simulation.

In this report, we do not address the interesting question of when a synthetic data set can be used as if the data were genuine, and when such an approach will permit valid inferences. To do this, we must have a genuine unit-record data set available. In this project, we do not assume we have such a data set; we work solely with sets of marginal tables that have been publicly released. Likewise, we do not address the interesting question of which marginal tables can be released without compromising the confidentiality of the unit-record data. For a discussion of this, see Dobra and Fienberg (2000).

1.1 The integer programming (IP) approach

The following simple example illustrates the IP approach.

Example 1. Consider the following toy example from Agresti (2002, p 332) on alcohol, cigarette and marijuana use among US high school students. 2276 high school students were asked if they had ever used alcohol, cigarettes or marijuana. This resulted in a data set with 3 variables:

A: Alcohol ever used (Y/N)

C: Cigarettes ever used (Y/N)

M: Marijuana ever used (Y/N)

The data generate the 3-dimensional contingency table shown in Table 1, which we can take as a representation of the unit-record data.

Table 1: Alcohol, cigarette and marijuana use by US high school seniors.

		Cigarette Use	
Marijuana	Alcohol	Yes	No
Yes	Yes	911	44
	No	3	2
No	Yes	538	456
	No	43	279

Table 2: Marginal tables for the ACM data.

		M	
		Yes	No
A	Yes	955	994
	No	5	322

		C	
		Yes	No
A	Yes	1449	500
	No	46	281

		M	
		Yes	No
C	Yes	914	581
	No	46	735

Suppose we are given the three marginal tables shown in Table 2. Can we recover a table that matches these margins? We can represent a generic $2 \times 2 \times 2$ table by a

Table 3: A generic $2 \times 2 \times 2$ table.

		Cigarette Use	
Marijuana	Alcohol	Yes	No
Yes	Yes	x_1	x_3
	No	x_2	x_4
No	Yes	x_5	x_7
	No	x_6	x_8

vector $x = (x_1, \ldots, x_8)$ of non-negative integers (cell counts) as shown in Table 3: The requirement that this complete table have identical margins to that of the original table gives the constraints

$$\begin{array}{lll} x_1 + x_2 = 914 & x_1 + x_5 = 1449 & x_1 + x_3 = 955 \\ x_3 + x_4 = 46 & x_2 + x_6 = 46 & x_2 + x_4 = 5 \\ x_5 + x_6 = 581 & x_3 + x_7 = 500 & x_5 + x_7 = 994 \\ x_7 + x_8 = 735 & x_4 + x_8 = 281 & x_6 + x_8 = 322 \end{array}$$

where the solutions are non-negative integers. A quick and dirty solution is to embed this problem in an integer programming problem, e.g. maximize $x_1 + \cdots + x_8$ subject to the constraints. All we want is a feasible solution. In actual fact the objective function has only one value as it is fixed by the constraints.

For example, in SAS, the following code does the job:

```
data test;
input id $ x1-x8 _type_ $ _rhs_;
datalines;
obj  1 1 1 1 1 1 1 1 max .
con1 1 1 0 0 0 0 0 0 eq 914
con2 0 0 1 1 0 0 0 0 eq 46
con3 0 0 0 0 1 1 0 0 eq 581
con4 0 0 0 0 0 0 1 1 eq 735
con5 1 0 0 0 1 0 0 0 eq 1449
con6 0 1 0 0 0 1 0 0 eq 46
con7 0 0 1 0 0 0 1 0 eq 500
con8 0 0 0 1 0 0 0 1 eq 281
con9 1 0 1 0 0 0 0 0 eq 955
con10 0 1 0 1 0 0 0 0 eq 5
con11 0 0 0 0 1 0 1 0 eq 994
con12 0 0 0 0 0 1 0 1 eq 322
upper 1500 1500 1500 1500 1500 1500 1500 1500 upperbd .
type 1 1 1 1 1 1 1 1 integer .
```

```
run;
proc lp;
run;
```

This results in the solution shown in Table 4.

Table 4: Solutions to the IP.

Original	SAS
911	909
3	5
44	46
2	0
538	540
43	41
456	454
279	281

1.1.1 Constraints

To every set of marginal tables, there corresponds a hierarchical log-linear model[1] of the form $\log(\mu) = X\alpha$ for the complete table, where μ is the vector of cell means. The marginal tables correspond to the maximal terms in the hierarchical model, so that for example, the three 2-dimensional margins in our example correspond to the model [AC][AM][CM].

Using the "treatment" parametrization in R, we prove in Section 2.2 that the constraint matrix A is just the transpose of the "model matrix" X corresponding to the model

```
 counts ~ A*C + A*M + C*M.
```

Using this idea, we can compute a solution using the R package `lpSolve` with the code (Note the R prompts ">")

```
> ACM = data.frame(counts=c(911, 3, 44, 2, 538, 43, 456, 279),
 expand.grid(A=c("Yes","No"), C=c("Yes","No"), M=c("Yes","No")) )
> A = t(model.matrix(counts ~ A*C + C*M + A*M, data=ACM))
> library(lpSolve)
> b = A%*%ACM$counts
> lp.soln = lp (direction = "min", rep(1,8), A,
     rep("=",7), b, transpose.constraints = TRUE, int.vec=1:8)
> cbind(ACM$counts, lp.soln$solution)
      [,1] [,2]
[1,]  911  914
```

[1]Log-linear models are discussed in more detail in Section 2.2.

```
[2,]    3   0
[3,]   44  41
[4,]    2   5
[5,]  538 535
[6,]   43  46
[7,]  456 459
[8,]  279 276
```

Note that A has 7 rows, corresponding to 7 constraints, so that the 5 redundant constraints have been eliminated. In this example, there are 6 solutions which can be found as follows: Note that $x_2 + x_4 = 5$, so that the only possible solutions for x_2 are are 0,1,2,3,4,5. There are 7 constraints and 8 variables, so that fixing any one variable determines the others. All these solutions are feasible, they are

	Soln 1	Soln 2	Soln 3	Soln 4	Soln 5	Soln 6
x_1	914	913	912	911	910	909
x_2	0	1	2	3	4	5
x_3	41	42	43	44	45	46
x_4	5	4	3	2	1	0
x_5	535	536	537	538	539	540
x_6	46	45	44	43	42	41
x_7	459	458	457	456	455	454
x_8	276	277	278	279	280	281

1.1.2 Scaling up

This example suggests representing the entries of a general table by variables $x_1 \ldots, x_n$ (there will be many of these in a real example, but only 8 above) and finding a solution to a linear equation of the form

$$Ax = b$$

where the elements of the vector b are the entries in the marginal tables. The elements of the solution x must be non-negative integers. This is straightforward in such a simple example, but several questions arise for more realistic problems, for example

- How can we automate construction of constraints, so that redundant equations are eliminated?
- Integer programming is computationally difficult, what are the practical limits on problem size?
- What algorithms should be used?
- Which solution to choose?
- Census tables are subject to base-3 rounding, so that the members of a set of marginal tables will not be consistent with one another. How do we adjust the tables to render them consistent?

1.2 The iterative proportional fitting approach

As noted previously, to each specified set of margins, there corresponds a unique hierarchical log-linear model. To fit the model using the iterative proportional fitting (IPF) algorithm (which we describe in more detail in Section 2.2.3), we need only the marginal tables, as these are sufficient statistics for the problem. The result of the fitting process is a set of estimated cell probabilities or cell means for the complete table. Using these, we can generate random samples of arbitrary size from the corresponding multinomial or Poisson distribution, or simply round the cell means. The following example illustrates the idea.

Example 2. (The Agresti data again.) We can represent the data in R as the data frame ACM :

```
> ACM
  counts   A   C   M
1    911 Yes Yes Yes
2      3  No Yes Yes
3     44 Yes  No Yes
4      2  No  No Yes
5    538 Yes Yes  No
6     43  No Yes  No
7    456 Yes  No  No
8    279  No  No  No
```

We can obtain the fitted cell probabilities using the IPF algorithm. Alternatively, since in this case we actually know the complete data, we will cheat and use the standard R `glm` software to fit the model corresponding to the three 2-dimensional marginal tables. The following R code fits the model, calculates the fitted probabilities for each cell, and generates 10 random samples of $N = 2276$ individuals from the probability distribution corresponding to the fitted table.

```
> ACM.glm = glm(counts~A*C + A*M + C*M, data=ACM, family=poisson)
> fitted.means = predict(ACM.glm, type="response")
> N=sum(ACM$counts)
> fitted.probs = fitted.means/N
> rmultinom(10, N, fitted.probs)
```

The results of 10 samplings, together with the original data, are shown in Table 5.

1.3 Organisation of the report

The simple examples discussed above illustrate the methods we will use to construct synthetic data sets that match a given set of margins. In the rest of the report, we discuss the software and algorithms we require to solve problems of a more realistic size. In Section 2, we outline the relevant theory, beginning with a discussion of log-linear models. We develop a general notation suitable for dealing with tables of arbitrary size, and discuss the connections between log-linear models, marginal tables, constraint matrices and the IPF

Table 5: Ten samples from the ACM distribution

		Sample									
	Actual	1	2	3	4	5	6	7	8	9	10
x_1	911	912	914	935	946	909	932	914	899	937	913
x_2	3	5	2	2	5	2	4	4	2	4	3
x_3	44	65	55	37	49	54	33	32	40	39	44
x_4	2	1	1	1	2	0	2	1	2	2	0
x_5	538	561	536	529	484	567	504	546	518	558	573
x_6	43	42	41	41	40	33	39	39	43	37	47
x_7	456	447	454	447	465	443	450	456	468	435	432
x_8	279	243	273	284	285	268	312	284	304	264	264

algorithm. We then go on to briefly outline the branch-and-bound algorithm for linear programming, discuss how solutions can be modified, and how tables can be adjusted to have compatible margins.

In Section 3, we describe the algorithms and software we use to generate our synthetic data. We use the R statistical environment (R Development Core Team, 2005) for most of our work, although other packages are used for the IP solutions. We work in a Linux environment, although much of our software will also run under Windows. The programs were developed under Red Hat Linux 4.0 Advanced Server, on a Dell Optiplex 745 with 4Gb of memory.

We begin by describing the data structures we use to represent tables, and then discuss three implementations of the IP approach, using the freeware program `lp_solve`, the SAS system and finally the mathematical programming package CPLEX. Next, we give a description of an IPF implementation, and then discuss R functions to harmonise tables and modify IP solutions. We close with a description of the R functions used.

In Section 4, we describe a set of marginal tables from the 2001 Census of Population and Dwellings, and show how they can be used to generate a variety of synthetic data sets. Summary and conclusions are in Section 5, and more formal documentation of the software appears in the Appendices.

2 THEORY

In this section we outline the theory that underpins our software, and develop a notation suitable for discussing contingency tables of arbitrary size.

2.1 Contingency tables

Suppose we have a population of N individuals, on each of whom we make K categorical measurements. Typical examples of such measurements or *factors* are age groups, employment status and so on. We denote these factors by $A_1, \ldots, A_K$. The set of possible categories for a factor is called the set of *levels* for that factor. Thus, the factor "Gender" has levels ("Female", "Male"), and the factor "Age group" has levels (0-4, 5-9, . . . , 80-84, 85+), or any other definition that might be appropriate. We assume that there are a finite (typically small) number of levels for each factor. We denote the number of levels of factor A_k by I_k. Thus, there are $I = I_1 \times I_2 \times \cdots I_k$ possible combinations of levels. A typical combination of levels is denoted by $i = (i_1, \ldots, i_K)$.

The usual way to represent the data on these N individuals is by a *contingency table*, an $I_1 \times I_2 \times \cdots I_k$ array of counts, where the array element in position $(i_1, \ldots, i_K)$ or "cell count", is the number out of the N individuals that have $A_1 = i_1, \ldots, A_K = i_K$. We use the notation $y[i_1, \ldots, i_k]$ or more compactly $y[i]$ to denote the cell count.

Note that the table depends on the order of the factors, and the ordering of the levels of each factor. Given a fixed order of the factors, (which in this report will usually be alphabetical), and a fixed ordering of the levels of each factor, we can arrange the cell counts in a one-dimensional array or *vector* by stringing the counts out in reverse lexigraphic order, where the leftmost index i_1 varies most rapidly, followed by i_2 and so on. Thus if $K = 2$, $I_1 = 2$, and $I_2 = 3$, the ordering would be

$$y[1,1],\ y[2,1],\ y[1,2],\ y[2,2],\ y[1,3],\ y[2,3].$$

In some contexts the array representation is more convenient, and in others the vector representation.

2.1.1 Marginal tables

Given a contingency table, we can form various marginal tables by summing over certain indices. For example, suppose we have a 3-dimensional table with three factors, say age group, employment status and gender. We can form the marginal age group $\times$ employment status table by summing over the index corresponding to gender. The marginal table has counts

$$y_M[i_1, i_2] = \sum_{i_3=1,\ldots,I_3} y[i_1, i_2, i_3].$$

In a similar manner, we can form the gender $\times$ employment status table by summing over age group, and the age group $\times$ employment status table by summing over gender. We also have one-dimensional tables: the gender table is formed by summing over age group and employment status, and so on. We can also regard the table grand total as a“null margin” formed by summing over all the factors. It is clear that to every subset S of $\{1, \ldots, K\}$ there corresponds a marginal table formed by summing over all the indices *not* in S. Thus, there are 2^K possible marginal tables, including the original table and the table total.

2.2 Log-linear models

In contingency table work, a common assumption is to regard every count in the table as the realization of a Poisson random variable with a given mean μ. A *model* for the table is a formula which specifies the mean μ as a function of the factor levels corresponding to each cell. We write μ as $\mu[i_1, \ldots, i_K]$ to emphasise the dependence on the factor levels. Since the Poisson means are necessarily positive this constraint is neatly handled by specifying the log of the mean. Such models, giving the form of $\log \mu[i_1, \ldots, i_K]$, are called log-linear models. Excellent discussions of these models are to be found in Agresti (2002) and Christensen (1997).

We now describe a useful class of log-linear models that correspond in a natural way to classes of marginal tables. We begin by considering the case where the table is two-dimensional, so that we have $K = 2$ and two factors A_1 and A_2. Put $\lambda[i_1, i_2] = \log \mu[i_1, i_2]$, and define four sets of parameters as follows.

The “constant term” : this is $\alpha_0 = \lambda[1, 1]$.

The “A_1 main effects” : There are I_1 of these, defined by $\alpha_1[i_1] = \lambda[i_1, 1] - \lambda[1, 1]$, $i_1 = 1, 2, \ldots, I_1$. Note that necessarily $\alpha_1[i_1] = 0$ by definition.

The “A_2 main effects” : There are I_2 of these, defined by $\alpha_2[i_2] = \lambda[1, i_2] - \lambda[1, 1]$, $i_2 = 1, 2, \ldots, I_2$. Again, $\alpha_2[1] = 0$ by definition.

The “A_1A_2 interactions” : There are $I_1 \times I_2$ of these, defined by $\alpha_{12}[i_1, i_2] = \lambda[i_1, i_2] - \lambda[i_1, 1] - \lambda[1, i_2] - \lambda[1, 1]$, $i_1, i_2 = 1, 2, \ldots, I_2$. Note that $\alpha_{12}[i_1, i_2] = 0$ if either of i_1 or i_2 is 1.

In terms of these quantities, we can write

$$\lambda[i_1, i_2] = \alpha_0 + \alpha_1[i_1] + \alpha_2[i_2] + \alpha_{12}[i_1, i_2].$$

Note that this parametrization puts no restrictions on the means $\mu[i_1, i_2]$: we have simply expressed the $I_1 \times I_2$ means in terms of $I_1 \times I_2$ non-zero new parameters (1 constant term, $(I_1 - 1)$ non-zero A_1 main effects, $(I_2 - 1)$ A_2 main effects, and $(I_1 - 1) \times (I_2 - 1)$ non-zero A_1A_2 interactions. By arranging the constant terms, main effects and interactions into a vector α, we can write

$$\log \mu = X\alpha, \tag{1}$$

where X is the *model matrix*.

By setting the elements of α corresponding to the interactions to zero, we obtain a new, restricted model for the cell means.

2.2.1 The general case

Suppose now we have a K-dimensional table, obtained by classifying the individuals in the population according to K criteria $A_1, \ldots, A_K$. For $l = 1, \ldots, K$ and $j = 2, .., I_l$, define a "dummy variable" $D_j^{(l)}$ by

$$D_j^{(l)}[i_1, \ldots, i_K] = \begin{cases} 1, & \text{if } i_l = j; \\ 0, & \text{otherwise.} \end{cases}$$

For a subset $\{l_1, \ldots, l_r\}$ of $\{1, \ldots, K\}$, let $X_{l_1,\ldots,l_r}$ be the matrix whose $I_1 \times I_2 \times \cdots \times I_K$ rows correspond to the different factor level combinations $(i_1, \ldots, i_K)$, and whose columns are of the form

$$D_{j_1}^{(l_1)} \times D_{j_2}^{(l_2)} \cdots \times D_{j_r}^{(l_r)}, \qquad 2 \le j_1 \le I_{l_1}, \ldots, 2 \le j_r \le I_{l_r},$$

where $\times$ denotes elementwise multiplication. Then put

$$X = \big[1|X_1|\cdots|X_K|X_{12}|\cdots|X_{(K-1),K}|\cdots|X_{1,\ldots,K}\big].$$

Thus, for example when $K = 3$,

$$X = \big[1|X_1|X_2|X_3|X_{12}|X_{13}|X_{23}|X_{123}\big],$$

where 1 is a column of ones. We can show that the matrix X is square, non-singular and has $I_1 \times I_2 \times \cdots \times I_K$ rows and columns, so that there is a vector α such that

$$\log \mu = X\alpha. \tag{2}$$

We call X the *saturated model matrix.*

If we partition the vector α conformably with X, we obtain subvectors $\alpha_{l_1,\ldots,l_r}$ corresponding to the submatrices $X_{l_1,\ldots,l_r}$. We call the elements of $\alpha_{l_1,\ldots,l_r}$ the $A_{l_1}A_{l_1}\cdots A_{l_r}$ interactions. By setting various of the subvectors $\alpha_{l_1,\ldots,l_r}$ equal to zero, we get various constrained sets of cell means. We can think of a particular model $\mathcal{M}$ as being specified by a particular set of non-zero interactions. The model matrix for the model $\mathcal{M}$ is the matrix $X_{\mathcal{M}}$ obtained by deleting the blocks corresponding to the zero interactions from the saturated model matrix X. Thus, the cell means specified by our model $\mathcal{M}$ are given by

$$\log(\mu) = X_{\mathcal{M}}\alpha_{\mathcal{M}}, \tag{3}$$

where $\alpha_{\mathcal{M}}$ is α with the zero interactions deleted.

Example 3. Consider the case $K = 2$, $I_1 = 2$, $I_2 = 3$. Then the dummy variables are

cell	$D_2^{(1)}$	$D_2^{(2)}$	$D_3^{(2)}$
11	0	0	0
21	1	0	0
12	0	1	0
22	1	1	0
13	0	0	1
23	1	0	1

and the matrix X is

$$\left|\begin{array}{c|c|cc|cc} 1 & D_2^{(1)} & D_2^{(2)} & D_3^{(2)} & D_2^{(1)}D_2^{(2)} & D_2^{(1)}D_3^{(2)} \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \end{array}\right|,$$

where the blocks in the matrix correspond to the partition α_0, α_1, α_2, α_{12} of α into constant term, A_1 main effects, A_2 main effects, and A_1A_2 interactions.

We will assume that the log-linear models we consider are *hierarchical*, in the sense that if $\alpha_{l_1,\ldots,l_r}$ is non-zero, so is the α corresponding to any subset of $\{l_1,\ldots,l_r\}$. Thus, if in a hierarchical model α_{12} is non-zero, then α_1 and α_2 must be non-zero as well. A model for which all the $\alpha_{l_1,\ldots,l_r}$'s are non-zero is called the *saturated* model, and the model for which all the $\alpha_{l_1,\ldots,l_r}$'s are zero (except for the constant term) is called the *null* model. The saturated model puts no restrictions on the cell means, but all other models do.

Since we are assuming that our models are hierarchical, we do not need to specify all the non-zero interactions, but only the maximal ones. Thus, if a hierarchical model has an $A_1A_2A_3$ interaction, we don't need to explicitly list the A_1A_2, A_1A_3 and A_2A_3 interactions, they are included implicitly. Thus, we can specify our models compactly by listing only these maximal interactions. In this report, we adopt the notation used by the R statistical system where, for example, the saturated model for 3 factors, with maximal interaction $A_1A_2A_3$, is written $A_1 * A_2 * A_3$, and the model with the $A_1A_2A_3$ interactions zero, with maximal interactions A_1A_2, A_1A_3 and A_2A_3, is written $A_1 * A_2 + A_1 * A_3 + A_2 * A_3$. An alternative notation often used in textbooks, is the "square bracket" notation. Using this, the saturated model is written $[A_1A_2A_3]$ and the model with the zero 3-factor interaction is written $[A_1A_2][A_1A_3][A_2A_3]$.

2.2.2 Estimation

To estimate the parameters of a model from a complete table, we use the method of maximum likelihood. Let $i = (i_1,\ldots,i_K)$ denote a typical cell. The cell count is $y[i]$ and the cell mean is $\mu[i] = \exp(\lambda[i])$, where $\lambda[i] = X_{\mathcal{M}}[i]^T\alpha_{\mathcal{M}}$, and $X_{\mathcal{M}}[i]^T$ is the row of $X_{\mathcal{M}}$ corresponding to cell i. The log-likelihood is

$$\begin{aligned} l &= \sum_i y[i]\log\mu[i] - \mu[i] \\ &= \sum_i y[i]\lambda[i] - \exp(\lambda[i]). \end{aligned}$$

To estimate the parameters, we must maximize l as a function of $\alpha_{\mathcal{M}}$. Consider a typical element a of $\alpha_{\mathcal{M}}$, corresponding to a column $D_{j_1}^{l_1}\cdots D_{j_r}^{l_r}$ of $X_{\mathcal{M}}$. Then, since $\lambda[i] = X_{\mathcal{M}}[i]^T\alpha_{\mathcal{M}}$, we get

$$\frac{\partial\lambda[i]}{\partial a} = D_{j_1}^{l_1}[i]\cdots D_{j_r}^{l_r}[i]$$

so that

$$\begin{aligned}\frac{\partial l}{\partial a} &= \sum_i \frac{\partial l}{\partial \lambda[i]}\frac{\partial \lambda[i]}{\partial a} \\ &= \sum_i (y[i] - \exp(\lambda[i]))D^{l_1}_{j_1}[i]\cdots D^{l_r}_{j_r}[i]\end{aligned}$$

so that at the maximum we get

$$\sum_i y[i]D^{l_1}_{j_1}[i]\cdots D^{l_r}_{j_r}[i] = \sum_i \mu[i]D^{l_1}_{j_1}[i]\cdots D^{l_r}_{j_r}[i]. \tag{4}$$

The expression on the left is just

$$\sum_{i_{l_1}=j_1,\ldots,i_{l_r}=j_r} y[i],$$

which is the $j_1,\ldots,j_r$ entry in the marginal table for $A_{l_1},\ldots,A_{l_r}$. It follows that to estimate the parameters, we don't require the original complete table of counts, but only the marginal tables corresponding to the non-zero interactions in the model. Thus, for example, to fit the model $A_1 * A_2 + A_1 * A_3 + A_2 * A_3$, we require only the A_1A_2, A_1A_3 and A_2A_3 marginal tables. We can then obtain the fitted mean counts for each cell of the complete table by solving the equations (4).

