Slice Models in General Purpose Modeling Systems[*]

Michael C. Ferris Meta M. Voelker[†]

December 14, 2000

Abstract

Slice models are collections of mathematical programs with the same structure but different data. Examples of slice models appear in Data Envelopment Analysis, where they are used to evaluate efficiency, and cross-validation, where they are used to measure generalization ability. Because they involve multiple programs, slice models tend to be data-intensive and time consuming to solve. However, by incorporating additional information in the solution process, such as the common structure and shared data, we are able to solve these models much more efficiently. In addition because of the efficiency we achieve, we are able to process much larger real-world problems and extend slice model results through the application of more computationally-intensive procedures.

1 Introduction: Slice Models

In a broad sense, a slice model consists of a group of mathematical programs that are closely related. More specifically, a slice model consists of mathematical programs which use the same model, but different data. Because of this, the basic structure in a slice model remains the same from program to program. Often, the programs are also related through the data: some or most of the data stay the same between programs, differing only in a few rows or columns.

If we consider all of the data for all of the programs at once, we can define a specific program by pulling out its appropriate "slice" of data from the full set.

[*]This material is based on research partially supported by the National Science Foundation under grants CCR-9972372 and CDA-9726385, the Air Force Office of Scientific Research under grant F49620-01-1-0040 and Microsoft Corporation

[†]Computer Sciences Department, University of Wisconsin – Madison, 1210 West Dayton Street, Madison, Wisconsin 53706 (`ferris,voelker@cs.wisc.edu`)

For the k-th slice program, this idea can be expressed as follows:

$$\begin{array}{ll} \min\limits_{\theta,x} & \theta \\ \text{subject to} & A^k \begin{pmatrix} x \\ \theta \end{pmatrix} = b^k \\ & \begin{pmatrix} x \\ \theta \end{pmatrix} \in \mathcal{X} \end{array} \tag{1}$$

where A^k represents the matrix of constraint coefficients which (along with right-hand-sides b^k) are unique to the k-th program. The set $\mathcal{X}$, on the other hand, represents the (core) constraints and program structure which remain constant between programs. Note that the set $\mathcal{X}$ can be very complicated, comprising other general constraints and possibly integrality conditions.

Modeling languages allow us to define slice models easily. Using parameters for the data, we only need to specify the program once. Then, the program can be solved multiple times using different data by simply specifying which data to use for each solve. This has been done for Data Envelopment Analysis models in [13, 16] by using a loop structure, inside which the data is redefined and the solve is re-executed. By using subsets, we can further control which individual programs are actually solved. In addition, because the model is separate from the solver, the same program structure can be used for linear, mixed integer, or non-linear programs and solved under different solvers.

Using the loop structure of a modeling language inside a slice model definition results in multiple model generations and multiple solver calls. But, moving from program k to program $k+1$ in a slice model consists only of a change of the data “slice” — here, the matrix A^k and vector b^k. By exploiting this fact, we can solve slice models more efficiently: rather than starting anew after program k is solved, *modify* that program to get program $k+1$. This approach eliminates the need to build the same structure over and over, and also enables us to use solution information from solve k in solve $k+1$ in hopes of improving overall solution time.

We discuss this modeling approach, implemented as an interface between GAMS and the CPLEX callable library. The interface is general, allowing us to deal easily with different slice models (including both linear and mixed integer models). It also appears to significantly outperform slice models implemented using the looping constructs of the modeling system. Because of this, we are able to solve very large real-world problems which were previously difficult to solve within the general framework. In addition, using this modeling approach, we are able to extend the results to include more information such as confidence intervals. Furthermore, a variety of other classes of mathematical programming models can easily be formulated to exploit our interface.

In the next section, we discuss an example of slice models, Data Envelopment Analysis (DEA) models, which are used for efficiency evaluation. Section 3 describes the implementation of the slice interface in GAMS and gives examples of its use, including a DEA model for evaluating the efficiency of hospitals and a cross-validation model for feature selection. Finally, extensions of slice modeling and the slice interface are discussed in Section 4, where we obtain confidence intervals for the efficiency scores of the hospitals.

2 DEA Models as Slice Models

Data Envelopment Analysis (DEA) is a technique to evaluate the relative performance of a number of decision-making-units (DMUs). A DMU can be anything that takes in inputs and produces outputs; examples include producers, banks, and even schools. DEA focuses on identifying inefficient DMUs by evaluating the set of DMUs with respect to the "best" DMU. This "best" DMU can either be an actual DMU in the set or a composite created from attributes of multiple DMUs.

The real power behind DEA is its ability to deal with multiple inputs and multiple outputs, without requiring that these inputs and outputs be related in any functional form [2]. Originally, DEA was developed by Charnes et al. [5] as an extension of Farrell's efficiency analysis techniques [11] to not-for-profit DMUs, where the data were not related economically. Their first application was to evaluate the efficiency of the educational program Program Follow Through [7]. In their evaluation, the inputs included the education level of the mother, the highest occupation of a family member, parental time spent counseling the child, and the number of teachers on site; the outputs included reading test scores, mathematics test scores, and self-esteem measures. Although these data were difficult to relate to each other and weight directly for evaluation purposes in the traditional manner, they were easily evaluated under the DEA approach. Since then, the basic DEA models have been extended and modified for a variety of applications [8].

To determine the relative performance of a DMU, we first define efficiency in terms of the inputs and outputs. We let the vector $Y_{\cdot,k}$ represent the set of outputs for the k-th DMU and the vector $X_{\cdot