Turning this around, given a set of margins, we can calculate the set of fitted cell means for the model for which the interactions corresponding to the margins are non-zero. Thus, if we have the A_1A_2, A_1A_3 and A_2A_3 margins, we can fit the model $A_1{*}A_2{+}A_1{*}A_3{+}A_2{*}A_3$.

The standard statistical algorithm for solving the equations (4) is known as iterated proportional fitting or IPF, and was invented in the 1940's by Deming and Stephan. We describe the algorithm briefly in the next subsection.

2.2.3 The IPF algorithm

The IPF is a simple but effective algorithm, which allows us to compute the fitted cell means corresponding to a given set of marginal tables, without having to compute the maximum likelihood estimates of α. It was introduced by Deming and Stephan (1940) and has been adapted to many other uses besides the fitting of log-linear models to contingency tables. The algorithm is as follows:

Step 1: Set $\mu[i] = 1$ for each cell i.

Step 2: For each margin in turn, update the $\mu[i]$'s by adjusting them so that the marginal table of fitted means matches the marginal table of cell counts. i.e. for the $l_1,\ldots,l_r$ margin, update the $\mu[i]$'s using the equation

$$\mu^{NEW}[i] = \mu^{OLD}[i] \times \frac{y_{l_1,\ldots,l_r}[i_{l_1},\ldots,i_{l_r}]}{\mu^{OLD}_{l_1,\ldots,l_r}[i_{l_1},\ldots,i_{l_r}]}.$$

Here, $y_{l_1,\ldots,l_r}[i_{l_1},\ldots,i_{l_r}]$ is the $i_{l_1},\ldots,i_{l_r}$ entry in the $l_1,\ldots,l_r$-margin of counts, and $\mu^{OLD}_{l_1,\ldots,l_r}[i_{l_1},\ldots,i_{l_r}]$ is the corresponding entry in the $l_1,\ldots,l_r$-margin of fitted mean counts.

Step 3: Repeat Step 2 until the process converges.

Example 4. For $K = 3$, to fit the model $A_1 * A_2 + A_1 * A_3 + A_2 * A_3$, step 2 takes the form:

Step 2a: Adjust the means to match the $A_1 A_2$ margin by

$$\mu^{NEW}[i_1, i_2, i_3] = \mu^{OLD}[i_1, i_2, i_3] \times \frac{y_{1,2}[i_1, i_2]}{\mu^{OLD}_{1,2}[i_1, i_2]},$$

where $y_{1,2}[i_1, i_2] = \sum_{i_3} y[i_1, i_2, i_3]$.

Step 2b: Adjust the means to match the $A_1 A_3$ margin by

$$\mu^{NEW}[i_1, i_2, i_3] = \mu^{OLD}[i_1, i_2, i_3] \times \frac{y_{1,3}[i_1, i_3]}{\mu^{OLD}_{1,3}[i_1, i_3]}.$$

Step 2c: Adjust the means to match the $A_2 A_3$ margin by

$$\mu^{NEW}[i_1, i_2, i_3] = \mu^{OLD}[i_1, i_2, i_3] \times \frac{y_{2,3}[i_2, i_3]}{\mu^{OLD}_{2,3}[i_2, i_3]}.$$

The algorithm converges reasonably quickly. The log-likelihood is increased at each step. An R implementation of the algorithm is described in Section 3.3.

2.2.4 Generating constraints

Given a set of margins, how can we calculate the constraint matrix A that relates the elements of the complete table to the elements of the given marginal tables? We will describe an algorithm that draws on the log-linear model theory discussed in Section 2.2. First, as described above, we identify a log-linear model with the given set of margins, with a set of maximal interactions in the model corresponding to the given margins. Now consider the blocks in the model matrix. The first block is the column of 1's corresponding to the constant term. The next is the submatrix X_{A_1} corresponding to the $(I_1 - 1)$ A_1 main effects, followed by the submatrix X_{A_2} corresponding to the $(I_2 - 1)$ A_2 main effects and so on.

We identify the columns of the model matrix with entries in the marginal tables as follows. First, the constant term is identified with the grand total. Second, the $(I_1 - 1)$ columns of X_{A_2} correspond to the $2, 3, \ldots I_1$ elements of the A_1 marginal table. We make similar identifications for the other main effects. Similarly, we identify the $(I_1 - 1)(I_2 - 1)$ columns of $X_{A_1:A:2}$ with the entries $y_{12}[j_1, j_2]$ of the $A_1 A_2$ marginal table with $2 \leq j_1 \leq I_1$, $2 \leq j_2 \leq I_2$. In general, the column of X with elements

$$D^{(l_1)}_{j_1}[i] \times D^{(l_1)}_{j_1}[i] \cdots \times D^{(l_r)[i]}_{j_r}$$

corresponds to the $j_1, \ldots, j_r$ element of the $A_{l_1}, \ldots, A_{l_r}$ marginal table. The equation relating this marginal element to the full table is

$$\sum_i D^{(l_1)}_{j_1}[i] \cdots D^{(l_r)}_{j_r}[i]\ y[i] = y_{l_1,\ldots,l_r}[j_1, \ldots, j_r],$$

representing a typical row of the matrix equation $Ay = b$. But

$$D_{j_1}^{(l_1)}[i] \cdots D_{j_r}^{(l_r)}[i]$$

is the element of the model matrix in row i and column $(j_1, \ldots, j_r)$ of the $(l_1, \ldots, l_r)$ block of X. It follows that the constraint matrix is the transpose of the model matrix.

This suggests that we can generate the constraint matrix by using available software that allows us to extract the model matrix from a log-linear model fit. The function `model.matrix` in R is an example of such a piece of software. Alternatively, we can generate the constraint matrix directly from the definition of the model matrix by forming the dummy variables and multiplying them together. The following algorithm does the job:

Step 1: Given a set of marginal tables, identify each table with a subset of $\{1, \ldots, K\}$, by identifying the $A_{l_1}, \ldots, A_{l_r}$ margin with the set $L = \{l_1, \ldots, l_r\}$. The grand total is identified with the empty set.

Step 2: Write down a list of these subsets, and all subsets of these subsets.

Step 3: Eliminate duplicates from the list.

Step 4: For each set $\{l_1, \ldots, l_r\}$ in the list, and each set of indices $\{j_1, \ldots j_r\}$ with $2 \leq j_1 \leq I_{l_1}$, $2 \leq j_2 \leq I_{l_2}$, $\cdots$, $2 \leq j_r \leq I_{l_r}$, create a row of the constraint matrix by forming the products $D_{j_1}^{(l_1)}[i] \cdots D_{j_r}^{(l_r)}[i]$. Add this row to the constraint matrix.

An R function to implement this algorithm is described in Section 3.2.3.

2.3 Integer programming

In this section we describe the "branch and bound" algorithm that is used by most commercial IP codes to solve the integer program

$$\text{maximize } c^T x$$

subject to

$$Ax = b,\ x \geq 0, x \text{ integral}.$$

A trivial modification of this can be used to find a non-negative integral solution to the linear equations $Ax = b$. An excellent discussion of this method may be found in Wolsey (1998).

Before describing the algorithm, we introduce the idea of the "LP relaxation". Suppose we want to solve the problem

$$\text{maximize } c^T x$$

subject to

$$Ax = b,\ x \geq 0.$$

This is the same problem as before, except that the requirement that the solution be integral has been dropped. This modified problem is called the LP relaxation of the original problem. It is much easier to solve, as there are efficient algorithms such as the simplex method available. It is tempting to suggest finding an approximate solution to the IP by solving the LP relaxation and rounding the result. However, this is not satisfactory, as the LP sulition may be very different from the IP solution, and in any event will not in general satisfy the constraints. Neverthe less, the idea of LP relaxation is very useful, and is the basis of the branch and bound algorithm which we now describe.

The branch and bound algorithm proceeds by recursively constructing a "solution set tree" where the current set of feasible solutions is successively divided in two. The algorithm proceeds as follows:

Step 1: At the first stage, the tree consists of a root node, which is the set of feasible solutions $\{x : Ax = b, x \geq 0, x \,\text{integral}\}$. Call this the *current node*; at this stage there is only one *active node*, namely the current one. Take the vector c to be any row of A.

Step 2: We solve the *LP relaxation* of the problem represented by the current node, which solves the linear program

$$\text{maximize } c^T x$$

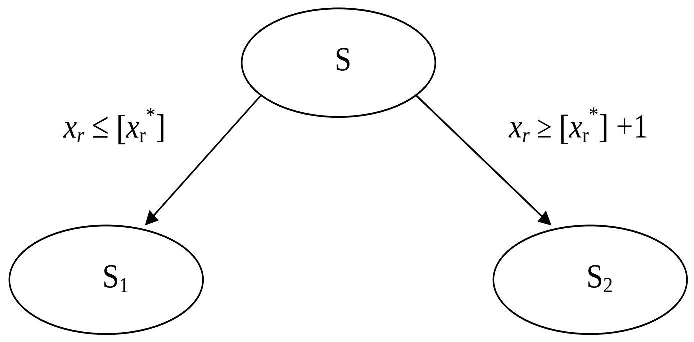
subject to the constraints associated with the current node (but ignoring the requirement that the solution be integral). If the solution to the LP relaxation happens to be integral, we are done: we have found an integral solution. If there is no feasible solution, we remove the current node from the list of active nodes and retrace our steps up the tree to the first active node. This then becomes the current node. If there is no active node, we are done, as there can be no feasible solution to the IP problem in this case.

Step 3: If neither of these outcomes occur, there will be a non-integer variable in the solution say $x_r = {x_r}^*$. We then split the problem into two sub-problems: The first is formed by adding an extra constraint of the form $x_r \leq [{x_r}^*]$, resulting in a new solution set $S_1 = \{x : Ax = b, x \geq 0, x \leq [{x_r}^*], x \,\text{integral}\}$ and the second by adding the constraint $x_r \geq [x_r^*] + 1$, resulting in a solution set $S_1 = \{x : Ax = b, x \geq 0, x \geq [{x_r}^*] + 1, x \,\text{integral}\}$. (We have used the notation $[x]$ to mean the greatest integer $\leq x$.) The node corresponding to S_1 becomes the current node, and that corresponding to S_2 is added to the list of active nodes. The splitting is shown in Figure 1.

Step 4: We keep repeating Steps 2 and 3, until no feasible solution can be found, or an integral solution is found.

If the constraint vector c is a row of A, the branch and bound algorithm for solving the IP implicitly limplements the algorithm described above, as it will terminate as soon as a feasible solution is found.

Figure 1: The branch and bound tree.



Example 5. To find a non-negative integral solution to

$$\begin{bmatrix} 2 & -2 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 7 & -1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \\ 14 \end{bmatrix},$$

we first solve the LP relaxation, maximizing $2x_1 - 2x_2 + x_5$. The solution is (2.2, 0.7, 0.0, 2.3, 0.0). We split on x_1, add the constraint $x_1 \leq 2$ to the problem, and solve the LP relaxation under the constraints $Ax = b,\ x_1 \geq 2,\ x \geq 0$. This gives the solution (2, 0.5, 0, 2.5, 1). Finally, splitting on x_4, we add the constraint $x_4 \leq 2$ to the problem, solve the resulting LP relaxation, and obtain the solution (2, 1, 1, 2, 2). Since this is integral, we are done. The solution set tree for the sequence of splits is shown in Figure 2.

Example 6. Suppose we have the contingency table

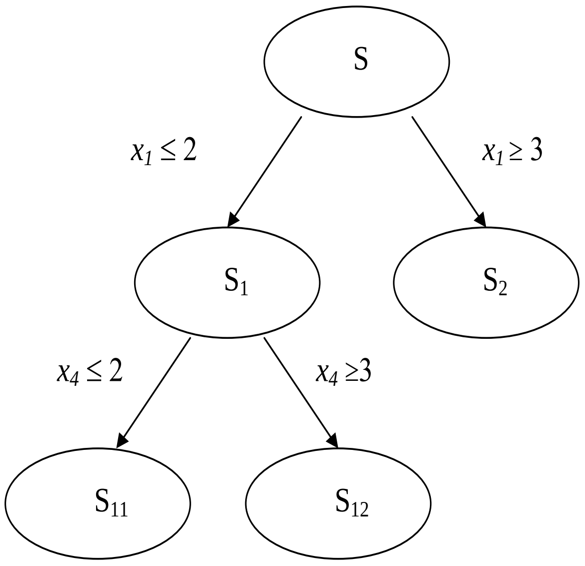
		A_2			
		1	2	3	Total
A_1	1	x_1	x_2	x_3	14
	2	x_4	x_5	x_6	8
	Total	8	4	10	22

which gives rise to the equations

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 22 \\ 8 \\ 4 \\ 10 \end{bmatrix}.$$

To find a table matching the given margins, we must find a non-negative integral solution to these equations. If we solve the LP relaxation, maximizing $x_1 + x_2 + x_3 + x_4 + x_5 + x_6$,

Figure 2: The branch and bound tree for Example 5.



we obtain the solution $(0, 4, 10, 8, 0, 0)$ which is integral. In this case the algorithm finds the solution after one iteration.

For small problems, this is not a coincidence. From LP theory, the solution (after some reordering of the variables) of the LP relaxation is of the form $x = (B^{-1}b, 0)$, where B is a non-singular submatrix of A. Since b has integer elements, x will be integral if the elements of B^{-1} are integers. By Cramer's rule, this will happen if the determinant of B is ± 1. In the present case, this is the case for *all* the non-singular submatrices B of A. In fact, for the types of constraint matrix arising from contingency tables, this tends to happen rather often, at least for small tables.

The form of the LP solution indicates that the solutions to the IP will have a fair number of zero elements, unless the tree gets very large. (In fact the number of zeros is the dimension of x minus the number of constraints, so unless the number of added constraints gets large, the solution will have a lot of zeroes.)

In practice, tables arising from classifying individuals according to a large number of factors will tend to be sparse, with many zero cells. In addition, there are usually a number of cells with very small cell counts. However, the tables constructed using the integer programming method described above tend to have too few cells with small but positive cell counts, so often don't appear very plausible. It seems desirable to modify them to rectify this problem. Next, we examine ways to do this.

2.4 Modifying solutions

In this section, we discuss three methods of modifying the solution obtained by the branch and bound algorithm.

2.4.1 Lattice bases

Let A be the constraint matrix relating the complete table to a set of marginal tables, and let x be a solution to $Ax = b$ with non-negative integer elements. Consider the set of all integral solutions to $Ax = 0$. (In mathematical terms, this set is a *lattice*). One way of modifying a table is to repeatedly select an integral solution, say x^*, of the equations $Ax = 0$, until $x+x^*$ has all non-negative elements. Then $A(x+x^*) = Ax+Ax* = b+0 = b$, so $x + x^*$ is a new non-negative integer solution.

We need a way to generate integer solutions to $Ax = 0$. We now show how to construct a matrix B with integer elements such that $x = Bu$ is a solution for all integral vectors u. The columns of B then form a lattice basis for the solution lattice. To find B, we permute the columns of A (assumed to have r rows and c columns) until the first r columns form a upper triangular matrix A_1 with ones on the diagonal. (An induction argument shows that is always possible.) This amounts to finding a permutation matrix[2] P such that

$$AP = [A_1|A_2].$$

Suppose x^* is a solution, and partition P^Tx^* conformably with A, so that

$$P^Tx^* = \begin{bmatrix} u \\ v \end{bmatrix}.$$

Then, since permutation matrices are orthogonal, we can write

$$\begin{aligned} Ax^* &= APP^Tx^* \\ &= [A_1|A_2]\begin{bmatrix} u \\ v \end{bmatrix} \\ &= A_1u + A_2v. \end{aligned}$$

Thus, $A_1u + A_2v = 0$, so $u = -A_1^{-1}A_2v$ and

$$P^Tx^* = \begin{bmatrix} -A_1^{-1}A_2 \\ I \end{bmatrix} v.$$

This implies that $x^* = Bv$ where

$$B = P\begin{bmatrix} -A_1^{-1}A_2 \\ I \end{bmatrix}.$$

Conversely, for any integer vector v, the vector $x^* = Bv$ is a solution, since

$$\begin{aligned} Ax^* &= APP^Tx \\ &= [A -_{|} A_2]\begin{bmatrix} -A_1^{-1}A_2 \\ I \end{bmatrix} v \\ &= [A_1(-A_1^{-1}A_2 + A_2]v \\ &= 0. \end{aligned}$$

[2]A permutation matrix has exactly one entry equal to 1 in each row and column, with the rest of the entries zero. Post-multiplication by such a matrix permutes columns, and pre-multiplication permutes rows.

Note that, by Cramer's rule, A_1^{-1} must have integer elements since the determinant of A_1 is 1.

Example 7. Consider the contingency table in Example 6, which has constraint matrix

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

The table obtained by the IP algorithm was (0,4,10,8,0,0). To permute A into the correct form, with the first four columns lower triangular, we postmultiply by

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix B is

$$B = \begin{bmatrix} 1 & 1 \\ -1 & 0 \\ 0 & -1 \\ -1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

We picked the vectors v with elements v_1, v_2 randomly chosen from {-10, -9, ..., 9, 10}, generated repeated solutions Bv, and retained those for which all elements of $Bv + x^*$ were positive. The first 5 solutions are shown in Table 6.

Table 6: Five modified tables for Example 7.

	Soln 1	Soln 2	Soln 3	Soln 4	Soln 5
x_1	9	6	2	5	8
x_2	1	4	3	1	1
x_3	1	4	8	5	2
x_4	2	2	7	6	3
x_5	6	6	1	2	5
x_6	3	0	1	3	3

2.4.2 Markov bases

A set of integer solutions $\{x_1^*, \ldots, x_M^*\}$ to the constraint equation $Ax = 0$ is called a Markov basis if for any two solutions x and x' to $Ax = b$, there is a subset $l_1, \ldots, l_s\}$ of $\{1, \ldots, M\}$ and constants $\epsilon_1, \ldots, \epsilon_s$ all either ± 1 such that

1. $x = x' + \sum_{j=1}^{s} \epsilon_j x^*_{l_j}$, and
2. The elements of the vector $x' + \sum_{j=1}^{r} \epsilon_j x^*_{l_j}$ are non-negative for $r = 1, 2, \ldots s$.

The idea is that given a basis, we can move between any two tables with the same fixed margins by adding or subtracting an element of the basis. At each stage we can pick a basis element so that the result remains non-negative, thus performing a "tour" through the collection of tables with the specified margins.

Given a Markov basis, and a table with specified margins, we can make another table with the same specified margins by repeatedly choosing an element of the basis at random, and either adding to or subtracting from the original table (each with probability 0.5) until we obtain a new table with all entries non-negative.

2.4.3 Constructing a Markov basis

To explain how this is done, we must venture briefly into the theory of polynomial ideals. See Cox, Little and O'Shea (1996) for further background. Consider the set of polynomials in n variables $u_1, \ldots, u_n$ where n is the number of columns of A. A special type of polynomial is the mononomial, which is a polynomial having a single term of the form $u_1^{x_1} \cdots u_n^{x_n}$ where $x_1, \ldots, x_n$ are non-negative integers. We shall use the abbreviation u^x to mean the mononomial $u_1^{x_1} \cdots u_n^{x_n}$.

More general polynomials can be obtained by adding one polynomial to another, or multiplying polynomials by constants. Thus, polynomials are linear combinations of mononomials. Polynomials can be added and multiplied, but in general one polynomial cannot be divided by another. In algebraic terms, polynomials form a ring.

Important sets of polynomials are *ideals*. These are sets $\mathcal{I}$ of polynomials that are closed under addition (i.e. if p_1 and p_2 are in $\mathcal{I}$, so is $p_1 + p_2$) and have the property that if p is in $\mathcal{I}$ and q is any polynomial, then the product pq is also in $\mathcal{I}$.

Given a finite set of polynomials $\{p_1, \ldots p_M\}$, we can form a bigger set

$$\mathcal{P} = \left\{ \sum_{i=1}^{M} p_i q_i : q_1, \ldots, q_M \text{ any polynomials} \right\}.$$

We call $\mathcal{P}$ the set *generated by* $\{p_1, \ldots p_M\}$, and write

$$\mathcal{P} = \langle p_1, \ldots p_M \rangle.$$

The polynomials $p_1, \ldots p_M$ are called a *basis* for $\mathcal{P}$, which is an ideal.

In a ring of polynomials, all ideals are finitely generated, in the sense that for any ideal $\mathcal{I}$ there is a finite set of polynomials that generates $\mathcal{I}$. (In other words, every ideal has a finite basis; this is the Hilbert basis Theorem.) A particular finite basis of interest is the Groebner basis; this has the property that a polynomial q is in the ideal if and only if when q is divided by the elements of the Groebner basis, the remainder is zero. Given a set of polynomials $\{p_1, \ldots p_M\}$, there is a well-defined algorithm (Burbacher's algorithm) for calculating the Groebner basis of the ideal generated by $\{p_1, \ldots p_M\}$. Implementations of this algorithm may be found, for example, in the computer algebra packages Maple and Mathematica.

Now consider two rings of polynomials: the first is the ring of polynomials in the variables $u_1, \ldots, u_n$, and the second the ring of polynomials in the variables $t_1, \ldots, t_m$, where m and n are the numbers of rows and columns of A. We now define a function ϕ between these two sets of polynomials. We first define ϕ for mononomials u^x by

$$\phi(u^x) = t^{Ax},$$

so that mononomials in u map onto mononomials in t. The mapping is extended to all polynomials by

$$\phi(p_1 + p_2) = \phi(p_1) + \phi(p_2)$$

and

$$\phi(p_1 p_2) = \phi(p_1)\phi(p_2)$$

for all polynomials p_1, p_2 in the variables $u_1, \ldots, u_n$. Now consider the kernel of this mapping, the set of all polynomials that map onto zero. The kernel is an ideal. The following result, due to Diaconis and Sturmfels (1998) gives a condition for a set of solutions of $Ax = 0$ to be a Markov basis.

Theorem 1 *For a solution x of $Ax = 0$, let x^+ be x with all negative elements set to 0, and x^- be the vector obtained by reversing the signs of the elements of x and then setting the negative elements of the result to 0. Then a set of solutions $\{x_1, \ldots, x_s\}$ of $Ax = 0$ is a Markov basis for the set of solutions if and only if the mononomial differences $\{u^{x_j^+} - u^{x_j^-} : j = 1, 2, \ldots, s\}$ generates the kernel of ϕ.*

Note that if x is a solution, then

$$\begin{aligned} \phi(u^{x^+} - u^{x^-}) &= \phi(u^{x^+} - \phi(u^{x^-}) \\ &= t^{Ax^+} - t^{Ax^-} \\ &= 0 \end{aligned}$$

since

$$Ax^+ - Ax^- = A(x^+ - x^-) = Ax = 0.$$

Thus if x is a solution, the mononomial difference $u^{x^+} - u^{x^-}$ is in the kernel of ϕ. In fact, the reverse is also true: every polynomial in the kernel can be written as a linear combination of mononomial differences $u^{x^+} - u^{x^-}$ corresponding to solutions.

It turns out that the Groebner basis of the kernel of ϕ consists entirely of mononomial differences $u^{x_j^+} - u^{x_j^-}, j = 1, \ldots, s$ and hence is a Markov basis. To find the Groebner basis for the kernel, we use the following theorem, also due to Diaconis and Sturmfels:

Theorem 2 *Consider the ring of polynomials in the variables $t_1, \ldots, t_m, u_1, \ldots, u_n,$. Let G be a Groebner basis for the ideal in this ring generated by the polynomials $u_1 - t^{a_1}, u_2 - t^{a_2}, \ldots, u_n - t^{a_n}$, where $a_1, \ldots, a_n$ are the first, second,…, nth columns of A. Delete from G all polynomials that involve t. Then the resulting set is a Groebner basis for the kernel of ϕ.*

Thus, we have the following procedure for finding the Markov basis.

1. Using a suitable package, compute a Groebner basis for the ideal generated by $x_1 - t^{a_1}, x_2 - t^{a_2}, \ldots, x_n - t^{a_n}$.

2. Delete from the basis all polynomials involving t. What is left is a set of mononomial differences $u^{x_j^+} - u^{x_j^-}, j = 1, \ldots, s$

3. The resulting set of x_j's form the Markov basis for the set of solutions to $Ax = 0$.

Example 8. Consider the contingency table introduced in Example 6. To find a Markov basis for this problem, we first must calculate the Groebner basis for the ideal generated by the polynomials $x_1 - t^{a_1}, x_2 - t^{a_2}, \ldots, x_n - t^{a_n}$, which in this case take the form

$$x_1 - t_1,\ x_2 - t_1t_3,\ x_3 - t_1t_4,\ x_4 - t_1t_1t_2,\ x_5 - t_1t_2t_3,\ x_6 - t_1t_2t_4.$$

This can be done in Maple using the code

```
with(Groebner):
ideal:=[u1 - t1, u2-t1*t3, u3-t1*t4, u4-t1*t2, u5-t1*t2*t3, u6-t1*t2*t4];
gbasis(ideal, plex(t1,t2,t3,t4,u1,u2,u3,u4,u5,u6));
```

which produces the output

```
[-u3 u5 + u6 u2, -u3 u4 + u1 u6, -u2 u4 + u1 u5, -u6 + t4 u4, t4 u1 - u3,
  t3 u6 - t4 u5, -u5 + t3 u4, t3 u3 - t4 u2, t3 u1 - u2, t2 u3 - u6,
 -u5 + t2 u2, t2 u1 - u4, -u1 + t1]
```

Deleting the polynomials in t, we get the polynomials $u_2u_6 - u_3u_5,\ u_1u_6 - u_3u_4,\ u_1u_5 - u_2u_4$. The corresponding solutions are

$$\begin{aligned} x_1^* &= (0, 1, -1, 0, -1, 1), \\ x_2^* &= (1, 0, -1, -1, 0, 1), \\ x_3^* &= (1, -1, 0, -1, 1, 0). \end{aligned}$$

Note that the last two of these coincide with the lattice basis.

Using the solution $x^* = (0, 4, 10, 8, 0, 0)$, we generated repeated solutions $x^* \pm x'$ where x' is one of $x_1^*,\ x_2^*,\ x_3^*$ chosen with equal probability, and the sign $\pm$ is also chosen with equal probability, retaining those for which all elements of $x^* \pm x'$ were non-negative. The first 5 solutions are shown in Table 7.

2.4.4 Quadratic programming

A quite different method for generating a solution without too many zeroes is to pick a solution that is as close as possible to the fitted cell means produced by the IPF solution. Suppose that the IPF solution is x^*, where x^* satisfies $Ax^* = b$, has non-negative elements, but is not necessarily integral. Then we can seek a vector x that satisfies $Ax = b$, has

Table 7: Five modified tables for Example 8, using a Markov basis.

	Soln 1	Soln 2	Soln 3	Soln 4	Soln 5
x_1	1	1	1	1	1
x_2	4	3	4	4	3
x_3	9	10	9	9	10
x_4	7	7	7	7	7
x_5	0	1	0	0	1
x_6	1	0	1	1	0

integer non-negative elements, and is as close as possible to x^*, in the sense that it minimises $||x - x^*||^2$, the sum of squared differences between the elements of x and the elements of x^*. This is a standard problem in integer quadratic programming: we want to minimize $||x - x^*||^2$ subject to the constraints $Ax = b$, $x \geq 0$, x integral.

Example 9. For the contingency table in Example 6, the IPF solution is

$$x^* = (5.090909, 2.545455, 6.363636, 2.909091, 1.454545, 3.636364),$$

so that we need to minimise

$$(x_1 - 5.090909)^2 + (x_2 - 2.545455)^2 + (x_3 - 6.363636)^2 + (x_4 - 2.909091)^2 \\ + (x_5 - 1.454545)^2 + (x_6 - 3.636364)^2 \qquad (5)$$

where $x = (x_1, \ldots, x_6)$ is a solution of $Ax = b$. Using the lattice basis for this problem, and the LP solution $(0, 4, 10, 8, 0, 0)$, any integral solution to $Ax = b$ must be of the form

$$\begin{aligned} x_1 &= u + v, \\ x_2 &= 4 - u, \\ x_1 &= 10 - v, \\ x_1 &= 8 - u - v, \\ x_5 &= u, \\ x_6 &= v, \end{aligned}$$

where u and v are integers. The requirement that the elements of x are non-negative implies that

$$4 \geq u, \; 8 \geq u + v, u \geq 0, \text{ and } v \geq 0.$$

Minimising (5) by direct search over this region gives the solution $(5, 3, 6, 3, 1, 4)$ corresponding to $u = 1, \; v = 4$. (This is also the result of rounding the IPF solution, but this in general won't work.)

2.5 Limitations

The methods described in this section are satisfactory for small problems, but will struggle with big problems. Integer programming (and particularly quadratic integer programming) is very demanding of computer time and memory, as the branch and bound trees can grow very large. The calculation of Groebner bases is likewise a difficult computational problem, although Diaconis and Sturmfels (1998) and Dobra (2003) discuss ways to calculate these in special cases. In contrast, IPF works well for quite big tables, but cannot of course find tables that exactly match the margins. In the next section, we will apply the methods described in this section to more realistic problems to get a sense of the limits of the IP approach.

3 DATA STRUCTURES, ALGORITHMS AND SOFTWARE

In this section, we define the data structures used in our software, and discuss a set of R functions which implement the methods described in Section 2. We apply these methods to more realistic examples, and explore the limits of the IP and IPF approaches.

3.1 Data Structures

There are two data structures in R suitable for representing contingency tables, *arrays* and *data frames*. We discuss each in turn, and then describe some R functions for converting from one representation to the other.

3.1.1 Arrays

An array in R can have one or more dimensions, and is indexed by one or more subscripts, one per dimension. Arrays can represent a table directly, as the cell count corresponding to A_1 at level i_1, A_2 at level i_2, and A_K at level i_K, can be stored in the $(i_1, i_2, \ldots, i_k)$ position in the array. For example, the Agresti data can be stored in a $2 \times 2 \times 2$ array `y`, with the array element `y[i1,i2,i3]` storing the count of the (i_1, i_2, i_3) cell.

Arrays are created in R using the `array` function. We need to supply several pieces of information to this function. First, we must specify the list of factors, in some fixed order, say alphabetical. Second, for each factor, we need to specify an ordered set of factor levels. Finally, we need to supply the vector of cell counts. Given the order of the factors and the ordering of the levels for each factor, we can establish a reverse lex order for the cells. The cell counts must be supplied in this order.

The array function has three arguments. The first is the cell counts in the appropriate order, as detailed above. The second is the vector of dimensions $I_1, \ldots, I_K$. The third encodes the factor name and level information, in the form of an R list. The following example illustrates the procedure.

Example 10. Consider the Agresti example on substance abuse by US teenagers. There are three factors, A, C, M, each of which has two levels, "Yes" and "No". With the factors in alphabetical order, and the levels in the order "Yes", "No", the counts in reverse lex order are 911, 3, 44, 2, 538, 43, 46, 279. The following R code creates the array and prints it:

```
 > counts = c(911, 3, 44, 2, 538, 43, 456, 279)
 > names.and.levels = list(A = c("Yes", "No"), C = c("Yes", "No"),
        M = c("Yes", "No"))
 > y =  array(counts, c(2,2,2), dimnames=names.and.levels)
 > y
 , , M = Yes

     C
A     Yes No
```

```
  Yes 911 44
  No    3  2

, , M = No

     C
A      Yes  No
  Yes 538  456
  No   43  279
```

Individual counts can be referred to:

```
> y[1,2,2]
[1] 456
```

Apart from the level information, all that needs to be stored is the vector of counts, which has $I_1 \times I_2 \times \cdots \times I_K$ elements.

3.1.2 Data frames

A data frame is the standard data structure in R for storing data in traditional row and column form, with rows storing data on individuals, and columns storing data on variables. To represent a contingency table as a data frame, we let each row correspond to a cell of the table. One variable, `counts` say, stores the cell counts, and there are a further K variables storing the factor level combinations. Thus, each row of the data frame has a cell count, plus the factor level combinations that identify the cell. For the Agresti data, we have

```
  counts   A   C   M
1    911 Yes Yes Yes
2      3  No Yes Yes
3     44 Yes  No Yes
4      2  No  No Yes
5    538 Yes Yes  No
6     43  No Yes  No
7    456 Yes  No  No
8    279  No  No  No
```

Note the row labels. These can be changed; for example we could label the rows with the factor level combinations:

```
    counts   A   C   M
111    911 Yes Yes Yes
211      3  No Yes Yes
121     44 Yes  No Yes
221      2  No  No Yes
112    538 Yes Yes  No
```

```
212      43  No Yes  No
122     456 Yes  No  No
222     279  No  No  No
```

Data frames are usually created by reading in the data from a text file. They can also be created directly from the appropriately ordered vector of counts, using the R function `expand.grid`. The following code does this.

```
> counts = c(911, 3, 44, 2, 538, 43, 46, 279)
> ACM = data.frame(counts=counts, expand.grid(A=c("Yes", "No"),
  C = c("Yes", "No"), M = c("Yes", "No")),
  row.names = c("111","211","121","221","112","212","122","222"))
```

In contrast to arrays, the identification of the factor levels with the counts is made explicitly. The advantage of this is that we do not need to include zero counts in the data frame. This may result in substantial saving in space if the table is sparse. Unlike arrays, there is no need to have any particular ordering of the rows.

3.1.3 Standard form

Not all data frames represent tables. However, if a data frame has a single count variable, having non-negative integer values, and all the other variables are factors, then the data frame corresponds to a unique table, up to the order of the factors. We will say a data frame is in *standard form* if the count variable is the first variable, the factors are arranged alphabetically, and the rows are in reverse lex order.

3.1.4 Converting between formats

We have provided some R functions to convert between data frames and arrays. To convert a data frame into an array, we must restore the zero counts in the proper places. To convert from an array into a data frame in standard form, we eliminate the zero counts. There is also functions to test if a data frame represents a table, and to convert such a data frame into a data frame in standard form. The functions are described in Table 8, and are fully documented in Appendix A.1. There is also an R class "table": and some functions (`is.table`, `as.table`, `as.data.frame`) for converting. However, these do not implement the idea of a standard table. Moreover, it is possible for arrays with negative elements to be tables in the R sense, so we make no use of the R "table" class in this report.

3.2 Integer programming implementations

3.2.1 The R implementation

The R implementation of an integer program solver is based on the freeware program `lp_solve`, originally developed by Michel Berkelaar at Eindhoven University of Technology, and subsequently developed further by several contributors. The current version is 5.5.0.9, and is released under a GNU Lesser General Public Licence. The R interface is by Micheal Berkelaar and Sam Buttrey, and is described in Buttrey (2005).

Table 8: R functions for converting between formats.

Function name	Argument	Purpose	Returns
`is.table.df`	Any R object	Checks if argument is a data frame which represents a table	TRUE or FALSE
`as.standard`	A data frame	Checks if argument is a data frame which represents a table. If so, converts it into a data frame in standard form	A data frame in standard form.
`table2df`	An array	Checks an array is non-negative and if so converts it into data frame in standard form	A data frame in standard form
`df2table`	A data frame	Checks if a data frame represents a table and if so converts it into an array	An array

Like most packages, `lp_solve` uses the branch and bound algorithm to solve the IP. It is a sophisticated package, which can be used as a stand-alone program, or as a collection of library routines which can be embedded in user-written software. In addition, `lp_solve` has interfaces to several other packages such as Matlab and AMPL as well as R. In this respect, `lp_solve` is similar to the commercial package CPLEX, which we describe more fully below in Section 3.2.3. Since CPLEX appears to be more powerful than `lp_solve`, we do not discuss the generic `lp_solve` package further. However, the implementation using CPLEX described in Section 3.2.3 could also be recreated using `lp_solve` if a non-commercial alternative to CPLEX is desired.

The R implementation is particularly simple to use, and consists of a single function `lp`. This is in the package `lpSolve` which must be loaded first. We need only supply the bare minimum of information to `lp` to specify the problem; the vector c defining the objective function $c^T x$, the constraint matrix A, the right hand side vector b, the direction of the constraints (in our case always "="), and the requirement that all variables be integral. The full calling sequence is (taken from the R help pages)

Usage:

```
lp (direction = "min", objective.in, const.mat, const.dir,
    const.rhs,  transpose.constraints = TRUE, int.vec,
    presolve=0, compute.sens=0)
```

Arguments:

`direction:` Character string giving direction of optimization: "min" (default) or "max."

`objective.in:` Numeric vector of coefficients of objective function

`const.mat:` Matrix of numeric constraint coefficients, one row per constraint, one column per variable (unless transpose.constraints = FALSE; see below).

`const.dir`: Vector of character strings giving the direction of the constraint: each value should be one of "<," "<=," "=," "==," ">," or ">=."

`const.rhs`: Vector of numeric values for the right-hand sides of the constraints.

`transpose.constraints`: By default each constraint occupies a row of const.mat, and that matrix needs to be transposed before being passed to the optimizing code. For very large constraint matrices it may be wiser to construct the constraints in a matrix column-by-column. In that case set transpose.constraints to FALSE.

`int.vec`: Numeric vector giving the indices of variables that are required to be integer. The length of this vector will therefore be the number of integer variables.

`presolve`: Numeric: presolve? Default 0 (no); any non-zero value means "yes." Currently ignored.

`compute.sens`: Numeric: compute sensitivity? Default 0 (no); any non-zero value means "yes."

Not all of these arguments need be set for our application. Generic code in our case is is

```
> library(lpSolve)
> stuff = lp ("max", objective.in = A[1,], A,
           const.dir=rep("=",dim(A)[1]), b, int.vec=1:dim(A)[2])
```

assuming the constraint matrix `A` and the RHS vector `b` have already been defined.

The main drawbacks of this implementation are (i) There is no provision for tuning the branch and bound algorithm to improve performance, and (ii) the constraint matrix A must be specified in the form of an R matrix array. If A is sparse, as in our application, this is wasteful of storage, and puts limitations on the size of problem that can be handled. Still, for small to medium problems, it is an effective and simple method.

As noted in the Introduction and Section 2, since the constraint matrix A is the transpose of the model matrix, we can use the R modelling software to compute the constraint matrix. We have written an R function `generate.data.lpSolve` based on the R modelling software and `lpSolve` to generate a data set matching a supplied set of margins. The function accepts as input either multiple data frames containing the margins, or a list of such data frames. The data frames need not be in standard form, as the function coerces them into standard form if this is possible, and terminates if it is not. The output is a data frame in standard form containing a complete table matching the margins. The function calculates the constraint matrix A and the RHS vector of marginal counts b and calls `lpSolve` to solve the resulting integer program. It is formally documented in Appendix A.1. The following examples illustrate its use.

Example 11. For our first example, we explore the limits of the `lpSolve` approach by generating a series of complete tables, calculating certain marginals, and running the function `generate.data.lpSolve` to generate a matching complete table. The results are shown in Table 9. We see from this table that the R approach is capable of solving our problem for some small to moderate tables. We were unable to push this approach

Table 9: Results of running `generate.data.lpSolve`.

Numberof factors (K)	Dimensions ($I_1, \ldots, I_K$)	Number of cells	Dimension of A	Margins supplied
3	3,4,2	24	18×24	All 2-dim
5	3,4,2,4,6	576	189×576	All 2-dim
6	2,3,5,6,2,5	1800	1640×1800	All 5-dim
7	2,3,5,3,2,5,4	3600	3408×3600	All 6-dim
7	2,3,5,5,2,5,4	6000	5616×6000	All 6-dim

beyond tables having about 6000 cells, but a computer with more memory could no doubt do better.

Example 12. Our next example is taken from the 2001 Census. Consider the following three variables from the census, classifying the working age population (i.e. 15+):

`AgeGroup`: Age group, one of 15-19, 20-24, 25-29, 30-34, 35-39, 40-44, 45-49, ,50-54, ,55-59, 60-64, 65+. (11 levels)

`HoursWork`: Hours worked per week, one of 1-4, 5-9, 10-14, 15-19, 20-24, 25-29, 30-34, 35-39, 40-44, 45-49, 50-54, 55-59, 60-64, 65-69, 70-74, 75+, (16 levels)

`Sex`: gender, one of M or F. (2 levels)

The complete table has $11 \times 16 \times 2 = 352$ cells. Suppose we have the three two-dimensional marginal tables, in the form of comma-delimited csv files `AgeGroup.HoursWork.csv`, `AgeGroup.Sex.csv` and `HoursWork.Sex.csv`. These csv files should be in a suitable form with rows corresponding to cells and columns corresponding to variables, all but one of which should be coded using alphabetic codes so that the variables will be recognised as factors. The remaining variable should have non-negative numeric values, representing the counts. In addition, there should be a header row giving the variable names. See Section 4 for a description of a library of such tables. An easy way to create such a csv file is to create a data frame that represents a table (all variables except one factors, the remaining variable having non-negative integer values) and then save it using the R command `write.table`, as in

```
write.table(my.df, "my.csv", row.names=F, sep=",")
```

Note the use of `row.names=F` to suppress the writing of case labels. An R script to read the data, make a list of data frames and create a matching 3-dimensional table is

```
AgeGroup.HoursWork.df = read.csv("AgeGroup.HoursWork.csv")
AgeGroup.Sex.df = read.csv("AgeGroup.Sex.csv")
HoursWork.Sex.df = read.csv("HoursWork.Sex.csv")
df.list = list(AgeGroup.HoursWork.df, AgeGroup.Sex.df,
           HoursWork.Sex.df)
AgeGroup.HoursWork.Sex.df = generate.data.lpSolve(df.list)
```

This creates a three dimensional table `AgeGroup.HoursWork.Sex.df`:

```
> AgeGroup.HoursWork.Sex.df
        y AgeGroup HoursWork    Sex
1    6843    15-19       1-4 Female
2    1818    20-24       1-4 Female
3    1743    25-29       1-4 Female
4    2808    30-34       1-4 Female
5    3141    35-39       1-4 Female
6    2394    40-44       1-4 Female
7    1743    45-49       1-4 Female
10    570    60-64       1-4 Female
12   7959    15-19     10-14 Female
13   7572    20-24     10-14 Female
14   4680    25-29     10-14 Female
15   6555    30-34     10-14 Female
16   7938    35-39     10-14 Female
17   6921    40-44     10-14 Female
18   5337    45-49     10-14 Female
..... 352 lines in all
```

Example 13. Here is another census example. This time we classify the adult population 15+ according to the variables

`AgeGroup`: Age group, one of 15-19, 20-24, 25-29, 30-34, 35-39, 40-44, 45-49, 50-54, 55-59, 60-64, 65-69, 70-74, 75-79, 80-84, 85+ (15 levels),

`Ethnic`: Ethnicity, one of Asian, European, Maori, Other, PI (5 levels),

`Qual`: Highest educational qualification, one of NoQual, FifthForm, SixthForm, HigherSchool, OtherNZ, Overseas, BasicVocational, SkilledVocational, IntermediateVocational, AdvancedVocational, BachelorDegree, HigherDegree, Other (13 levels),

`Sex`: gender, one of M or F (2 levels).

The complete four-dimensional table has $15 \times 5 \times 13 \times 2 = 1950$ cells. We will find a table that matches the two-dimensional margins. Once again, assume that these margins are in suitable csv files, say `AgeGroup.Ethnic.csv`, `AgeGroup.Ethnic.csv`, `AgeGroup.Sex.csv`, `Ethnic.Qual.csv`, `Ethnic.Sex.csv` and `Qual.Sex.csv`. The following R script creates a matching complete table:

```
> AgeGroup.Ethnic.df = read.csv("AgeGroup.Ethnic.csv")
> AgeGroup.Qual.df = read.csv("AgeGroup.Qual.csv")
> AgeGroup.Sex.df = read.csv("AgeGroup.Sex.csv")
> Ethnic.Qual.df = read.csv("Ethnic.Qual.csv")
> Ethnic.Sex.df = read.csv("Ethnic.Sex.csv")
> Qual.Sex.df = read.csv("Qual.Sex.csv")
> df.list = list(AgeGroup.Ethnic.df, AgeGroup.Qual.df, AgeGroup.Sex.df,
             Ethnic.Qual.df, Ethnic.Sex.df, Qual.Sex.df)
> AgeGroup.Ethnic.Qual.Sex.df = generate.data.lpSolve(df.list)
```

3.2.2 The SAS implementation

The SAS procedure PROC LP has capabilities somewhat superior to those of lp_Solve. We have written an R function to create a suitable SAS script and input data set. The script uses PROC LP, and the data are in a sparse matrix form. The R function then runs the SAS script on this data set, to produce a solution which can then be used to create a complete table in the form of an R data frame in standard form.

Our first task is to generate the required SAS data set using the SAS sparse matrix format. This data set has four variables with special SAS names: `_type_`, `_column_`, `_row_` and `_coefficient_`. Each non-zero entry in the constraint matrix and right hand side generates a separate line in the data set, as do the coefficients of the objective function. In addition, there are lines specifying the type of the solution variables (integer or real), and upper bounds for each solution. We set these to be the table total. Note that zero entries in the constraint matrix and RHS vector are ignored, and do not appear in the data set. The variable `_type_` refers to the type of the coefficient, and has value `max` for the objective value coefficients, value `eq` for constraints, `rhs` for right hand sides, `upperbd` for the upper bounds, and `integer` for the type of solution variable.

Rows and columns are referred to symbolically by character values. We use `r1`, `r2`, . . . to denote rows, and `x1`, `x2`, . . . to denote columns. The RHS is denoted by `rhs`, and the rows representing the objective function, the upper bounds and the variable types are denoted by `obj`, `upper` and `int` respectively. The variable `_coef_` stores the numerical value of the appropriate coefficient. Note that the values of the variable `_type_` are keywords and must be as shown, but the row and column names can be chosen by the user. The following example illustrates the form of the data set.

Example 14. The two-dimensional problem discussed in Example 6 involves maximizing $x_1 + x_2 + \cdots + x_6$, subject to

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} 22 \\ 8 \\ 4 \\ 10 \end{bmatrix}$$

where $x_1, \ldots, x_6$ are non-negative and integral. The SAS input data file in "sparse" form is (with variables in the order `_type_`, `_column_`, `_row_` and `_coefficient_`)

```
max x1 obj 1
max x2 obj 1
max x3 obj 1
max x4 obj 1
max x5 obj 1
max x6 obj 1
eq  x1 r1 1
eq  x2 r1 1
eq  x3 r1 1
eq  x4 r1 1
eq  x5 r1 1
eq  x6 r1 1
rhs _rhs_ r1 22
eq  x4 r2 1
eq  x5 r2 1
eq  x6 r2 1
rhs _rhs_ r2 4
eq  x2 r3 1
eq  x5 r3 1
rhs _rhs_ r3 10
eq  x3 r4 1
eq  x6 r4 1
rhs _rhs_ r4 8
upperbd x1 upper 22
upperbd x2 upper 22
upperbd x3 upper 22
upperbd x4 upper 22
upperbd x5 upper 22
upperbd x6 upper 22
integer x1 int 1
integer x2 int 1
integer x3 int 1
integer x4 int 1
integer x5 int 1
integer x6 int 1
```

Given the sparse matrix representation of the constraint matrix used by CPLEX (see below in Section 3.2.3), it is a simple matter to write an R script that will generate both a "sparse" SAS input data file in the format described above, and also a SAS script to run the IP. The R script will then run the SAS program it has created and assemble the result into a data frame. The R function `generate.data.SAS` performs these tasks.

This implementation can handle quite large problems. We have successfully run problems with 20,000 cells, taking a few minutes to obtain a solution.

Example 15. The following code generates a data set for the margins in Example 6:

```
df.list = list(AgeGroup.Ethnic.df, AgeGroup.Qual.df, AgeGroup.Sex.df,
             Ethnic.Qual.df, Ethnic.Sex.df, Qual.Sex.df)
AgeGroup.Ethnic.Qual.Sex.df = generate.data.sas(df.list)
```

3.2.3 The CPLEX implementation

In this section we describe a more powerful IP solver. The CPLEX package is a sophisticated and expensive commercial package which represents the state-of-the-art in mathematical programming. Like `lp_solve`, it can be used in conjunction with other packages such as AMPL and Matlab, as a stand-alone program with its own programming interface, or as a collection of C library routines that can be embedded in C or Fortran programs written by the user. We will use it in the latter mode. CPLEX can solve a variety of mathematical programming problems. We will use it for just two. First, we will use it as a more powerful alternative to `lp_solve`, to solve the integer programming problem

$$\text{maximize } c^T x$$

subject to

$$Ax = b,\ x \geq 0,\ x \text{ integral}.$$

We will also use it to solve the quadratic integer program

$$\text{minimize } \tfrac{1}{2} x^T D x - \ c^T x$$

where D is positive-definite, subject to

$$Ax = b,\ x \geq 0,\ x \text{ integral},$$

which we require to implement the table harmonisation algorithm described in Section 3.4, and also for modifying tables to avoid too many empty cells as described in Section 2.4.

A feature of CPLEX is its ability to handle sparse matrices in a compact way. In this way, we avoid having to reserve storage for the whole of the constraint matrix A. Even in the small examples considered in Table 9, the matrix A had up to $5,616 \times 6,000 = 33,696,000$ entries. CPLEX allows for two ways of representing a matrix, a row form and a column form. We shall describe and use the row form. This requires three vectors, which are referred to in the CPLEX documentation as `rmatbeg`, `rmatind` and `rmatval`. The non-zero elements of A are placed in row order into `rmatval`. The columns of A in which these elements lie are represented by the corresponding elements of `rmatind`, so that these two vectors have length `nzcnt`, the number of non-zero elements of A. Finally, the position in `rmatind` corresponding to the start of row i of A is recoded in the ith element of `rmatbeg`, so the length of `rmatbeg` is the number of rows of A. As a final complication, since CPLEX routines are coded in C, all arrays start at position 0, rather than position 1.

Example 16. Consider the matrix A of Example 6, given by

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}.$$

There are 13 non-zero elements, all 1, so that the array `rmatval` consists of 13 1's in this example. There are 6 columns, which in C fashion are labelled 0,1,2,3,4,5. The columns containing the 1's are the elements of `rmatind`, these are

$$0, 1, 2, 3, 4, 5, \quad 3, 4, 5, \quad 1, 4, \quad 2, 5.$$

The positions in this sequence where a new row starts are 0, 6, 9, 11; these are the elements of `rmatbeg`.

To solve an IP or integer QP in CPLEX using the CPLEX library routines, we must write a C or Fortran program to

- Read in the problem elements: the matrix D and vector c defining the objective function, the RHS vector b, and the constraint matrix A, coded in the form described above. Recall that in our application, we use a row of the constraint matrix A as our vector c. In our case this will be a vector of ones, as this is always the first row of A. The matrix D will always be diagonal in the cases we consider;
- Initialize the problem, reserve memory;
- Populate the data structures;
- Perform the optimisation;
- Extract the solution vector;
- Shut down the CPLEX environment.

The CPLEX library functions we use are described in Table 10. A full description of the calling sequences can be found in the CPLEX documentation. We have written some C and Fortran code to perform the tasks listed above, using the library routines in Table 10. These programs are listed in Appendix A.4. To create the necessary R interface to these routines, we have written an R function `generate.data.cplex`. This has similar inputs and outputs to `generate.data.lpSolve`, but is different internally. To avoid calculating the constraint matrix A explicitly, we do not use the R modelling software to calculate a model matrix. Rather, we calculate the sparse matrix representation of A (i.e. the arrays `rmatbeg`, `rmatind`) directly from the "dummy variable" representation of the model matrix described in Section 2.2.4. This is implemented in a mixture of R and Fortran code. The R function then calls a function written in C (using the R `.C` function) to interface with the CPLEX Callable Library.

The function `generate.data.cplex` can also be used to calculate a table that satisfies the constraints $Ax = b$, and is the closest quadratic approximation to the IPF solution, thus implementing the method described in Section 2.4.4. This is done by means of the argument `objective`.

Example 17. Consider again the margins in Example 13. To find the table satisfying $Ax = b$ using the IP approach, we can use the R code

```
df.list = list(AgeGroup.Ethnic.df, AgeGroup.Qual.df, AgeGroup.Sex.df,
               Ethnic.Qual.df, Ethnic.Sex.df, Qual.Sex.df)
AgeGroup.Ethnic.Qual.Sex.df = generate.data.cplex(df.list)
```

Table 10: CPLEX functions.

Function	Purpose
CPXopenCPLEX	Open CPLEX environment
CPXgeterrorstring	Get error message
CPXsetintparam	Set integer parameters
CPXcreateprob	Set up new problem
CPXaddrows	Add rows to the constraint matrix, RHS
CPXchgobj	Define vector c in the objective function
CPXchgctype	Define all solutions to be integer
CPXcopyqpsep	Define diagonal elements of matrix D
CPXlpwrite	Write out problem description to a file
CPXmipopt	Solve linear or quadratic IP
CPXgetstat	Get status of solution
CPXgetx	Extract solution
CPXfreeprob	Delete problem information, free up space
CPXcloseCPLEX	Close CPLEX environment

To find the closest quadratic approximation to the IPF solution, we use the code

```
AgeGroup.Ethnic.Qual.Sex.quad.df = generate.data.cplex(df.list,
                                             objective="quadratic")
```

3.3 Iterated proportional fitting

The theory of iterated proportional fitting was described briefly in Section 2.2.3. In this section we describe an implementation of this algorithm which will allow us to generate synthetic data sets whose margins approximately match the supplied marginal tables. Although we cannot obtain an exact match with this technique, we can handle much larger tables (say up to a million cells).

Given a set of marginal tables, we can use IPF to obtain a complete table of fitted means determined by the log-linear model that corresponds to the given set of marginal tables. We can then generate complete tables by randomly generating cell counts from Poisson distributions with these means. This will result in a table with a random total sample size. Alternatively, if a table with a fixed sample size is desired, we can convert the table of means into a table of probabilities by dividing each cell mean by the total of all the cell means. We can then draw a sample of the desired size from the multinomial distribution. A third alternative is simply to round the complete table of fitted means.

We have written an R function `generate.data.ipf` which features a Fortran implementation of the IPF algorithm. This runs under Linux, but we have also supplied a Windows version written completely in R. The function calculates the cell probabilities and then generates a table either using multinomial sampling, or by rounding the fitted means. We can control the number of iterations of the IPF algorithm by specifying the maximum number of iterations (default 20), and the tolerance (default 1.0e-8). When the maximum amount of change in mean count at an iteration is less than the tolerance, the iterations stop.

Example 18. Recall the Census example discussed in Example 13. The variables are `AgeGroup`, `Ethnic`, `Qual` and `Sex`. As before suppose we have the six two-dimensional marginal tables, stored in in comma-delimited text files `AgeGroup. Ethnic.csv`, `AgeGroup. Qual.csv`, `AgeGroup.Sex.csv`, `Ethnic.Qual.csv`, `Ethnic.Sex.csv` and `Qual.Sex.csv`. The following R script creates a matching complete table using `generate.data.ipf`:

```
AgeGroup.Ethnic.df = read.csv("AgeGroup.Ethnic.csv")
AgeGroup.Qual.df = read.csv("AgeGroup.Qual.csv")
AgeGroup.Sex.df = read.csv("AgeGroup.Sex.csv")
Ethnic.Qual.df = read.csv("Ethnic.Qual.csv")
Ethnic.Sex.df = read.csv("Ethnic.Sex.csv")
Qual.Sex.df = read.csv("Qual.Sex.csv")

df.list = list(AgeGroup.Ethnic.df, AgeGroup.Qual.df, AgeGroup.Sex.df,
            Ethnic.Qual.df, Ethnic.Sex.df, Qual.Sex.df)
AgeGroup.Ethnic.Qual.Sex.df = generate.data.ipf(df.list)$data
```

Example 19. To explore the limits of this technique, we constructed the series of test problems shown in Table 11. All problems ran successfully, and the error shown is the biggest difference between the margins of the complete table of fitted means, and the supplied margins. The timings shown are in seconds, using a maximum of 20 iterations, multinomial sampling, and the default tolerance.

Table 11: Results of running `generate.data.ipf`

Number of factors (K)	Dimensions ($I_1, \ldots, I_K$)	Number of cells	margins	Time (Seconds)	Error
3	3,4,2	24	All 2-dim	0.01	1.5×10^{-9}
5	3,4,2,4,6	576	All 2-dim	0.06	16.2×10^{-9}
6	2,3,5,6,2,5	1800	All 5-dim	0.18	9.1×10^{-7}
7	2,3,5,3,2,5,4	3600	All 6-dim	0.44	1.0×10^{-3}
7	2,3,5,5,2,5,4	6000	All 6-dim	0.688	4.8×10^{-3}
7	2,5,5,4,6,5,4	24,000	All 3-dim	4.10	3.6×10^{-12}
9	2,5,5,4,6,3,5,4,2	144,000	All 3-dim	47.20	5.8×10^{-11}
10	2,5,5,4,6,3,5,4,2,8	1,152,000	All 3-dim	478.64	9.3×10^{-10}
7	2,5,5,4,6,5,4	24,000	All 4-dim	6.64	2.3×10^{-8}
9	2,5,5,4,6,3,5,4,2	144,000	All 4-dim	99.18	7.9×10^{-8}
10	2,5,5,4,6,3,5,4,2,8	1,152,000	All 4-dim	1205.43	1.2×10^{-8}

3.4 Harmonising tables

The methods we have described assume that the given marginal tables have been obtained by summing over a complete table. However, due to accidentally or deliberately introduced errors, the actual marginal tables we work with may not correspond to the margins of *any* complete table. For example, tables are often modified using base-3 rounding to preserve confidentiality.

If this is the case, then the submargins of one margin may even be incompatible with the submargins of the other. Thus, for example, suppose we have four factors A, B, C and D, and the ABC and ABD marginal tables. If these tables have been derived without error from a complete $ABCD$ table, then the AB submargin of the ABC table must necessarily match the AB submargin of the ABD table. On the other hand, if the tables have been modified, this will not necessarily be the case.

We will say that a set of marginal tables is *compatible* if any submargin derived from one table in the set matches the same submargin derived from any other table in the set. Clearly, compatibility of margins is a necessary (but not sufficient) condition for the existence of a complete table whose margins coincide with the given set.

If no complete table exists whose margins match the supplied set, we need to adjust the supplied margins so that they are the margins of some complete table. We call this process of adjustment *harmonising* the tables. We next describe two methods of doing this, one based on quadratic programming that is satisfactory for small problems, and another based on iterative proportional fitting that will handle much bigger tables.

To describe the first method, let A be the constraint matrix representing the given set of margins, and let b^* be the vector of supplied margins. We want to find a complete table x whose margins Ax match the given margins as closely as possible. If we interpret "as closely as possible" as minimizing $||Ax - b^*||^2$ we can find the adjusted table x by solving the quadratic program

$$\text{Minimize } ||Ax - b^*||^2$$

subject to $x \geq 0$, x integral. An equivalent formulation is to set $z = (x, b)$ and solve

$$\text{Minimize } ||b - b^*||^2$$

subject to $Bz = 0$, $z \geq 0$, z integral, where $B = [A| - I]$. This method simultaneously adjusts the margins and calculates the best-fitting complete table, using a quadratic program in $m + n$ variables, where m and n are the number of rows and columns of A.

However, as quadratic integer programs are extremely demanding of computer time and memory, a more heuristic approach is required for bigger tables. The IPF algorithm may not converge if the supplied margins are not derived from some complete table. Nevertheless, after a few cycles of the algorithm, the fitted cell means will approximate a set of means derived from some related set of marginals that do correspond to a complete table. We simply round the fitted cell means to obtain a complete table, and calculate the margins of this table corresponding to the supplied margins. This gives a rough-and-ready method of harmonising larger tables.

Of course, the simplest and most satisfactory method of producing a set of harmonised marginal tables is to calculate and make available the margins of the real, complete table.

We can also adjust the marginal tables to make them compatible without finding the complete table. This requires solving a smaller quadratic program in as many variables as there are entries in all the marginal tables. However, in this method, unlike the other two above, there is no guarantee that there will exist a complete table matching the adjusted marginals.

To describe this last method, recall from Section 2.2.4 that the constraint matrix depends on the margins and the full table. Also, it is convenient to identify each marginal table with a subset of $1, \ldots, K$, where K is the number of factors in the complete table. Thus, we identify the $A_1A_2A_3$ marginal table with the subset $\{1, 2, 3\}$, and the marginal table $A_2A_3A_5$ with $\{2, 3, 5\}$ and so on. For two such tables (subsets) S_i and S_j say, let A_{ij} be the constraint matrix corresponding to the single margin $S_i \cap S_j$, treating S_i as the complete table. Similarly, let A_{ji} be the constraint matrix corresponding to the single margin $S_i \cap S_j$, treating S_j as the complete table. Then if b_i and b_j are vectors of table counts for S_i and S_j, the two marginal tables will be compatible if $A_{ij}b_i = A_{ji}b_j$.

Thus, to adjust all the margins, we find a set of compatible marginal tables $b_1, \ldots, b_M$ which most closely match the supplied tables $z^* = (b_1^*, \ldots, b_M^*)$, by setting $z = (b_1, \ldots, b_M)$ and solving the quadratic program

$$\text{Minimize } ||z - z^*||^2$$

subject to $Bz = 0$, $z \geq 0$, z integral, where

$$B = \left[\begin{array}{c|c|c|c|c|c} A_{12} & -A_{21} & 0 & \cdots & 0 & 0 \\ A_{13} & 0 & -A_{31} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_{M-1,M} & -A_{M,M-1} \end{array}\right].$$

Example 20. As an example of this technique, consider the three marginal tables introduced in Example 13. These tables are compatible, but if we round them to base 3 they cease to be so:

```
# round tables to base 3
# first define a function

> base3.round=function(x){
+ xx= x%%3
+ ifelse(xx==0,x, ifelse(xx==1, x-1, x+1))
+ }

> AgeGroup.HoursWork.df$y = base3.round(AgeGroup.HoursWork.df$y)
> AgeGroup.Sex.df$y = base3.round(AgeGroup.Sex.df$y)
> HoursWork.Sex.df$y = base3.round(HoursWork.Sex.df$y)

> sum(AgeGroup.HoursWork.df$y)
[1] 1621038
> sum( AgeGroup.Sex.df$y)
[1] 1621017
> sum(HoursWork.Sex.df$y)
[1] 1621011
```

We harmonise them using the R function `harmonise`:

```
> df.list = list(AgeGroup.HoursWork.df, AgeGroup.Sex.df, HoursWork.Sex.df)
> new.df.list = harmonise(df.list, method = "complete")
```

They are now compatible:

```
> sum(new.df.list[[3]]$y)
[1] 1621016
> sum(new.df.list[[2]]$y)
[1] 1621016
> sum(new.df.list[[1]]$y)
[1] 1621016
```

4 APPLICATION TO CENSUS DATA

4.1 Factors, tables and margins

The Table Builder tool on the Statistics New Zealand website allows one to download marginal tables based on variables from the 2001 Census of Population and Dwellings. In this section we describe a subset of the available tables, which we will use to illustrate the methods described in the previous sections of this report. The tables are based on the set of variables listed in Table 12. We will use these variable names in what follows.

Table 12. Description of the variables in the census tables.

Name:	Age
Abbreviation:	`age`
Definition:	Age is the length of time a person has been alive, measured in complete elapsed years. It is measured as the difference between 'date of birth' and '6 March 2001'.
Number of Levels:	18
Levels:	0-4 Years; 5-9 Years; 10-14 Years; 15-19 Years; 20-24 Years; 25-29 Years; 30-34 Years; 35-39 Years; 40-44 Years; 45-49 Years; 50-54 Years; 55-59 Years; 60-64 Years; 65-69 Years; 70-74 Years; 75-79 Years; 80-84 Years; 85 Years and Over
Reference Population:	Census Night Population Count.
Name:	Birthplace
Abbreviation:	`birthplace`
Definition:	Birthplace refers to the country where the respondent was born. A country is the current name, either short or official of a country, dependency or other area of particular geopolitical interest. The term country is defined to include: independent countries recognised by the New Zealand government; units which are recognised geographic areas; administrative subdivisions of Australia and the United Kingdom; overseas dependencies, external territories of independent countries.
Number of levels:	9
Levels:	Asia, Australia, Europe (excl. United Kingdom and Ireland), New Zealand, North America, Not Elsewhere Included, Other, Pacific Islands, United Kingdom and Ireland.
Reference Population:	Census Night Population Count.
Name:	Ethnic Group
Abbreviation:	`ethnic`
Definition:	Ethnicity is the ethnic group or groups that people identify with or feel they belong to. Thus, ethnicity is self-perceived and people can belong to more than one ethnic group. Ethnicity is a measure of cultural affiliation, as opposed to race, ancestry, nationality or citizenship. An ethnic group is a social group whose members have the following four characteristics:

Definition	(Cont) share a sense of common origins; claim a common and distinctive history and destiny; possess one or more dimensions of collective cultural individuality; feel a sense of unique collective solidarity.
Number of levels:	5
Levels:	Asian, European, Maori, Pacific Peoples, Other
Reference Population:	Census Night Population Count.
Other	Total responses means that a person can be counted in more than one ethnic group. In 2001 a person can be counted in up to 6 ethnic groups. Although ethnicity is an hierarchical classification with four levels, only one set of levels is used for this variable, as listed.

Name:	Highest Qualification
Abbreviation:	`highqual`
Definition:	Highest Qualification combines Highest School Qualification and Post-School Qualification to derive a single Highest Qualification by category of attainment. A qualification is a formally recognised award for attainment resulting from a full-time (20 hours per week) learning course of at least three months, or from part-time study for an equivalent period of time or from on the job training.
Number of levels:	13
Levels:	Advanced Vocational Qualification, Bachelor Degree Basic Vocational Qualification, Fifth Form Qualification Higher Degree, Higher School Qualification, Intermediate Vocational Qualification, No Qualification, Not Elsewhere Included, Other NZ Secondary School Qualification, Overseas Secondary School Qualification, Sixth Form Qualification, Skilled Vocational Qualification
Reference population:	Census Usually Resident Population Count aged 15 years and over

Name:	Hours Worked in Employment per Week
Abbreviation:	`hoursperweek`
Definition:	Hours worked in employment is the total number of hours worked in employment per week by all people aged 15 and over who at the time of the census: worked for one hour or more for pay, profit or payment in kind in a job, business, farm or professional practice; or worked without pay for one hour or more in work which contributed directly to the operation of a farm, business or professional practice operated by a relative; or had a job or business they were temporarily absent from.
Number of levels:	9
Levels:	1-9 Hours; 10-19 Hours; 20-29 Hours; Part-time Not Elsewhere Included; 30-39 Hours; 40-49 Hours; 50-59 Hours; 60 Hours or More; Full-time Not Elsewhere Included;
Reference Population:	Employed Census Usually Resident Population Count aged 15 years and over.

Name:	Legal Marital Status
Abbreviation:	`legalms`
Definition:	Marital status is a persons reported status with respect to the marriage laws or customs of the country. There are two types of marital status: Legal Marital Status and Social Marital Status. Legal Marital Status is a persons status with respect to registered marriage. Social Marital Status is a persons status with respect to consensual union. People who are in a consensual union are partnered; people who are not in a consensual union are non-partnered.
Number of levels:	7
Levels:	Never Married; Married (Not Separated); Separated; Divorced; Widowed; Not Elsewhere Included
Reference Population:	Census Usually Resident Population Count aged 15 years and over.

Name:	Number of languages spoken
Abbreviation:	`languages`
Definition:	Number of languages spoken - counts the number of languages spoken as indicated by the respondent. Language spoken -this collects information on whether a person can speak and understand spoken or sign language.
Number of levels:	8
Levels:	None; One Language; Two Languages; Three Languages; Four Languages; Five Languages; Six Languages; Not Elsewhere Included
Reference Population:	Census Usually Resident Population Count.

Name:	Occupation
Abbreviation:	`occupation`
Definition:	An occupation is defined as a set of jobs which involve the performance of a common set of tasks. A job is a set of tasks performed or designed to be performed by one individual. Two jobs are similar if they require the performance of a similar set of tasks or to fulfill the technical requirements of an occupation. Skill is defined as the ability of an individual to perform a set of tasks or to fulfils the technical requirements of an occupation.
Number of levels:	10
Levels:	Legislators, Administrators and Managers; Professionals; Technicians and Associate Professionals; Clerks; Service and Sales Workers; Agriculture and Fishery Workers; Trades Workers; Plant and Machine Operators and Assemblers; Elementary Occupations ; Not Elsewhere Included
Reference Population:	Employed, Census Usually Resident Population Count aged 15 years and over.
Other:	There are some cases where people have been coded to an occupation where in fact they are not really meant to be in the labour force, because occupation is coded independently of work and labour force or employment status. Care should be taken when using this variable particularly if relating it to other Labour Market variables (ie Status in Employment, Work and Labour Force Status, Hours worked, or Total Income) due to the different time frames (census day, last four weeks, previous week, previous year).

Name:	Relgious Affiliation
Abbreviation:	`religion`
Definition:	Religious affiliation is the self-identified association of a person with a religion, denomination or sub-denominational religious group. A denomination is the church or religious sect that forms a sub-group of a religion. Denominations of a particular religion share the same principles but differ from each other in aspects such as the form of worship used and the way in which they are governed.
Number of levels:	10
Levels:	No Religion; Buddhist; Christian; Hindu; Islam/Muslim; Judaism/Jewish; Maori Christian; Spiritualism and New Age Religions; Other Religions; Object to Answering
Reference Population:	Census Usually Resident Population Count.
Other:	Up to four responses are coded.

Name:	Sex
Abbreviation:	`sex`
Definition:	Sex is the distinction between males and females based on the biological differences in sexual characteristics.
Number of levels:	2
Levels:	Male; Female
Reference Population:	Census Night Population Count.

Name:	Social Marital Status
Abbreviation:	`socialms`
Definition:	Marital status is a persons reported status with respect to the marriage laws or customs of the country. There are two types of marital status: Legal Marital Status and Social Marital Status. Legal Marital Status is a persons status with respect to registered marriage. Social Marital Status is a persons status with respect to consensual union. People who are in a consensual union are partnered; people who are not in a consensual union are non-partnered.
Number of levels:	9
Levels:	Partnered (not further defined); Legal Spouse; Other Partnership; Non-partnered (not further defined); Non-partnered, Never Married; Non-partnered, Separated; Non-partnered, Divorced; Non-partnered, Widowed; Not Stated
Reference Population:	Census Usually Resident Population Count aged 15 years and over.

Name:	Total Personal Income
Abbreviation:	`tpincome`
Definition:	Information on total personal income received is collected from individuals in the 2001 Census. It represents the before-tax income for the respondent in the twelve months ended 31 March 2001. Total personal income is collected as an income range rather than an actual dollar income.

Definition:	(Cont)Total personal income is aggregated to form a number of other income outputs including: total household income; total family income; combined parental income for couples with child(ren);total extended family income.
Number of levels:	14
Levels:	Loss; Zero Income; \$1 - \$5,000; \$5,001 - \$10,000; \$10,001 - \$15,000; \$15,001 - \$20,000; \$20,001 - \$25,000; \$25,001 - \$30,000; \$30,001 - \$40,000; \$40,001 - \$50,000; \$50,001 - \$70,000; \$70,001 - \$100,000; \$100,001 or More; Not Stated
Reference Population:	Census Usually Resident Population Count aged 15 years and over.

Name:	Work and Labour Force Status
Abbreviation:	`workstatus`
Definition:	Work and labour force status classifies people aged 15 years and over according to their inclusion or exclusion from the labour force. For people who are employed, it distinguishes whether they are employed full-time (30 hours or more per week) or part-time (fewer than 30 hours per week). Work and labour force status imputation supplies a value for work and labour force status, where this cannot be derived from the labour force information supplied by the respondent. The work and labour force status imputation uses whatever labour force information has been given, and various other responses from the individual (for example, age and income). A work and labour force status is then imputed to equal the known work and labour force status of a similar person.
Number of levels:	5
Levels:	Employed Full-time; Employed Part-time; Unemployed; Not in the Labour Force; Work and Labour Force Status Unidentifiable
Reference Population:	Census Usually Resident Population Count aged 15 years and over.

We refer to a table in our list using a name constructed from its variables. Thus, a table containing the variables age, ethnic and sex will be referred to as the age×ethnic×sex table, and is stored in a file `age_ethnic_sex.txt`. It can be read into R as a data frame using the R statement

```
age_ethnic_sex.df = read.table("age_ethnic_sex.txt", header=T)
```

This produces a data frame in standard form (i.e. counts as the first variable, and the factors age, ethnic and sex in successive alphabetical order.)

The tables fall into three groups, according to which population the tables represent. The three groups are

Group 1: Tables classifying the usually resident census night population.

Group 2: Tables classifying the usually resident census night population aged 15 years and over.

Group 3: Tables classifying the employed usually resident census night population aged 15 years and over.

The complete list of tables is shown in Table 13.

Due to rounding, missing values and other unexplained factors the tables do not have the same totals. In addition, any table with ethnicity as a factor (variable `ethnic`) is problematical, as this classification is not exclusive (an individual may belong to arbitrarily many ethnic groups). We have dealt with this in a very crude manner by normalizing the tables in each group to have approximately the same totals. This was done by multiplying each count by a conversion factor and rounding the result. This "preharmonises" the tables and makes the application of the harmonisation algorithm more efficient.

Note also that the variable age takes different values in each of the three groups. For tables in the first group having age as a factor, the levels are 0-4 Years, 5-9 Years, ..., 80-84 Years, 85 Years and Over. For tables in the second group, the levels for age are 15-19 Years, 20-24 Years, ..., 80-84 Years, 85 Years and Over, while for tables in the

Table 13: The census tables.

Table	File	Table Total	Number of cells
Group 1: Usually resident census population			
age_languages_sex	`age_languages_sex.txt`	3,737,262	288
age_religion_sex	`age_religion_sex.txt`	3,737,261	360
birthplace_ethnic_sex	`birthplace_ethnic_sex.txt`	3,737,260	90
ethnic_languages_sex	`ethnic_languages_sex.txt`	3,737,260	80
ethnic_religion_sex	`ethnic_religion_sex.txt`	3,737,263	100
Group 2: Usually resident census population aged 15 and over			
age_ethnic_highqual_sex	`age_ethnic_highqual_sex.txt`	2,889,609	2080
age_ethnic_sex_tpincome	`age_ethnic_sex_tpincome.txt`	2,889,598	2100
age_legalms_sex	`age_legalms_sex.txt`	2,889,552	180
age_sex_socialms	`age_sex_socialms.txt`	2,889,539	270
ethnic_highqual_sex_tpincome	`ethnic_highqual_sex_tpincome.txt`	2,889,598	2100
ethnic_sex_socialms	`ethnic_sex_socialms.txt`	2,889,552	90
ethnic_sex_tpincome_workstatus	`ethnic_sex_tpincome_workstatus.txt`	2,889,565	700
Group 3: Employed usually resident census population aged 15 and over			
age_ethnic_occupation_sex	`age_ethnic_occupation_sex.txt`	1,727,258	1100
age_hoursperweek_sex	`age_hoursperweek_sex.txt`	1,727,274	198
age_occupation_sex	`age_occupation_sex.txt`	1,727,259	220
highqual_occupation_sex	`highqual_occupation_sex.txt`	1,727,247	260
hoursperweek_sex_tpincome	`hoursperweek_sex_tpincome.txt`	1,727,262	252
occupation_sex_tpincome	`occupation_sex_tpincome.txt`	1,727,233	280

third group the levels are 15-19 Years, 20-24 Years, ..., 60-64 Years, 65 Years and Over.

These three sets of tables are not harmonised. A discussion of how they may be harmonised is given in the next section.

4.2 Generating census data sets

To generate complete tables from these three sets of margins, we use the functions introduced in Section 3.

4.2.1 Group 1 margins

First, we generate a data set based on the complete census night usually resident population. A set of margins relating to this population comprises Group 1 in table 13. The variables for this population are `age` (18 levels), `birthplace` (9 levels), `ethnic` (5 levels), `languages` (8 levels) `religion` (10 levels) and `sex` (2 levels). The complete table has 259,200 cells.

The first step is to harmonise the tables. The methods based on quadratic programming won't cope with such a big table, but the IPF method has no problems. The following code sets up the list of tables and performs the harmonisation. The adjusted set of tables is in the R list `harmonised.group1.list`. We are assuming the input text files containing the margins are in the directory `tables`, this will have to be changed to suit the user's own directory structure.

```
> input.directory = "tables/"  # change this to suit
> group1.files = paste(input.directory, c("age_languages_sex.txt",
    "age_religion_sex.txt", "birthplace_ethnic_sex.txt",
    "ethnic_languages_sex.txt", "ethnic_religion_sex.txt"), sep="")
> group1.list = vector(mode="list", length=length(group1.files))
> for (i in 1:length(group1.files))group1.list[[i]]=
                             read.table(group1.files[i], header=T)
> harmonised.group1.list = harmonise(group1.list, method="ipf")
```

We can get a complete table `group1.df` based on rounding the IPF solution using the code

```
> group1.ipf.df = generate.data.ipf(harmonised.group1.list, result="ro")
```

A table with 259,200 cells is a bit beyond the integer programming methods. However, if we drop the variable `birthplace`, and the birthplace_ethnic_sex margin, the complete table now has only 28,800 cells, within the reach of CPLEX. A complete table (the best quadratic approximation to the IPF solution) that exactly matches the margins can be obtained using the code

```
> group1a.list = list(harmonised.group1.list[[1]],
            harmonised.group1.list[[2]], harmonised.group1.list[[4]],
            harmonised.group1.list[[5]])
> group1a.cplex.df  =
            generate.data.cplex(group1a.list, objective="quad")
```

4.2.2 Group 2 margins

The second set of margins, in Group 2, refers to the usually resident census night population aged 15 and over. The variables for this table are `age` (15 levels), `ethnic` (5 levels), `highqual` (13 levels), `legalms` (6 levels), `sex` (2 levels), `socialms` (9 levels), `tpincome`, (14 levels) and `workstatus` (5 levels), so that the complete table has 7,371,000 cells. If we drop the table ethnic_sex_tpincome_workstatus, we get a complete table with 1,474,200 cells, which is just within the reach of the IPF harmonisation method. The following code accomplishes this.

```
> group2.files = paste(input.directory, c("age_ethnic_highqual_sex.txt",
   "age_ethnic_sex_tpincome.txt", "age_legalms_sex.txt",
   "age_sex_socialms.txt", "ethnic_highqual_sex_tpincome.txt",
   "ethnic_sex_socialms.txt", "ethnic_sex_tpincome_workstatus.txt")
   , sep="")
> group2.list = vector(mode="list", length=length(group1.files))
> for (i in 1:length(group2.files))group2.list[[i]]=
    read.table(group2.files[i], header=T)
> group2a.list = list(group2.list[[1]], group2.list[[2]],
    group2.list[[3]], group2.list[[4]], group2.list[[5]],
    group2.list[[6]])
> harmonised.group2a.list = harmonise(group2a.list, method="ipf")
```

A table can be generated from these margins using the “random” IPF method by the code

```
> group2a.ipf.df = generate.data.ipf(harmonised.group2a.list,
                                                result="random")
```

Again, this table, with 1,474,200 cells, is too big to be generated using the integer programming methods. A smaller table, involving only the variables `age`, `ethnic`, `highqual` and `socialms` can be generated using the quadratic CPLEX method by the code

```
> group2b.list = list(harmonised.group2a.list[[1]],
     harmonised.group2a.list[[4]],  harmonised.group2a.list[[6]])
> group2b.cplex.df = generate.data.cplex(group2b.list, objective="q")
```

4.2.3 Group 3 margins

The final set of margins, in Group 3, refers to the employed usually resident census night population aged 15 and over. The variables for this table are `age` (11 levels), `ethnic` (5 levels), `highqual` (13 levels), `hoursperweek` (9 levels), `occupation` (10 levels), `sex` (2 levels), and `tpincome` (14 levels), so that the complete table has 1,801,800 cells. The margin data is assembled into a list `group3.list` using code similar to that above. This set of margins, corresponding to a complete table with 1.8 million cells, is just a bit beyond the IPF harmonisation method, but if we drop the variable `highqual`, and the table highqual_occupationsex_txt, we get a set of margins corresponding to a much smaller table with only 138,600 cells. We can harmonise the reduced set of 5 tables using the IPF method, and generate a complete table as above.

5 SUMMARY AND CONCLUSIONS

We have described two approaches to the problem of creating synthetic data sets using publicly available marginal tables. In the first approach, we used integer programming to find a complete table which exactly matched the given margins. Three implementations of this idea were described, based on the packages lp_solve, SAS and CPLEX. The first will run under Windows. All implementations run under Linux, provided the relevant package (SAS or CPLEX) is available.

The complete table often has many zero entries, which may not be acceptable. We discussed three methods for modifying tables having this problem, two based on adding solutions of the equation $Ax = 0$ to the table, and the third based on quadratic programming. We have provided an implementation of this last method based on the quadratic programming capabilities of CPLEX.

The second approach is based on iterative proportional fitting. The set of supplied margins corresponds to a hierarchical log-linear model, and the IPF algorithm can be used to calculate a set of cell probabilities or cell means for the complete table, based on this log-linear model. These probabilities can then be used to generate a complete table whose margins will approximately match the supplied margins. Alternatively, the mean cell counts can be rounded.

The methods we describe all require that the tables be harmonised. We described three methods of harmonising tables. The first, based on quadratic programming, produces marginal tables that correspond to a complete table, but is only practical for small to medium problems. The second, heuristic method uses IPF and will handle bigger problems, while the third, again based on quadratic programming, will merely adjust the tables to be compatible. The first and third methods require the CPLEX package.

These methods have been illustrated with numerous examples, and we have described a set of marginal tables sourced from the Statistics New Zealand web site that can be used to generate synthetic data sets whicjh mimic the 2001 Census of Population and Dwellings. We have written a set of R functions to implement our methods, and have illustrated the use of these to create synthetic data from the SNZ tables. These are documented in the text and more formally in the Appendices.

REFERENCES

Agresti, A. (2002). *Categorical Data Analysis, 2nd Ed.* Johns Hopkins University Press, Baltimore.

Bishop, Y.M.M., Fienberg, S.E. and Holland, P.W. (1975). *Discrete Multivariate Analysis.* MIT Press, Cambridge.

Buttrey, S.E. (2005). Calling the lp_solve linear program software from R, S-Plus and Excel. *J. Statistical Software*, **14**.

Christensen, R. (1997). *Log-Linear Models and Logistic Regression.* Springer-Verlag, New York.

Cox, D., Little, J and O'Shea, D. (1996). *Ideals, Varieties and Algorithms.* Springer-Verlag, New York.

Deming, W.E and Stephan, F.F. (1940). On a least squares adjustment of a sampled frequency table when the expected marginal totals are known. *Ann. Math. Statist.*, **11**, 427 – 444.

Diaconis, P and Sturmfels, B. (1998). Algebraic algorithms for sampling from conditional distributions. *Ann. Statist.*, **26**, 363 – 397.

Dobra, A. (2003). Markov bases for decomposable graphical models. *Bernoulli*, **9**, 1093–1108.

R Development Core Team (2005). *R: A language and environment for statistical computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Wolsey, L.A. (1998). *Integer Programming.* Wiley, New York.

A APPENDICES

A.1 Description of the R functions

In this appendix we document the following R functions:

```
as.standard
check.data.frame
df2table
generate.data.cplex
generate.data.ipf
generate.data.lpSolve
generate.data.sas
harmonise
table2df
```

as.standard	package:SNZ	R documentation

Coerces a data frame into standard form

Description:

Coerces a data frame representing a contingency table into a standard form

Usage:

```
as.standard(data.df)
```

Arguments:

`data.df`: a data frame.

Details:

If the data frame represents a contingency table (i.e. if one variable contains counts and the rest are factors) it is transformed into standard form. Otherwise an error message is returned.

Value:

A data frame in standard form.

Example:

```
 ## convert data frame data.df to standard form
data.df = as.standard(data.df)
```

```
check.data.frame              package:SNZ              R documentation
```

Checks a data frame

Description:

Checks if a data frame is in standard form

Usage:

```
check.data.frame(data.df)
```

Arguments:

`data.df`: a data frame.

Value:

A list with elements

- `status`: has value 1 if the argument can be coerced into a data frame representing a contingency table in standard form, and zero otherwise;
- `data`: if status = 1, the data frame representing the contingency table, otherwise NULL;
- `message`: if status = 0, an error message.

Example:

```
 ## check data frame data.df to see if it represents
## a contingency table in standard form
data.df=check.data.frame(data.df)$data
```

```
df2table                      package:SNZ                     R documentation
```

Converts a data frame in standard form into an array

`Description:`

Converts a data frame in standard form into an array, and issues an error message if the input cannot be converted to standard form.

`Usage:`

```
df2table(data.df)
```

`Arguments:`

`data.df`: a data frame.

`Details:`

If the data frame represents a contingency table (i.e. if one variable contains counts and the rest are factors) it is transformed into an array. Otherwise an error message is returned.

`Value:`

An array representing the contingency table.

`Example:`

```
 ## convert data frame data.df to an array my.table
my.table = df2table(data.df)
```

```
generate.data.cplex              package:SNZ              R documentation
```

Generates a contingency table whose margins match a set of specified margins.

`Description:`

Generates a contingency table whose margins match a set of specified margins. The margins are in the form of data frames in standard form, or a list of such data frames

`Usage:`

```
data.df = generate.data.cplex(..., objective="linear", MAXITER=20,
TOL=1.0e-8)
```

`Arguments:`

- ... : Data frames representing the margins, or a single list of such data frames.
- `objective`: Has value "linear", in which case the table will be found by solving the IP with objective function equal to the sum of the table entries, or "quadratic", in which case the best integral approximation to the IPF solution is found. The abbreviations `"l"` or `"q"` suffice.
- `MAXITER`: The maximum number of iterations for the iterative proportional fitting algorithm.
- `TOL`: Stopping criterion for the iterative proportional fitting algorithm.

`Details:`

The margins are assumed to be consistent, as no checks for consistency are made. The data frames are checked to determine if they are in standard form. The contingency table is generated by solving an integer program using the CPLEX callable library. The function also assumes that the shared library `SNZ.so` has been loaded.

`Value:`

A data frame in standard form representing the contingency table.

Example:

```
 array.dim<-c(3,4,2)
y = round(100*runif(prod(array.dim)))
X<-array( y, dim=array.dim, dimnames =
list(A = paste("a",1:array.dim[1], sep=""),
B=paste("b",1:array.dim[2], sep=""),
C=paste("c",1:array.dim[3], sep="")))

# get marginal tables (AB, BC, AC)

AB<-apply(X, c(1,2), sum)
BC<-apply(X, c(2,3), sum)
AC<-apply(X, c(1,3), sum)

AB.df<-table2df(AB)
BC.df<-table2df(BC)
AC.df<-table2df(AC)

df.list = list( BC.df, AC.df, AB.df)

ABC.df = generate.data.cplex(df.list)
```

```
generate.data.ipf              package:SNZ              R documentation
```

Generates a contingency table whose margins match approximately a set of specified margins.

`Description:`

Generates a contingency table whose margins match approximately a set of specified margins, using the iterated proportional fitting algorithm. The margins are in the form of data frames in standard form, or a list of such data frames.

`Usage:`

```
data.df = generate.data.ipf(..., result= "random", MAXITER=20,
          TOL=1.0e-8)
```

`Arguments:`

... : Data frames representing the margins, or a single list of such data frames.

`result`: One of “random” or “rounded”. If the former, a contingency table is generated using multinomial sampling from the table of fitted probabilities. If the latter, the table is obtained by rounding the fitted cell means.

`MAXITER`: The maximum number of iterations for the iterative proportional fitting algorithm.

`TOL`: Stopping criterion for the iterative proportional fitting algorithm.

`Details:`

The margins are assumed to be consistent, as no checks for consistency are made. The data frames are checked to determine if they are in standard form. The contingency table is generated by calculating cell means using the IPF algorithm, and then generating the table by multinomial sampling. The function assumes that the shared library `SNZ.so` has been loaded using `dyn.load`.

`Value:`

A list having four components:

`data:` A data frame in standard form representing the generated contingency table,

`probs:` The array of fitted cell probabilities,

`iter:` The number of IPF iterations taken (Windows version only),

`error:` The maximum relative change in the fitted means at the last IPF iteration.

`Example:`

```
 array.dim<-c(3,4,2)
y = round(100*runif(prod(array.dim)))
X<-array( y, dim=array.dim, dimnames =
list(A = paste("a",1:array.dim[1], sep=""),
B=paste("b",1:array.dim[2], sep=""), C=paste("c",1:array.dim[3], sep="")))

# get marginal tables (AB, BC, AC)

AB<-apply(X, c(1,2), sum)
BC<-apply(X, c(2,3), sum)
AC<-apply(X, c(1,3), sum)

AB.df<-table2df(AB)
BC.df<-table2df(BC)
AC.df<-table2df(AC)

df.list = list( BC.df, AC.df, AB.df)

ABC.df = generate.data.ipf(df.list)
```

```
generate.data.lpSolve            package:SNZ            R documentation
```

Generates a contingency table whose margins match a set of specified margins

Description:

Generates a contingency table whose margins match a set of specified margins. The margins are in the form of data frames in standard form, or a list of such data frames.

Usage:

```
data.df = generate.data.lpSolve(...)
```

Arguments:

... : Data frames representing the margins, or a single list of such data frames.

Details:

The margins are assumed to be consistent, as no checks for consistency are made. The data frames are checked to determine if they are in standard form. The contingency table is generated by solving an integer program using R package lpSolve, which must be present.

Value:

A data frame in standard form representing the contingency table.

Example:

```
 y = round(100*runif(prod(array.dim)))
X<-array( y, dim=array.dim, dimnames =
list(A = paste("a",1:array.dim[1], sep=""),
B = paste("b",1:array.dim[2], sep=""),
C = paste("c",1:array.dim[3], sep="")))

# get marginal tables (AB, BC, AC)

AB<-apply(X, c(1,2), sum)
```

```
BC<-apply(X, c(2,3), sum)
AC<-apply(X, c(1,3), sum)

AB.df<-table2df(AB)
BC.df<-table2df(BC)
AC.df<-table2df(AC)

df.list = list( BC.df, AC.df, AB.df)

ABC.df = generate.data.lpSolve(df.list)
```

```
generate.data.sas              package:SNZ              R documentation
```

Generates a contingency table whose margins match a set of specified margins

Description:

Generates a contingency table whose margins match a set of specified margins. The margins are in the form of data frames in standard form, or a list of such data frames.

Usage:

```
 data.df = generate.data.sas(...)
```

Arguments:

... : Data frames representing the margins, or a single list of such data frames.

Details:

The margins are assumed to be consistent, as no checks for consistency are made. The data frames are checked to determine if they are in standard form. The contingency table is generated by solving an integer program using the SAS system, which must be present. The function also assumes that the shared library `SNZ.so` has been loaded. The function removes all the generated SAS files except the file `lp.log`, which should be checked to see if the integer program was successfully solved.

Value:

A data frame in standard form representing the contingency table.

Example:

```
 y = round(100*runif(prod(array.dim)))
X<-array( y, dim=array.dim, dimnames =
list(A = paste("a",1:array.dim[1], sep=""),
B = paste("b",1:array.dim[2], sep=""),
C = paste("c",1:array.dim[3], sep="")))
```

```
# get marginal tables (AB, BC, AC)

AB<-apply(X, c(1,2), sum)
BC<-apply(X, c(2,3), sum)
AC<-apply(X, c(1,3), sum)

AB.df<-table2df(AB)
BC.df<-table2df(BC)
AC.df<-table2df(AC)

df.list = list( BC.df, AC.df, AB.df)

ABC.df = generate.data.sas(df.list)
```

```
harmonise                  package:SNZ                  R documentation
```

Modifies a set of marginal tables to make them consistent

Description:

Takes a set of marginal tables and ajusts them so that the adjusted set is consistent. The margins are in the form of data frames in standard form, or a list of such data frames. The result is a list of the adjusted data frames.

Usage:

```
 adjusted.list = harmonise(..., method = "ipf", MAXITER = 20, TOL
= 1.0e-8)
```

Arguments:

- ... : Data frames representing the margins to be adjusted, or a single list of such data frames.
- `method`: One of “complete”, “partial”, “ipf” corresponding to the three harmonisation methods discussed in the text.
- `MAXITER`: The maximum number of iterations for the iterative proportional fitting algorithm.
- `TOL`: Stopping criterion for the iterative proportional fitting algorithm.

Details:

The data frames are first checked to determine if they can be coerced standard form. The harmonisation adjustment is made in one of three ways depending on the value of `method`. If `method = "complete"`, then a complete table is found whose margins match those supplied, using a quadratic criterion. If `method = "partial"`, the tables are adjusted to be compatible as described inn the text. Finally, if `method = "ipf"` (the default), then a complete table is found whose margins match those supplied, using the IPF method. The first two methods assume that the using the CPLEX Callable Library is installed. The function also assumes that the shared library `SNZ.so` has been loaded using `dyn.load`.

Value:

A list of the adjusted data frames

Example:

```
df.list = list( BC.df, AC.df, AB.df)

adjusted.df.list = harmonise(df.list)
```

```
table2df                    package:SNZ                  R documentation
```

Converts an array to a data frame in standard form

`Description:`

Converts an array into a data frame in standard form, and issues an error mesage if the input is not capable of conversion to standard form (e.g. if there are negative or non-integer entries in the array.)

`Usage:`

```
table2df(my.table)
```

`Arguments:`

`my.table`: a multidimensional array.

`Details:`

If the array represents a contingency table (i.e. if the entries are non-negative integers) it is transformed into a data frame in standard form. Otherwise an error message is issued. The count variable in the data frame is named `y`.

`Example:`

```
 ## convert array my.table to a data frame data.df in standard form
data.df = table2df(my.table)
```

A.2 Installation of the R functions

We have provided three versions of the R functions.

A.2.1 Three versions

The first set of functions, in file `SNZ.Windows.zip`, includes the functions `as.standard`, `check.data.frame`,`df2table`, `generate.data.lpSolve`, `generate.data.ipf`, `harmonise`, and `table2df`. These make no use of the CPLEX Callable Library, and may be run under both the Windows and Linux versions of R.

The second set, in file `SNZ.noCPLEX.tar`, is designed to run under Linux, and uses some Fortran code. To use these functions, you will need to create an R shared library `SNZ.so`. See below for instructions for doing this. This set contains the same set of functions as the Windows version, plus the function `generate.data.sas`. To use this function, you will need to have SAS installed.

The third set, in file `SNZ.CPLEX.tar`, contains all the functions, and is also designed to run under Linux. It uses some Fortran code and C code, and requires the use of the CPLEX Callable Library. Again, a shared library `SNZ.so` will have to be created.

A.2.2 The Windows version

To use Windows versions of these functions, simply unzip the file `SNZ.Windows.zip`, place the contents (including the R source code in file SNZ.r) in some suitable folder. Then invoke R, choose "Source R code..." from the R file menu, and select the file `StatsNZ`.r. If you want to use the function `generate.data.lpSolve`, you will also have to load the library `lpSolve` by selecting "Install package(s)..." from the packages menu and following the dialog. You are then ready to go.

A.2.3 The non-CPLEX Linux version

Extract the files from the tar file supplied and place them in some suitable directory. Set your working directory to this directory, and create the shared library `SNZ.so` by running the commands in the file `make.R`. Do this by typing

```
sh make.R
```

which will create the shared library `SNZ.so`. Then, to load the shared library and the R functions, invoke R and type

```
source("SNZ.r")
```

This assumes that the shared library is in the directory from which you invoked R. If not, an error message will be given. If you want to invoke R from one directory and have the shared library in another, you will have to edit the first executable line `dyn.load("SNZ.so")` in the file `StatsNZ.r`. As before, if you want to use the function `generate.data.lpSolve`, you will need the `lpSolve` package loaded in R.

A.2.4 The CPLEX Linux version

To install this, the procedure is the same as for the non-CPLEX version. The only complication is that location of the CPLEX Callable Library must be specified in the files `make.R` and `Makevars`. On my system, the library is in directory

```
/usr/local/ilog/cplex101/lib/x86_rhel4.0_3.4/static_pic
```

This string must be replaced by the one appropriate to your system in the file `Makevars`. In addition, the C code requires some CPLEX include files, which in my system are in the directory

```
/usr/local/ilog/cplex101/include
```

Again, this string should be replaced by the name of the appropriate directory in the file make.R.

A.3 R function listings

In this appendix, we provide a listing of the R functions in the file `StatsNZ.r` supplied with the CPLEX version. The non-CPLEX and Windows versions have fewer functions.

```
##############################################################################

#  Linux Cplex version  Last Modified  25 Jan 2007

# First load shared library

dyn.load("SNZ.so")

# Functions

##############################################################################
#
check.data.frame=function(data.df){

# checks that the data frame is in standard form
# returns a list with elements status
# (=0 if arg cannot be coereced into the standard form, otherwise 1)
# and data ( the possibly modified data frame in standard form,
# if status =1, otherwise NULL)
# standard form is a data frame with the first column a numeric variable,
# and the other variables factors.

# The numeric variable must be a non-negative vector of counts.

# first check argument is a data frame, quit if not

if(!is.data.frame(data.df))return (list(status=0, data=NULL,
 message = "Not a data frame"))

n.var=dim(data.df)[2]
is.number=logical(n.var)
for(j in 1:n.var) is.number[j]=is.numeric(data.df[,j])

# check if exactly one is numeric
if(!(sum(is.number)==1)) return (list(status=0, data=NULL, message =
"More than one numeric variable"))

# check to see if y is non-neg
if(!all(data.df[,is.number]>=0)) return (list(status=0, data=NULL,
message ="Cannot have non-negative counts"))

# check that all factors have more than one level
use.cols = (1:n.var)[!is.number]
max.levels = numeric(length(use.cols))
for(i in 1:length(use.cols)) max.levels[i]=length(levels(data.df[,i]))
if(any(max.levels==1))stop(paste("All factors must have at least 2 levels.\n
The following variables have only one:",
                     paste(var.names[max.levels==1], collapse=", ")) )

# now re-arrange factors into alphabetic order
# first get name of count variable
count.name = names(data.df)[is.number]
data.df = data.frame(data.df[,is.number],
data.df[, use.cols[order(names(data.df)[!is.number])]])
names(data.df)[1] = count.name
# and counts in reverse lexographic order

dims = get.dim(data.df)
sortvec = rep(1,dim(data.df)[1]); myprod = 1
for(l in 1:length(dims)){
sortvec=(unclass(data.df[,l+1]) - 1)*myprod + sortvec
myprod = myprod * dims[l]
}

# combine duplicated rows
data.df = data.df[order(sortvec),]
sortvec = sort(sortvec)
no.dups = !duplicated(sortvec)
counts = tapply(data.df[,1], sortvec, sum)
data.df = data.df[no.dups, ]
data.df[,1] = counts

list(status=1, data=data.df, message ="Data frame OK")
}


###########################################################################
#
get.dim = function(data.df){

# gets the number of levels of the factors in the data frame data.df
# not required by users

J = dim(data.df)[2] -1  # assumes counts are in column 1
dims=numeric(J)
for(j in 1:J)dims[j] = length(levels(data.df[,j+1]))
dims
}

###########################################################################
```

```
#
is.table.df = function(data.df){
# wrapper for check.data.frame, tests if object is a data
# frame representing a table
check.data.frame(data.df)$status ==1
}

##########################################################################
#
as.standard = function(data.df){

# wrapper for check.data.frame, tests if object is a data frame
# representing a table, returns table in standard form if so
result =  check.data.frame(data.df)
if(result$status!=1)stop(result$message)
result$data
}

#########################################################################
#
table2df= function(table){

# function to convert an array "table" to a data frame
# in standard form
# First check input is an array
if(!is.array(table))stop("Input must be an array")
# then check presence of  dimnames
dimlist=dimnames(table)
if(is.null(dimlist)){
n.var=length(dim(table))
var.names= paste("V", 1:n.var, sep="")
dimlist=vector(n.var, mode="list")
for(i in 1:n.var) dimlist[[i]] = as.character(1:(dim(table)[i]))
names(dimlist)=var.names
}
y=as.vector(table)
if(any(y<0))stop("Table must have non-negative counts")
data.df=data.frame(y, expand.grid(dimlist))
data.df[y!=0,]
}

########################################################################
#
df2table = function(data.df){

# converts data frame to array

# first checks that the data frame is in standard form,
# if not, coerces into standard form, bails out if not possible.
# then forms array after resoring missing zero counts

# first convert input to standardised data frame, quit if not

data.df = as.standard(data.df)


# now restore zero counts
dims = get.dim(data.df)
sortvec = rep(1,dim(data.df)[1]); myprod = 1
for(l in 1:length(dims)){
sortvec=(unclass(data.df[,l+1]) - 1)*myprod + sortvec
myprod = myprod * dims[l]
}
counts = numeric(prod(dims))
counts[sortvec] = data.df[,1]

# get levels for each factor

levels.list = vector(mode="list", length = length(dims))
for ( i in 1:length(dims))levels.list[[i]] = levels(data.df[,i+1])
names(levels.list) = names(data.df)[-1]
array(counts, dims, dimnames=levels.list)
}

##########################################################################
#
n.to.nvec = function(n, base.vec){

# converts the integer n into a mixed-base representation
# bases are in base.vec
# not requireed by users

# eg  if (n-1) = i-1 + (j-1)*I + (k-1)*I*J + (l-1)*I*J*K
# returns (i-1,j-1,k-1,l-1)

Nvar=length(base.vec)
l.index = numeric(Nvar)
cumlevels = (c(1,cumprod(base.vec)[-Nvar]))
nn=n-1
for(index in length(base.vec):1){
l.index[index] = nn%/%cumlevels[index]
nn = nn %% cumlevels[index]
}
l.index
}
```

```
###########################################################################
#
generate.data.sas=function(...){

#  R function to generate artificial data set matching specified margins
#  SAS implementation

#  inputs are data frames containing margins
#  requires fortran shared library



df.list=list(...)
df.list = if(is.list(df.list[[1]]) && length(df.list)==1) df.list[[1]]
else df.list


# inputs: either a single list of df's or each argument is a list
df.list=list(...)

# df.list: a list of data frames in standard form


if((length(df.list)==1)&is.list(df.list[[1]]))  df.list=df.list[[1]]

# now check data frames, and convert to arrays, expanding to include
# zero counts.

array.list = vector(mode="list", length=length(df.list))

for(i in 1:length(df.list))array.list[[i]] = df2table(df.list[[i]])

# get names and dimensions
names.list = vector(mode="list", length=length(df.list))
dim.list = vector(mode="list", length=length(df.list))

vars=NULL; dims = NULL

for(i in 1:length(df.list)) {
names.list[[i]] = names(dimnames(array.list[[i]]))
dim.list[[i]] = sapply(dimnames(array.list[[i]]), length)
vars = c(vars, names.list[[i]])
dims = c(dims, dim.list[[i]])
}

# remove duplicates and sort variable names into order
```

```
dup=duplicated(vars)

var.names=vars[!dup]
max.levels=dims[!dup]

var.order = order(var.names)
var.names = var.names[var.order]
max.levels = max.levels[var.order]

# check for consistency of levels, record factor levels

levels.list = vector(mode="list", length=length(var.names))
for(j in 1:length(var.names)){
  first=TRUE
  name = var.names[j]
  for(i in 1:length(array.list)){
    k = match(name, names(dimnames(array.list[[i]])))
    if( !is.na(k)){
       current.levels = dimnames(array.list[[i]])[[k]]
       if(first) { first.levels = current.levels
   first = FALSE} else
           {
            if(length(first.levels) !=  length(current.levels)){
        stop(paste("Factor levels for factor",name, "not compatible"))}
            if(any(first.levels!= current.levels)){
    stop(paste("Factor levels for factor", name, "not compatible"))}
            }
        }
  }
levels.list[[j]] = first.levels
}

names(levels.list) = var.names
row.names=NULL; effect.names=NULL; b=NULL

for(i in 1:length(df.list)){
result=get.b(names.list[[i]], dim.list[[i]],
                as.vector(array.list[[i]]), dimnames(array.list[[i]]))
row.names = c(row.names,result$row.names)
effect.names = c(effect.names,result$effect.names)
b = c(b,result$b)
}
dup = duplicated(row.names)
row.names = row.names[!dup]
b = b[!dup]

names(b) = row.names
```

```
edup = duplicated(effect.names)
effect.names = effect.names[!edup]

Nvar=length(var.names)
Nterms = length(effect.names)
Nrows = prod(max.levels)

# make a binary matrix representing effect names
# ie 0 1 0 1 represents B:D, 0 1 1 1 B:C:D etc


index.mat = matrix(F,Nterms, Nvar)
for (i in 1: Nterms){
index.mat[i,match(unlist(strsplit(effect.names[i],":")),var.names)] = T
}

# now create constraint matrix A in coded row form
# get dimensions of A

Ncols = prod(max.levels)  # no of columns of A (constraint) matrix

Nrows = 1 + sum(apply(index.mat, 1, function(x)prod((max.levels-1)[x])))
# no of rows of A matrix

max.levels.m1 = max.levels -1
term.offset=numeric(Nterms)
term.offset[1]=1
for(i in 1:(Nterms-1)) term.offset[i+1] = term.offset[i] +
             prod(max.levels.m1[index.mat[i,]])

# get sum of matrix entries nzcnt, this will be the
# number of non-zero entries in A

nzcnt = Ncols  # top row of constraint matrix  is all 1's
for( i in 1:Nterms){
nzcnt = nzcnt + prod(max.levels[!index.mat[i,]])*
             prod(max.levels.m1[index.mat[i,]])
}

# values of non-zero matrix elements are  all ones
# rmatind = numeric(nzcnt)  #  col in which they occur
# rmatbeg = numeric(Nrows)   # starting position in
# rmatind where row i cols occur

# in  the following call, rows (nrow) and columns (ncol) refer
# to the model matrix, the transpose of the constraint matrix.
# Thus nrow = Ncol, ncol = Nrow
```

```
stuff = .Fortran("modmat", indmat= as.integer(index.mat*1),
              maxlev = as.integer(max.levels),
rmatbeg = integer(Nrows), rmatind = integer(nzcnt),
nterms =as.integer(Nterms), nvar=as.integer(Nvar),
nrow=as.integer(Ncols),  ncol=as.integer(Nrows),
nzcnt=as.integer(nzcnt))


# now construct SAS program
outfile = "'x'"
sink("lp.sas")
cat("data test;\n")
cat(paste("infile 'lpdata';\n" ))
cat("input _type_ $ _col_ $ _row_ $ _coef_;\n")
cat("run;\n")
cat("proc lp primalout = x maxit1=10000 maxit2=10000
                 imaxit=10000 time=10000 sparsedata;\n")
cat("run;\n")
cat("data out;\n")
cat("set x;\n")
cat("keep _var_ _value_;\n")
cat("if _type_ eq 'INTEGER';\n")
cat("run;\n")
cat("PROC EXPORT DATA= WORK.OUT OUTFILE= ",outfile,
                                   "DBMS=TAB REPLACE;\n")
cat("run;\n")
sink()

# and data

rmatind = stuff$rmatind
rmatbeg = c(stuff$rmatbeg, nzcnt)

sink("lpdata")
# objective function
for(j in 1:Ncols)cat("max", paste("x",j,sep=""), "obj 1 \n")
# constraints
for(i in 1:Nrows){
   start = rmatbeg[i] + 1; stop = rmatbeg[i+1]
for( j in start:stop)cat("eq", paste("x",rmatind[j]+1,sep=""),
                      paste("r",i,sep=""), 1,"\n")
      cat("rhs _rhs_", paste("r",i,sep=""), b[i],"\n")
}

# upperbounds
for(j in 1:Ncols)cat(paste("upperbd x",j,sep=""), "upper", b[1],"\n")

# variable types
```

```
for(j in 1:Ncols)cat("integer", paste("x",j,sep=""), "int 1 \n")
sink()

# and run it

system("sas lp.sas")
# get x

temp = read.table("x", header=T)
var.order =  order(as.numeric(substring(temp[,1],2)))
x = round(temp[var.order,2])

# clean up, leaving sas log for diagnostic purposes

system("rm x")
system("rm lp.sas")
system("rm lpdata")

table2df(array(x, dim = max.levels, dimnames = levels.list))
}

convert=function(n, base.levels){

# converts the integer n into a mixed-base representation
# bases are in base.vec

# eg  if (n-1) = i-1 + (j-1)*I + (k-1)*I*J + (l-1)*I*J*K
# returns (i-1,j-1,k-1,l-1)

nbase = length(base.levels)
cumlevels = numeric(nbase)
cumlevels[1] = 1
if(nbase>1)for( i in 2:nbase)cumlevels[i]= cumlevels[i-1]*
                                            base.levels[i-1]

l.index=numeric(nbase)
nn=n-1
for(index in nbase:1){
l.index[index] = nn%/%cumlevels[index]
nn = nn - cumlevels[index]*l.index[index]
}
l.index
}

###########################################################################
#
generate.data.cplex=function(..., objective="linear", MAXITER=20,TOL=1.0e-8){
```

```
#  R function to generate artificial data set matching specified margins

# inputs are data frames containing margins

# requires  shared library "SNZ.o" to have been loaded

# If objective="linear", then the IP solution is returned. Otherwise
# the closest quadratic integer approximation to the IPF solution
# is returned



df.list=list(...)
df.list = if(is.list(df.list[[1]]) && length(df.list)==1) df.list[[1]]
else df.list

if (substr(objective,1,1)!="l" && substr(objective,1,1)!="q"){
                  stop("Invalid value for objective")}
objective = if (substr(objective,1,1)=="l" ) "l" else "q"

# inputs: either a single list of df's or each argument is a list
df.list=list(...)

# df.list: a list of data frames in standard form

if((length(df.list)==1)&is.list(df.list[[1]]))  df.list=df.list[[1]]

# now check data frames, and convert to arrays, expanding to
# include zero counts.

array.list = vector(mode="list", length=length(df.list))

for(i in 1:length(df.list))array.list[[i]] = df2table(df.list[[i]])

# get names and dimensions
names.list = vector(mode="list", length=length(df.list))
dim.list = vector(mode="list", length=length(df.list))

vars=NULL; dims = NULL

for(i in 1:length(df.list)) {
names.list[[i]] = names(dimnames(array.list[[i]]))
dim.list[[i]] = sapply(dimnames(array.list[[i]]), length)
vars = c(vars, names.list[[i]])
dims = c(dims, dim.list[[i]])
}
```

```
# remove duplicates and sort variable names into order

dup=duplicated(vars)

var.names=vars[!dup]
max.levels=dims[!dup]

var.order = order(var.names)
var.names = var.names[var.order]
max.levels = max.levels[var.order]

# check for consistency of levels, record factor levels

levels.list = vector(mode="list", length=length(var.names))
for(j in 1:length(var.names)){
  first=TRUE
  name = var.names[j]
  for(i in 1:length(array.list)){
    k = match(name, names(dimnames(array.list[[i]])))
    if( !is.na(k)){
       current.levels = dimnames(array.list[[i]])[[k]]
       if(first) { first.levels = current.levels
   first = FALSE} else
           {
            if(length(first.levels) !=  length(current.levels))
      stop(paste("Factor levels for factor", name, "not compatible"))
            if(any(first.levels!= current.levels))
      stop(paste("Factor levels for factor", name, "not compatible"))
            }
       }
   }

levels.list[[j]] = first.levels
}

names(levels.list) = var.names
row.names=NULL; effect.names=NULL; b=NULL

for(i in 1:length(df.list)){
  result=get.b(names.list[[i]], dim.list[[i]],
       as.vector(array.list[[i]]), dimnames(array.list[[i]]))
  row.names = c(row.names,result$row.names)
  effect.names = c(effect.names,result$effect.names)
  b = c(b,result$b)
}
dup = duplicated(row.names)
row.names = row.names[!dup]
b = b[!dup]
```

```
names(b) = row.names

edup = duplicated(effect.names)
effect.names = effect.names[!edup]

Nvar=length(var.names)
Nterms = length(effect.names)
Nrows = prod(max.levels)

# make a binary matrix representing effect names
# ie 0 1 0 1 represents B:D, 0 1 1 1 B:C:D etc


index.mat = matrix(F,Nterms, Nvar)
for (i in 1: Nterms){
index.mat[i,match(unlist(strsplit(effect.names[i],":")),
      var.names)] = T
}



# now create constraint matrix A in coded row form
# get dimensions of A

Ncols = prod(max.levels) # no of columns of A  matrix

Nrows = 1 + sum(apply(index.mat, 1,
                       function(x)prod((max.levels-1)[x])))
# no of rows of A matrix

max.levels.m1 = max.levels -1
term.offset=numeric(Nterms)
term.offset[1]=1
for(i in 1:(Nterms-1)) term.offset[i+1] = term.offset[i] +
              prod(max.levels.m1[index.mat[i,]])

# get sum of matrix entries nzcnt, this will be the
# number of non-zero entries in A

nzcnt = Ncols  # top row of constraint matrix  is all 1's
for( i in 1:Nterms){
nzcnt = nzcnt + prod(max.levels[!index.mat[i,]])*
             prod(max.levels.m1[index.mat[i,]])
}


# values of non-zero matrix elements are  all ones
# rmatind = numeric(nzcnt)  #  col in which they occur
# rmatbeg = numeric(Nrows)   # starting position in
# rmatind where row i cols occur

# in  the following call, rows (nrow) and columns (ncol) refer
# to the model matrix, the transpose of the constraint matrix.
# Thus rrow = Ncol, ncol = Nrow

Astuff = .Fortran("modmat", indmat= as.integer(index.mat*1),
maxlev = as.integer(max.levels), rmatbeg = integer(Nrows),
rmatind = integer(nzcnt), nterms =as.integer(Nterms),
nvar=as.integer(Nvar), nrow=as.integer(Ncols),
ncol=as.integer(Nrows), nzcnt=as.integer(nzcnt))

if(objective=="l"){

# call cplex
x=.C("docplex", as.integer(Nrows), as.integer(Ncols),
as.integer(nzcnt), as.integer(Astuff$rmatbeg),
as.integer(Astuff$rmatind), as.double(b),
x=double(Ncols))$x

} else {

names(levels.list) = var.names
# make a binary matrix representing effect names
# ie 0 1 0 1 represents B:D, 0 1 1 1 B:C:D etc

N.effect = length(df.list)
Nvar = length(var.names)
N=prod(max.levels)

effect.mat = matrix(F,N.effect, Nvar)
margin = vector(mode="list", length= N.effect)

for (i in 1: N.effect){
effect.mat[i,match(names.list[[i]],var.names)] = T
}

margin.ind = cumsum(c(0, sapply(array.list,length)))[-(N.effect+1)]
margin = unlist(array.list)

# Now do IPF iterations by calling Fortran subroutine

xstar = .Fortran("IPF", effmat = as.integer(effect.mat*1),
neff = as.integer(N.effect),
nvar = as.integer(Nvar),
maxlev = as.integer(max.levels),
marginvec = as.integer(margin),
nmarg = as.integer(length(margin)),
marginind = as.integer(margin.ind),
x = double(N),
n = as.integer(N),
maxiter = as.integer(MAXITER),
tol = as.double(TOL),
crit = double(1))$x

#  call quadratic program

x=.C("quadcplex", as.integer(Nrows), as.integer(Ncols),
as.integer(nzcnt), as.integer(Astuff$rmatbeg),
as.integer(Astuff$rmatind), as.double(b),
as.double(xstar), x=double(Ncols))$x
}

table2df(array(round(x), dim = max.levels, dimnames = levels.list))
}



#################################################################
#
generate.data.ipf=function(..., result = "random", MAXITER=20, TOL=1.0e-8){

# program to generate data using IPF
# requires  shared library "SNZ.so"

# inputs: either a single list of df's or each argument is a list

# check input "result"

if(substr(result,1,2)=="ra") result = "ra" else
if(substr(result,1,2)=="ro") result = "ro" else
stop(" result must be 'random' or 'rounded' ")

df.list=list(...)

# df.list: a list of data frames in standard form

if((length(df.list)==1)&is.list(df.list[[1]]))  df.list=df.list[[1]]

# now check data frames, and convert to arrays, expanding to
# include zero counts.

array.list = vector(mode="list", length=length(df.list))

for(i in 1:length(df.list))array.list[[i]] = df2table(df.list[[i]])
```

```
# get names and dimensions
names.list = vector(mode="list", length=length(df.list))
dim.list = vector(mode="list", length=length(df.list))

vars=NULL; dims = NULL

for(i in 1:length(df.list)) {
names.list[[i]] = names(dimnames(array.list[[i]]))
dim.list[[i]] = sapply(dimnames(array.list[[i]]), length)
vars = c(vars, names.list[[i]])
dims = c(dims, dim.list[[i]])
}

# remove duplicates and sort variable names into order

dup=duplicated(vars)

var.names=vars[!dup]
max.levels=dims[!dup]

var.order = order(var.names)
var.names = var.names[var.order]
max.levels = max.levels[var.order]

# check for consistency of levels, record factor levels

levels.list = vector(mode="list", length=length(var.names))
for(j in 1:length(var.names)){
  first=TRUE
  name = var.names[j]
  for(i in 1:length(array.list)){
    k = match(name, names(dimnames(array.list[[i]])))
    if( !is.na(k)){
       current.levels = dimnames(array.list[[i]])[[k]]
       if(first) { first.levels = current.levels
   first = FALSE} else
           {
            if(length(first.levels) !=  length(current.levels)) {
   stop(paste("Factor levels for factor", name, "not compatible"))}
            if(any(first.levels!= current.levels)) {
   stop(paste("Factor levels for factor", name, "not compatible"))}
            }
       }
   }
levels.list[[j]] = first.levels
}
```

```
names(levels.list) = var.names
# make a binary matrix representing effect names
# ie 0 1 0 1 represents B:D, 0 1 1 1 B:C:D etc

N.effect = length(df.list)
Nvar = length(var.names)
N=prod(max.levels)

effect.mat = matrix(F,N.effect, Nvar)
margin = vector(mode="list", length= N.effect)

for (i in 1: N.effect){
effect.mat[i,match(names.list[[i]],var.names)] = T
}

margin.ind = cumsum(c(0, sapply(array.list,
                               length)))[-(N.effect+1)]
margin = unlist(array.list)

# Now do IPF iterations by calling Fortran subroutine

ipf.list = .Fortran("IPF", effmat = as.integer(effect.mat*1),
neff = as.integer(N.effect),
nvar = as.integer(Nvar),
maxlev = as.integer(max.levels),
marginvec = as.integer(margin),
nmarg = as.integer(length(margin)),
marginind = as.integer(margin.ind),
x = double(N),
n = as.integer(N),
maxiter = as.integer(MAXITER),
tol = as.double(TOL),
crit = double(1))

# now generate a data set
xsum =sum(ipf.list$x)
probs = ipf.list$x/xsum
if(result=="ra"){
list(data =table2df(array( rmultinom(1, xsum, probs),
dim = max.levels, dimnames = levels.list)), probs = probs,
error=ipf.list$crit)
} else
{
list(data =table2df(array( round(ipf.list$x), dim = max.levels,
   dimnames = levels.list)), probs=probs,  error=ipf.list$crit )
}
}
```

```
###########################################################################
#
generate.data.lpSolve=function(...){

#  R function to generate artificial data set matching specified margins

# inputs are data frames containing margins in
# requires R library lpSolve

# inputs: either a single list of df’s or each argument is a df

df.list=list(...)

# df.list: a list of data frames in standard form

if((length(df.list)==1)&is.list(df.list[[1]]))  df.list=df.list[[1]]

# now check data frames, and convert to arrays, expanding to
# include zero counts.

array.list = vector(mode="list", length=length(df.list))

for(i in 1:length(df.list))array.list[[i]] = df2table(df.list[[i]])

# get names and dimensions
names.list = vector(mode="list", length=length(df.list))
dim.list = vector(mode="list", length=length(df.list))

vars=NULL; dims = NULL

for(i in 1:length(df.list)) {
names.list[[i]] = names(dimnames(array.list[[i]]))
dim.list[[i]] = sapply(dimnames(array.list[[i]]), length)
vars = c(vars, names.list[[i]])
dims = c(dims, dim.list[[i]])
}

# remove duplicates and sort variable names into order

dup=duplicated(vars)

var.names=vars[!dup]
max.levels=dims[!dup]

var.order = order(var.names)
var.names = var.names[var.order]
max.levels = max.levels[var.order]
```

```
# check for consistency of levels, record factor levels

levels.list = vector(mode="list", length=length(var.names))
for(j in 1:length(var.names)){
  first=TRUE
  name = var.names[j]
  for(i in 1:length(array.list)){
    k = match(name, names(dimnames(array.list[[i]])))
    if( !is.na(k)){
       current.levels = dimnames(array.list[[i]])[[k]]
       if(first) { first.levels = current.levels
   first = FALSE} else
           {
            if(length(first.levels) !=  length(current.levels)){
    stop(paste("Factor levels for factor",name, "not compatible"))}
            if(any(first.levels!= current.levels)){
    stop(paste("Factor levels for factor", name, "not compatible"))}
            }
       }
   }
levels.list[[j]] = first.levels
}


names(levels.list) = var.names
# first construct formula

complete.formula = ""

for(i in 1:length(df.list)) {
  listvars=names.list[[i]]
  complete.formula=if(i==1){
  paste(complete.formula,paste(listvars, collapse="*"), sep="")} else
  paste( complete.formula,"+", paste(listvars, collapse="*"),sep="")
  }

# get vector of terms in formula

term.vec = unlist(strsplit(complete.formula, "\\+"))

# loop over the effects in the model, getting b's as we go, plus
# expand star terms into main effects, interactions etc

row.names=NULL; b=NULL

for(i in 1:length(term.vec)){
        response = names(df.list[[i]])[1]
formula = formula(paste(response,"~",term.vec[i]))
A=model.matrix(formula, df.list[[i]])
row.names = c(row.names,dimnames(A)[[2]])
b = c(b,t(A)%*%df.list[[i]][,1])
}
dup = duplicated(row.names)
row.names = row.names[!dup]
b = b[!dup]

names(b) = row.names


# now get model matrix for complete model
complete.formula = formula(paste("y~",complete.formula))

# make dummy data frame

dummy.data = data.frame(y=rep(1, prod(max.levels)),
                                 expand.grid(levels.list))
A = t(model.matrix(complete.formula, dummy.data))

# sort individual row labels of A so they match those of b
row.labels = row.names(A)
for( i in 1:length(row.labels)){
split.vec = sort(unlist(strsplit(row.labels[i], "\\:")))
row.labels[i] = paste(split.vec, collapse=":")
}
row.names(A)=row.labels

# rearrange the elements of b to match A
perm= match(row.names(A),names(b))
b=b[perm]

# now solve the IP
library(lpSolve)
ip.obj = lp ("max", objective.in = A[1,], A,
         const.dir=rep("=",dim(A)[1]), b, int.vec=1:dim(A)[2])

# convert back to a data frame

table2df(array( round(ip.obj$solution), dim = max.levels,
         dimnames = levels.list))
}


harmonise = function(..., method="ipf", MAXITER=20, TOL=1.0e-8){

# function to harmonise tables
# wrapper for functions  harmonise.partial, harmonise.ipf, harmonise.all
# which follow

#  inputs:
#   ... :    either a single list of data frames or data frames
#   method:  One of "partial", "complete", "ipf" (first letter suffices)
#   MAXITER: maximum number of iterarions for ipf method
#            (ignored if method is not ipf)
#   TOL:     stopping criterion for ipf method
#            (ignored if method is not ipf)

#  output: list of adjusted data frames

df.list=list(...)

if((length(df.list)==1)&is.list(df.list[[1]]))  df.list=df.list[[1]]



if(substr(method,1,1) =="i") harmonise.ipf(df.list, MAXITER, TOL) else
if(substr(method,1,1) =="c") harmonise.all(df.list) else
if(substr(method,1,1) =="p") harmonise.partial(df.list) else
stop("CPLEX not avaliable, Method must be ipf")
}

############################################################################
#
harmonise.ipf = function(df.list, MAXITER=20, TOL=1.0e-8){

# inputs: df.list, a list of data frames

# first check data frames, and convert to arrays, expanding
# to include zero counts.

array.list = vector(mode="list", length=length(df.list))

for(i in 1:length(df.list))array.list[[i]] = df2table(df.list[[i]])

# get names and dimensions
names.list = vector(mode="list", length=length(df.list))
dim.list = vector(mode="list", length=length(df.list))

vars=NULL; dims = NULL

for(i in 1:length(df.list)) {
        names.list[[i]] = names(dimnames(array.list[[i]]))
        dim.list[[i]] = sapply(dimnames(array.list[[i]]), length)
        vars = c(vars, names.list[[i]])
        dims = c(dims, dim.list[[i]])
        }

# remove duplicates and sort variable names into order

dup=duplicated(vars)

var.names=vars[!dup]
max.levels=dims[!dup]

var.order = order(var.names)
var.names = var.names[var.order]
max.levels = max.levels[var.order]

# check for consistency of levels, record factor levels

levels.list = vector(mode="list", length=length(var.names))
for(j in 1:length(var.names)){
  first=TRUE
  name = var.names[j]
  for(i in 1:length(array.list)){
    k = match(name, names(dimnames(array.list[[i]])))
    if( !is.na(k)){
       current.levels = dimnames(array.list[[i]])[[k]]
       if(first) { first.levels = current.levels
           first = FALSE} else
           {
            if(length(first.levels) !=  length(current.levels)){
          stop(paste("Factor levels for factor", name, "not compatible"))}
            if(any(first.levels!= current.levels)){
          stop(paste("Factor levels for factor", name, "not compatible"))}
            }
      }
   }
levels.list[[j]] = first.levels
}

names(levels.list) = var.names
# make a binary matrix representing effect names
# ie 0 1 0 1 represents B:D, 0 1 1 1 B:C:D etc

N.effect = length(df.list)
Nvar = length(var.names)
N=prod(max.levels)

effect.mat = matrix(F,N.effect, Nvar)
margin = vector(mode="list", length= N.effect)
```

```
for (i in 1: N.effect){
effect.mat[i,match(names.list[[i]],var.names)] = T
}

margin.ind = cumsum(c(0, sapply(array.list,length)))[-(N.effect+1)]
margin = unlist(array.list)

# Now do IPF iterations by calling Fortran subroutine

x = .Fortran("IPF", effmat = as.integer(effect.mat*1),
neff = as.integer(N.effect),
nvar = as.integer(Nvar),
maxlev = as.integer(max.levels),
marginvec = as.integer(margin),
nmarg = as.integer(length(margin)),
marginind = as.integer(margin.ind),
x = double(N),
n = as.integer(N),
maxiter = as.integer(MAXITER),
tol = as.double(TOL),
crit = double(1))$x

# now generate harmonised margins
X = array(round(x), dim = max.levels, dimnames = levels.list)
for(i in 1:length(df.list)){
X.temp = apply(X, match(names.list[[i]], var.names), sum)
dimnames(X.temp) = dimnames(array.list[[i]])
df.list[[i]] = table2df(X.temp)
}
df.list
}

###########################################################################
#
harmonise.partial = function(df.list){

#  inputs: either a single list of data frames or each argument
#  is a data frame
#  output: list of adjusted data frames

array.list = vector(mode="list", length=length(df.list))

for(i in 1:length(df.list))array.list[[i]] = df2table(df.list[[i]])

# get names and dimensions
names.list = vector(mode="list", length=length(df.list))
dim.list = vector(mode="list", length=length(df.list))
```

```
vars=NULL; dims = NULL

for(i in 1:length(df.list)) {
names.list[[i]] = names(dimnames(array.list[[i]]))
dim.list[[i]] = sapply(dimnames(array.list[[i]]), length)
vars = c(vars, names.list[[i]])
dims = c(dims, dim.list[[i]])
}

# remove duplicates and sort variable names into order

dup=duplicated(vars)

var.names=vars[!dup]
max.levels=dims[!dup]

var.order = order(var.names)
var.names = var.names[var.order]
max.levels = max.levels[var.order]

# get marginal tables

b = NULL
for(i in 1:length(df.list)) b = c(b, as.vector(array.list[[i]]))

#  get length of solution vector

x.lengths = sapply(dim.list, prod)
x.start = c(0,cumsum(x.lengths))

rmatind = NULL
rmatbeg = 0
rmatval = NULL

# loop over pairs of tables to get intersections
for( i in 1:(length(df.list)-1)){
  for(j in  (i+1):length(df.list)){
    Si=match(names.list[[i]], var.names)
    Sj=match(names.list[[j]], var.names)
# if Si and Sj intersect form  the contrast matrices Aij and
# Aji in row form
    if(length(intersect(Si,Sj))>0){
      result = get.constraint.matrices(Si, Sj, max.levels)
      off1 = x.start[i]
      off2 = x.start[j]
      lmb=length(result$rmatbeg1)
      for(k in 1:lmb){
        index1 = if (k<lmb) (result$rmatbeg1[k] + 1):
                 result$rmatbeg1[k+1] else
                    (result$rmatbeg1[k]+1): length(result$rmatind1)
        index2 = if (k<lmb) (result$rmatbeg2[k] + 1):
                 result$rmatbeg2[k+1] else
                    (result$rmatbeg2[k]+1): length(result$rmatind2)
        rmatind= c(rmatind, result$rmatind1[index1]+off1,
                    result$rmatind2[index2]+off2)
        rmatval = c(rmatval, rep(1, length(index1)),
                    rep(-1, length(index2)))
        rmatbeg = c(rmatbeg, length(index1) + length(index2))
        }
     }
   }
}

# check for no intersecting tables

if(is.null(rmatind)) return(df.list)

# otherwise adjust tables


newb = .C("h2cplex", as.integer(length(rmatbeg)),
as.integer(length(b)), as.integer(length(rmatind)),
as.integer(cumsum(rmatbeg)),  as.integer(rmatind),
as.double(rmatval), as.double(b), x=double(length(b)))$x

new.list = vector(mode="list", length=length(df.list))
start=0
stop = 0
for(i in 1: length(df.list)){
start=stop + 1
stop = stop + dim(df.list[[i]])[1]
df.list[[i]][,1] = newb[start:stop]
new.list[[i]] = df.list[[i]]
}
new.list
}

############################################################################
#
harmonise.all=function(df.list){

# R function to harmonise tables, using quadratic method
# called by function harmonise
# input: df.list, a list of data frames
# requires fortran and cplex shared libraries
# first check data frames, and convert to arrays, expanding to
# include zero counts.

array.list = vector(mode="list", length=length(df.list))

for(i in 1:length(df.list)){
  array.list[[i]] = df2table(df.list[[i]])
  }

# get names and dimensions
names.list = vector(mode="list", length=length(df.list))
dim.list = vector(mode="list", length=length(df.list))

vars=NULL; dims = NULL

for(i in 1:length(df.list)) {
  names.list[[i]] = names(dimnames(array.list[[i]]))
  dim.list[[i]] = sapply(dimnames(array.list[[i]]), length)
  vars = c(vars, names.list[[i]])
  dims = c(dims, dim.list[[i]])
  }

# remove duplicates and sort variable names into order

dup=duplicated(vars)

var.names=vars[!dup]
max.levels=dims[!dup]

var.order = order(var.names)
var.names = var.names[var.order]
max.levels = max.levels[var.order]

# check for consistency of levels, record factor levels

levels.list = vector(mode="list", length=length(var.names))
for(j in 1:length(var.names)){
  first=TRUE
  name = var.names[j]
  for(i in 1:length(array.list)){
    k = match(name, names(dimnames(array.list[[i]])))
    if( !is.na(k)){
      current.levels = dimnames(array.list[[i]])[[k]]
      if(first) { first.levels = current.levels
first = FALSE} else
         {
         if(length(first.levels) !=  length(current.levels)){
```

```
    stop(paste("Factor levels for factor",name, "not compatible"))}
         if(any(first.levels!= current.levels)){
    stop(paste("Factor levels for factor", name, "not compatible"))}
          }
       }
   }
levels.list[[j]] = first.levels
}

names(levels.list) = var.names
row.names=NULL; effect.names=NULL; b=NULL

for(i in 1:length(df.list)){
  result=get.b(names.list[[i]], dim.list[[i]],
     as.vector(array.list[[i]]), dimnames(array.list[[i]]))
  row.names = c(row.names,result$row.names)
  effect.names = c(effect.names,result$effect.names)
  b = c(b,result$b)
}
dup = duplicated(row.names)
row.names = row.names[!dup]
b = b[!dup]

names(b) = row.names

edup = duplicated(effect.names)
effect.names = effect.names[!edup]

Nvar=length(var.names)
Nterms = length(effect.names)
Nrows = prod(max.levels)

# make a binary matrix representing effect names
# ie 0 1 0 1 represents B:D, 0 1 1 1 B:C:D etc


index.mat = matrix(F,Nterms, Nvar)
for (i in 1: Nterms){
  index.mat[i,match(unlist(strsplit(effect.names[i],":")),
       var.names)] = T
}

# now create constraint matrix A in coded row form
# get dimensions of A

Ncols = prod(max.levels)  # no of columns of A  matrix

Nrows = 1 + sum(apply(index.mat, 1,
          function(x)prod((max.levels-1)[x])))
# no of rows of A matrix

max.levels.m1 = max.levels -1
term.offset=numeric(Nterms)
term.offset[1]=1
for(i in 1:(Nterms-1)) term.offset[i+1] = term.offset[i] +
              prod(max.levels.m1[index.mat[i,]])

# get sum of matrix entries nzcnt, this will be the
# number of non-zero entries in A

nzcnt = Ncols  # top row of constraint matrix  is all 1's
for( i in 1:Nterms){
nzcnt = nzcnt + prod(max.levels[!index.mat[i,]])*
              \prod(max.levels.m1[index.mat[i,]])
}



# values of non-zero matrix elements are  all ones
# rmatind = numeric(nzcnt)  #  col in which they occur
# rmatbeg = numeric(Nrows)   # starting position in
# rmatind where row i cols occur

# in  the following call, rows (nrow) and columns (ncol)
# refer to the model matrix, the transpose of the constraint matrix.
# Thus rrow = Ncol, ncol = Nrow

Astuff = .Fortran("modmat", indmat= as.integer(index.mat*1),
maxlev = as.integer(max.levels), rmatbeg = integer(Nrows),
rmatind = integer(nzcnt), nterms =as.integer(Nterms),
nvar=as.integer(Nvar), nrow=as.integer(Ncols),
ncol=as.integer(Nrows), nzcnt=as.integer(nzcnt))

# now adjust rmatbeg etc

rmatind = Astuff$rmatind
rmatval = rep(1, length(rmatind))
rmatbeg = c(Astuff$rmatbeg, length(rmatind))

insert = function(x, val,pos)if(pos==0)c(val,x) else
             if(pos==length(x)) c(x,val) else
             c(x[1:pos], val, x[(pos+1):length(x)])
for( i in 1:Nrows){
rmatbeg[i+1] = rmatbeg[i+1]+i
rmatind = insert(rmatind, Ncols + i-1, rmatbeg[i+1]-1)
rmatval = insert(rmatval, -1, rmatbeg[i+1]-1)
}
```

```
rmatbeg = rmatbeg[-length(rmatbeg)]
newNcols = Ncols + Nrows
nzcnt = nzcnt + Nrows
zstar = c(rep(0,Ncols),b)

# call cplex

z = .C("h1cplex", as.integer(Nrows), as.integer(Ncols),
as.integer(newNcols), as.integer(nzcnt), as.integer(rmatbeg),
as.integer(rmatind), as.double(rmatval),
as.double(zstar), z=double(newNcols))$z

# now produce list of harmonised tables

X = array(z[1:Ncols], max.levels)
for(i in 1:length(df.list)){
X.temp = apply(X, match(names.list[[i]], var.names), sum)
dimnames(X.temp) = dimnames(array.list[[i]])
df.list[[i]] = table2df(X.temp)
}

df.list
}



############################################################################
#
 get.b = function(namevec, dimvec,y, level.list){

# namevec: vector of variable names for table margin
# dimvec: table dimensions
# y: vector of counts in standard form
#     (as.vector applied to array)
# level.list: levels for factors in current term
#     (dimnames for current array)
# returns list with margins in b, names in row.names
# used in generate.data.cplex, generate.data.sas

Nvars = length(namevec)
Nterms = 2^Nvars-1

#  make index.mat
index.mat = matrix(F, Nterms, Nvars)

for(i in 1:Nterms) index.mat[i,] = as.logical(n.to.nvec(i+1,
                  rep(2,Nvars)))
```

```
# make rownames for the margin b

rownames=NULL
effectnames=NULL
for(i in 1:Nterms){
tempnames1 = ""
tempnames2 = ""
use = (1:Nvars)[index.mat[i,]]
for(k in 1:length(use)){
fac2 = paste(namevec[use[k]],level.list[[use[k]]][-1], sep="")
fac3 = namevec[use[k]]
tempnames1 = outer(tempnames1, fac2,
                 function(x,y)ifelse(x=="", y, paste(x,y,sep=":")))
tempnames2 = outer(tempnames2, fac3,
                 function(x,y)ifelse(x=="", y, paste(x,y,sep=":")))
}
rownames=c(rownames, tempnames1)
effectnames=c(effectnames, tempnames2)
}
rownames=c("Intercept", rownames)


# now create constraint matrix A in coded row form
# get dimensions of A

Ncols = prod(dimvec)  # no of columns of A  matrix
dimvec.m1 = dimvec -1

# no of rows of A matrix
Nrows = 1 + sum(apply(index.mat, 1,
            function(x)prod((dimvec.m1)[x])))


term.offset=numeric(Nterms)
term.offset[1]=1
for(i in 1:(Nterms-1)) term.offset[i+1] = term.offset[i] +
              prod(dimvec.m1[index.mat[i,]])

# get sum of matrix entries nzcnt, this will be the number
# of non-zero entries in A

nzcnt = Ncols  # top row of constraint matrix  is all 1's
for( i in 1:Nterms){
nzcnt = nzcnt + prod(dimvec[!index.mat[i,]])*
             prod(dimvec.m1[index.mat[i,]])
}
```

```
#initialize rmatind
rmatind = numeric(nzcnt)


# in  the following call, rows (nrow) and columns (ncol) refer to
# the model matrix, the transpose of the constraint matrix.
# Thus rrow = Ncol, ncol = Nrow

stuff = .Fortran("modmat", indmat= as.integer(index.mat*1),
maxlev = as.integer(dimvec), rmatbeg = integer(Ncols),
rmatind = integer(nzcnt), nterms =as.integer(Nterms),
nvar=as.integer(Nvars), nrow=as.integer(Ncols),
ncol=as.integer(Nrows), nzcnt=as.integer(nzcnt))

# calculate b

rmatbeg = c(stuff$rmatbeg, nzcnt)

b=numeric(Nrows)
for( i in 1:Nrows){
start = rmatbeg[i] + 1
stop = rmatbeg[i+1]
b[i] = sum(y[stuff$rmatind[start:stop]+1])
}

list(b=b, row.names = rownames, effect.names=effectnames)

}

############################################################################

get.constraint.matrices=function(S1, S2, factor.levels){

# function to extract Aij and Aji in row form, given
# margin sets and factor levels

# S1, S2:  two subsets corresponding to two different
# marginal tables e.g S1={1,3} corresponds to the A1A3 table
# max.levels: vector containing the number of levels in each
# factor outputs model matrices in coded form in vectors
# rmatbeg1 and rmatind1 and rmatbeg2 and rmatind2



# make a binary matrix representing effect names for a
# saturated model; rows are binary reps of 1:(k-1) where
# k is number of elements in  intersect(S1,S2)
```

```
S=intersect(S1,S2)
k=length(S)
k1=length(S1)
k2=length(S2)

if(k>0){
bin.mat = matrix(0,2^k-1, k)
for (i in 1: (2^k-1)){
bin.mat[i,] = convert(i+1, rep(2,k))
}
}

# first matrix

max.levels = factor.levels[S1]

index.mat = matrix(0, (2^k-1), k1)
index.mat[,match(S,S1)] = bin.mat
Ncols = 1+sum(apply(index.mat,1,function(x,max.levels){
      prod((max.levels-1)^x)},max.levels))

Nrows=prod(max.levels)
Nterms = dim(index.mat)[1]
Nvar = dim(index.mat)[2]

# Initialize rmatbeg
rmatbeg = numeric(Ncols)
rmatbeg[1]=0


# get sum of matrix entries

max.levels.m1 = max.levels -1
matsum=prod(max.levels)
for( i in 1:Nterms){
matsum = matsum + prod(max.levels[index.mat[i,]!=1])*
             prod(max.levels.m1[index.mat[i,]==1])
}

#initialize rmatind

rmatind = numeric(matsum)

# now call .Fortran function

stuff = .Fortran("modmat", indmat= as.integer(index.mat*1),
maxlev = as.integer(max.levels), rmatbeg = integer(Ncols),
rmatind = integer(matsum), nterms =as.integer(Nterms),
nvar=as.integer(Nvar), nrow=as.integer(Nrows),
ncol=as.integer(Ncols), matsum=as.integer(matsum))
rmatbeg1=stuff$rmatbeg
rmatind1=stuff$rmatind

# now for second matrix

max.levels = factor.levels[S2]

index.mat = matrix(0, (2^k-1), k2)
index.mat[,match(S,S2)] = bin.mat
Ncols = 1+sum(apply(index.mat,1,function(x,max.levels){
       prod((max.levels-1)^x)},max.levels))

Nrows=prod(max.levels)
Nterms = dim(index.mat)[1]
Nvar = dim(index.mat)[2]

# Initialize rmatbeg
rmatbeg = numeric(Ncols)
rmatbeg[1]=0

# get sum of matrix entries

max.levels.m1 = max.levels -1
matsum=prod(max.levels)
for( i in 1:Nterms){
matsum = matsum + prod(max.levels[index.mat[i,]!=1])*
               prod(max.levels.m1[index.mat[i,]==1])
}

#initialize rmatind
rmatind = numeric(matsum)

# now call .Fortran function

stuff = .Fortran("modmat", indmat= as.integer(index.mat*1),
maxlev = as.integer(max.levels), rmatbeg = integer(Ncols),
rmatind = integer(matsum), nterms =as.integer(Nterms),
nvar=as.integer(Nvar), nrow=as.integer(Nrows),
ncol=as.integer(Ncols), matsum=as.integer(matsum))

list(rmatbeg1=rmatbeg1, rmatind1=rmatind1, rmatbeg2=stuff$rmatbeg,
rmatind2=stuff$rmatind)
}
```

A.4 Fortran and C code listings

Finally, we give the listings of the C and Fortran code used to construct the shared libraries.

In file Rcplex.c:

```
/* Includes and definitions */


#include <ilcplex/cplex.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define ROWSPACE    50000
#define COLSPACE    100000
#define NZSPACE     500000

 /* cplex  program for first quadratic harmonisation program*/

void  h1cplex(int *numrows, int *numcols, int *newnumcols, int *nzcnt,
  int *rmatbeg, int *rmatind, double *rmatval, double *zstar, double *z){

int      numcols1;  /*  numcols-1 */

   /*  Declare and allocate space for the variables and arrays where we will
   store the optimization results including the status, objective value,
   variable values, dual values, row slacks and variable reduced costs. */

       char     sense[ROWSPACE];
       double   rhs[ROWSPACE];
       double   value[COLSPACE];
       char     ctype[COLSPACE];
       int      index[COLSPACE];

       CPXENVptr     env = NULL;
       CPXLPptr      lp = NULL;
       int           status;
       int           i;
       int           newnumcols1;
       char          errmsg[1024];
       char          *filename = "harm1.lp";
       char          *probname = "harm1";

 /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);
```

```
/* If an error occurs, the status value indicates the reason for
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no output,
   so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.
   but we do not do this  */

   if ( env == NULL ) {
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }

/*     Load the problem. */

      lp = CPXcreateprob(env, &status, probname);
      if (lp == NULL){
         printf("Failed to load LP");
         goto TERMINATE;
      }

/*   set RHS equal to zero*/

      for( i=0; i<*numrows; i++) rhs[i] = 0.0;

/*    get vector sens, length NUMMROWS */

      for( i=0; i<*numrows; i++) sense[i] = 'E';

/*    now add rows to lp */

  status = CPXaddrows(env, lp, *newnumcols, *numrows,
         *nzcnt, rhs, sense, rmatbeg, rmatind, rmatval, NULL, NULL);

        if(status!=0)goto TERMINATE;

/*  add coefficients to objective function */

for( i=0; i<*newnumcols; i++){
 index[i] = i;
         value[i] = -zstar[i];
 ctype[i]='I';
 }
status = CPXchgobj(env, lp,  *newnumcols, index, value);
        if(status!=0)goto TERMINATE;
```

```
/*  now add quadratic terms  (reuse vector value to save space)*/

       for( i=0; i<*numcols; i++){
         value[i] = 0.0;
 }
   for( i=0; i<*numrows; i++){
         value[i+*numcols] = 1.0;
 }


       status = CPXcopyqpsep(env,lp,value);
       if(status!=0)goto TERMINATE;

/*  and change their type */

       status = CPXchgctype(env,lp, *numcols, index, ctype);
       if(status!=0)goto TERMINATE;

/* write out problem specification*/

       status = CPXlpwrite(env, lp, filename);
       if(status!=0)goto TERMINATE;

printf("mip saved\n");

/* do optimisation */

status = CPXmipopt(env, lp);
if(status!=0)goto TERMINATE;

/* printf("mipopt done\n");*/

newnumcols1 = *newnumcols - 1;
status = CPXgetx(env,lp,z, 0, newnumcols1);

if(status!=0)goto TERMINATE;
        printf("found x\n ");

    TERMINATE:

         if (lp != NULL){
             status = CPXfreeprob(env, &lp);
             if(status!= 0) printf( "CPXfreeprob failed, error code %d", status);
          }

/*      Free up the CPLEX environment, if necessary*/

      if(env !=NULL){
         status = CPXcloseCPLEX(&env);

         if(status != 0){
            printf("Could not close CPLEX environment");
            CPXgeterrorstring(env ,status, errmsg);
            printf("%s",errmsg);
         }
}

 }

/* #####################################################################*/

/* cplex  function h2cplex  for second quadratic harmonisation program*/

void  h2cplex(int *numrows, int *numcols, int *nzcnt, int *rmatbeg,
            int *rmatind, double *rmatval, double *b, double *x){

   int      numcols1;  /*  numcols-1 */


/*  Declare and allocate space for the variables and arrays where we will
   store the optimization variables. */

      char     sense[ROWSPACE];
      double   rhs[ROWSPACE];
      double   value[COLSPACE];
      char     ctype[COLSPACE];
      int      index[COLSPACE];

       CPXENVptr     env = NULL;
       CPXLPptr      lp = NULL;
       int           status;
       int           i;
       char          errmsg[1024];
       char          *filename = "h2cplex.lp";
       char          *probname = "h2cplex";

 /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

      if ( env == NULL ) {
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }
```

```
/*     Load the problem. */

        lp = CPXcreateprob(env, &status, probname);
        if (lp == NULL){
         printf("Failed to load LP");
         goto TERMINATE;
      }

/*   set RHS equal to zero*/

      for( i=0; i<*numrows; i++) rhs[i] = 0.0;

/*    get vector sens, length NUMMROWS */

      for( i=0; i<*numrows; i++) sense[i] = 'E';


/*    now add rows to lp */

  status = CPXaddrows(env, lp, *numcols, *numrows,
         *nzcnt, rhs, sense, rmatbeg, rmatind, rmatval, NULL, NULL);


      if(status!=0)goto TERMINATE;

 /*  add coefficients to objective function */

for( i=0; i<*numcols; i++){
 index[i] = i;
         value[i] = -b[i];
 ctype[i]='I';
 }
status = CPXchgobj(env, lp,  *numcols, index, value);
        if(status!=0)goto TERMINATE;



/*  now add quadratic terms  (reuse vector value to save space)*/

       for( i=0; i<*numcols; i++){
         value[i] = 1.0;
 }
        status = CPXcopyqpsep(env,lp,value);
        if(status!=0)goto TERMINATE;

/*  and change their type */
```

```
       status = CPXchgctype(env,lp, *numcols, index, ctype);
       if(status!=0)goto TERMINATE;

/* write out problem specification*/

       status = CPXlpwrite(env, lp, filename);
       if(status!=0)goto TERMINATE;

printf("mip saved\n");

/* do optimisation */

status = CPXmipopt(env, lp);
if(status!=0)goto TERMINATE;

/* printf("mipopt done\n");*/

numcols1 = *numcols - 1;
status = CPXgetx(env,lp,x, 0,numcols1);

if(status!=0)goto TERMINATE;
        printf("found x\n ");

TERMINATE:

         if (lp != NULL){
             status = CPXfreeprob(env, &lp);
             if(status!= 0) printf( "CPXfreeprob failed, error code %d", status);
          }

/*      Free up the CPLEX environment, if necessary*/

      if(env !=NULL){
         status = CPXcloseCPLEX(&env);


         if(status != 0){
            printf("Could not close CPLEX environment");
            CPXgeterrorstring(env ,status, errmsg);
            printf("%s",errmsg);
         }
}

 }
/* ###########################################################################
/* cplex function for linear objective function  */

 void  docplex(int *numrows, int *numcols, int *nzcnt, int *rmatbeg,
              int *rmatind, double *rhs, double *x){

   int      numcols1;  /*  numcols-1 */

   /*  Declare and allocate space for the variables and arrays where we will
   store the optimization variables */

      double   obj[COLSPACE];
      char     sense[ROWSPACE];
      double   rmatval[NZSPACE];
      double   value[COLSPACE];
      char     ctype[COLSPACE];
      int      index[COLSPACE];


       CPXENVptr     env = NULL;
       CPXLPptr      lp = NULL;
       int           status;
       int           i;
       char          errmsg[1024];
       char          *filename = "linear1.lp";
       char          *probname = "linear1";

      /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }

/*    status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
    if (status) {
        fprintf (stderr, "Failure to turn on screen indicator, error %d.\n", status);
        goto TERMINATE;
    }

*/
```

```
/*     Load the problem. */

        lp = CPXcreateprob(env, &status, probname);



      if (lp == NULL){
         printf("Failed to load LP");
         goto TERMINATE;
      }






/*    get vector sens, length NUMMROWS */

      for( i=0; i<*numrows; i++) sense[i] = 'E';


/*   make vector rmatval, all values 1, length nzcnt */

for( i=0; i<*nzcnt; i++) rmatval[i] = 1.0;


/*    now add rows to lp */


status = CPXaddrows(env, lp, *numcols, *numrows,
         *nzcnt, rhs, sense, rmatbeg, rmatind, rmatval, NULL, NULL);


        if(status!=0)goto TERMINATE;

/*  add coefficients to objective function */

for( i=0; i<*numcols; i++){
 index[i] = i;
         value[i] = 1.0;
 ctype[i]='I';
 }
status = CPXchgobj(env, lp,  *numcols, index, value);
        if(status!=0)goto TERMINATE;



/*  and change their type */
```

```
       status = CPXchgctype(env,lp, *numcols, index, ctype);
       if(status!=0)goto TERMINATE;

/* write out problem specification*/


       status = CPXlpwrite(env, lp, filename);
       if(status!=0)goto TERMINATE;

printf("mip saved\n");

/* do optimisation */

status = CPXmipopt(env, lp);
if(status!=0)goto TERMINATE;

/* printf("mipopt done\n");*/

numcols1 = *numcols - 1;
status = CPXgetx(env,lp,x, 0,numcols1);
if(status!=0)goto TERMINATE;
        printf("found x\n ");

TERMINATE:


         if (lp != NULL){
             status = CPXfreeprob(env, &lp);
             if(status!= 0) printf( "CPXfreeprob failed, error code %d", status);
          }

/*      Free up the CPLEX environment, if necessary*/

      if(env !=NULL){
         status = CPXcloseCPLEX(&env);

/*      Note that CPXcloseCPLEX produces no output,
        so the only way to see the cause of the error is to use
        CPXgeterrorstring.  For other CPLEX routines, the errors will
        be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.
*/

         if(status != 0){
            printf("Could not close CPLEX environment");
            CPXgeterrorstring(env ,status, errmsg);
            printf("%s",errmsg);
         }
}

 }

/* ####################################################################*/
/* cplex function used to find integer solution closest to ipf solution */

void  quadcplex(int *numrows, int *numcols, int *nzcnt, int *rmatbeg,
     int *rmatind, double *rhs,  double *xstar, double *x){

   int      numcols1;  /*  numcols-1 */

/*  Declare and allocate space for the variables and arrays where we will
   store the optimization variables. */

      double   obj[COLSPACE];
      char     sense[ROWSPACE];
      double   rmatval[NZSPACE];
      double   value[COLSPACE];
      char     ctype[COLSPACE];
      int      index[COLSPACE];

       CPXENVptr     env = NULL;
       CPXLPptr      lp = NULL;
       int           status;
       int           i;
       char          errmsg[1024];
       char          *filename = "quad.lp";
       char          *probname = "quad";


       /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);
   if ( env == NULL ) {
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }

/*     Load the problem. */

        lp = CPXcreateprob(env, &status, probname);
        if (lp == NULL){
        printf("Failed to load LP");
        goto TERMINATE;
      }
```

```
/*    get vector sens, length NUMMROWS */

      for( i=0; i<*numrows; i++) sense[i] = 'E';

/*   make vector rmatval, all values 1, length nzcnt */

for( i=0; i<*nzcnt; i++) rmatval[i] = 1.0;

/*    now add rows to lp */

status = CPXaddrows(env, lp, *numcols, *numrows,
         *nzcnt, rhs, sense, rmatbeg, rmatind, rmatval, NULL, NULL);
        if(status!=0)goto TERMINATE;

/*  add coefficients to objective function */

for( i=0; i<*numcols; i++){
 index[i] = i;
         value[i] = -xstar[i];
 ctype[i]='I';
 }
status = CPXchgobj(env, lp,  *numcols, index, value);
        if(status!=0)goto TERMINATE;

/*  now add quadratic terms  (reuse vector value to save space) */

for( i=0; i<*numcols; i++){
         value[i] = 1.0;
 }

          status = CPXcopyqpsep(env,lp,value);
          if(status!=0)goto TERMINATE;

/*  and change their type */

       status = CPXchgctype(env,lp, *numcols, index, ctype);
       if(status!=0)goto TERMINATE;

/* write out problem specification*/

       status = CPXlpwrite(env, lp, filename);
       if(status!=0)goto TERMINATE;
```

```
printf("mip saved\n");

        status = CPXchgctype(env, lp, *numcols, index, ctype);
                if(status!=0)goto TERMINATE;

        status = CPXlpwrite(env, lp, filename);
                if(status!=0)goto TERMINATE;

        printf("program saved\n");

 /* do optimisation */

status = CPXmipopt(env, lp);
if(status!=0)goto TERMINATE;

/* get solution */

numcols1 = *numcols - 1;
status = CPXgetx(env,lp,x, 0,numcols1);

     printf("found x\n ");

TERMINATE:

        if (lp != NULL){
             status = CPXfreeprob(env, &lp);
             if(status!= 0) printf( "CPXfreeprob failed, error code %d", status);
          }

/*      Free up the CPLEX environment, if necessary*/

      if(env !=NULL){
         status = CPXcloseCPLEX(&env);

/* close CPLEX */

         if(status != 0){
            printf("Could not close CPLEX environment");
            CPXgeterrorstring(env ,status, errmsg);
            printf("%s",errmsg);
         }
}
}
```

In file SNZ.f

```
C    Subroutine for iterated proportional fitting

        subroutine IPF(effmat, neff, nvar, maxlev, marginvec,
     ;  nmarg, marginind, x, n, maxiter, tol, crit)
C
C    effmat: the matrix of effects, coded as for vector use above
C    neff: the number of effects (margins), this is the number of
C          rows of effmat
C    nvar: the number of columns of effmat, equal to the number
C          of variables in the full table
C    maxlev: the vector containing the number of levels for each factor
C    marginvec: a vector of length nmarg containing the margins
C    nmarg: length of marginvec
C    marginind: the offsets for the individual margins
C    x:  the computed table of fitted means
C    n:  length of x
C    maxiter: max number of iterations for the IPF algorithm
C    tol: stopping criterion


        integer effmat(neff,*), maxlev(nvar), marginvec(nmarg)
integer marginind(neff), iter
double precision x(n), tol, crit

C local variables
        integer i, j, jj, k,  nprod, use(nvar), curoff
double precision xx(n), change, newcrit
integer lindex(nvar), ivec(nvar)
integer istar, bvec(nvar), index

    do 1 i=1,n
        x(i) = 1.0D0
1       continue

        do 2 iter = 1, maxiter
    crit = 0.0D0
    do 3 i = 1, neff
        nprod = 1
do 4  j = 1, nvar
    nprod = nprod * maxlev(j)**effmat(i,j)
    use(j) = effmat(i,j)
4               continue

C  nprod is number of elements in current configuration

               call getconfig(x, n, xx, nprod, maxlev, use, nvar)
```

```
       curoff = marginind(i)
       do 5 k = 1, nprod
           check = DABS(xx(k))
           if(check. GT. 1.0D-8) then
    xx(k) = marginvec(k+curoff)/xx(k)
   else
        xx(k) = 0.0D0
   endif
5              continue

               do 7 j=1,n
           call convert(j, maxlev, nvar, lindex)
           index = 0
           do 10 jj = 1, nvar
           if(use(jj).eq. 0) goto 10
      index = index + 1
      ivec(index) = lindex(jj)
      bvec(index) = maxlev(jj)
10                 continue
       call nvec2j(ivec, bvec, index, istar)

       change = (xx(istar) -1)*x(j)

       newcrit = DABS(change)
       if(newcrit.gt.crit) then
       crit = newcrit
       endif
       x(j) = change + x(j)
7              continue
3           continue

C  check stoping criterion

    if(crit.lt.tol)goto 9999
2       continue
9999    return
    end

        subroutine nvec2j(ivec, bvec, n, j)
C
C   converts a mixed base representation to an integer
C  i.e. if j-1 = (i1-1) + (i2-1)*I1 + (i3-1)*I1*I2 + ...
C  converts the vector ivec = (i1-1,i2-1,...) of length n into j
C  the mixed bases I1, I2, are in vector bvec of dimension n
C  only the first n-1 elements of bvec are used
```

```
      integer ivec(n), bvec(n)

      integer cumvec(n),i
C  cumvec contains the cumulative products 1, I1, I1*I2

   cumvec(1) = 1
      do 1 i=2,n
     cumvec(i) = cumvec(i-1)*bvec(i-1)
1     continue
      j=1
   do 2 i=1,n
      j = j + ivec(i)*cumvec(i)
2     continue
      return
      end

      subroutine convert(n, bselev,  nbase, lindex)

C        converts the integer n into a mixed-base representation in lindex
C        bases are in bselev, of length nbase

C        eg  if (n-1) = i-1 + (j-1)*I + (k-1)*I*J + (l-1)*I*J*K
C        returns (i-1,j-1,k-1,l-1)


       integer bselev(nbase),lindex(nbase)

C      local variables

       integer cumlev(nbase), nn, i, j

       cumlev(1) = 1
       if(nbase.ge.2) then
          do 1 i=2,nbase
               cumlev(i) = cumlev(i-1) * bselev(i-1)
1         continue
       endif

       nn = n - 1
       do 2 j = nbase, 1, -1
           lindex(j) = nn/cumlev(j)
           nn = nn - lindex(j)*cumlev(j)
2      continue
       return
       end
```

```
        subroutine getconfig(x, n, xx, nprod, maxlev, use, nvar)
C
C  converts a vector x of length n into a "configuration" of length
C  nprod. The configuration corresponds to a marginal table specified
C  by the vector use, of length nvar

C e.g. in a table x of 3 variables A, B, C, the effect(marginal table)
C AB would have use = (1,1,0)

    integer maxlev(nvar), use(nvar)
    double precision xx(nprod), x(n)


C local variables
        integer lindex(nvar), ivec(nvar), j, jj
    integer istar, bvec(nvar), index

    do 3 j=1,nprod
        xx(j) = 0.0D0
3       continue

        do 2 j=1, n
    call convert(j, maxlev, nvar, lindex)
    index = 0
    do 1 jj = 1, nvar
     if(use(jj).eq. 0) goto 1
index = index + 1
ivec(index) = lindex(jj)
bvec(index) = maxlev(jj)
1           continue
    call nvec2j(ivec, bvec, index, istar)
C     print *, 'istar', istar
    xx(istar) = xx(istar) + x(j)
2       continue
    return
    end



C######################################################################
C
C  fortran subroutine to create model matrix for GLM in form
C  suitable for CPLEX input
C  model matrix is nrow by ncol

        subroutine modmat(indmat, maxlev, rmatbeg, rmatind, nterms, nvar,
     .  nrow, ncol, nzcnt)
```

```
       integer  indmat(nterms,*), maxlev(nvar), rmatbeg(ncol)
       integer rmatind(nzcnt)


C local variables

       integer  i,j,n1,index1,lindex1(nvar),n2,index2,lindex2(nvar)
       integer  cumlev(nvar), iprod, nuse, ind, uselev(nvar), nfix
       integer  colind, bselev(nvar), ipos



       rmatbeg(1) = 0
do 10 i=1,nrow
   rmatind(i) = i - 1
10     continue
       ipos = nrow + 1
       colind = 2
       cumlev(1) = 1
       if(nvar.ge.2) then
           do 1 i=2,nvar
        cumlev(i) = cumlev(i-1) * maxlev(i-1)
1          continue
       endif

C    loop over rows of indmat to process each term in turn



       do 2 i=1,nterms


           n1 = 1
   nuse = 0
           do 3 j=1,nvar
               nuse = indmat(i,j) + nuse
              n1 = n1*(maxlev(j) ** indmat(i,j))
3    continue

C compute subvector uselev of cumlev and bselev of max.levels

   ind = 0
   do 4 j = 1,nvar
if(indmat(i,j).eq.1) then
    ind = ind + 1
    uselev(ind) = cumlev(j)
    bselev(ind) = maxlev(j)
```

```
endif
4    continue

C ind will never be 0 provided no row of indmat is all F or all T

C now generate all columns for the current term

   do 5 index1 = 1, n1

C get mixed base representation of row index
C   nuse is rowsum of indmat (= number of variables in effect)
C   bselev is corresponding subvector of maxlev


   call convert(index1, bselev, nuse, lindex1)

C check no level equals baseline. If so, go directly to end of loop

   do 6 j=1, nuse
   if(lindex1(j).eq.0) goto 5
6    continue

C compute nfix

   nfix = 0
   do 7 j=1, nuse
nfix = nfix + lindex1(j)*uselev(j)
7    continue

C  now update  rmatbeg and rmatind

   rmatbeg(colind) = ipos - 1
   call genrow(rmatind, ipos, maxlev,  cumlev, indmat, nfix,
     c  nterms, nrow, nvar, i, nzcnt)
           colind = colind + 1
5    continue
2       continue
        return
        end


       subroutine genrow(rmatind, ipos, maxlev,  cumlev, indmat, nfix,
     c nterms, nrow, nvar, i, nzcnt)


    integer  rmatind(nzcnt),  maxlev(nvar), indmat(nterms, nvar)
        integer  cumlev(nvar)
```

```
C local variables

    integer  n,  j, index, uselev(nvar), bselev(nvar)
    integer ind, nvary,  nuse, lindex(nvar)

C     get number of vars not in effect (nuse)  and loop extent (n)


    n = 1
    nuse = nvar
        do 3 j=1,nvar
            nuse =  nuse - indmat(i,j)
           n = n*(maxlev(j) ** (1-indmat(i,j)))
3     continue

C compute subvector uselev of maxlev

    ind = 0
    do 4 j = 1,nvar
if(indmat(i,j).eq.0) then
    ind = ind + 1
    uselev(ind) = cumlev(j)
    bselev(ind) = maxlev(j)
endif
4     continue

    do 5 index = 1, n

    call convert(index, bselev, nuse, lindex)

C compute nvary

    nvary = 0
    do 7 j=1, nuse
 nvary = nvary + lindex(j)*uselev(j)
7     continue
    rmatind(ipos) = nvary + nfix
    ipos = ipos + 1
5     continue
    return
    end
```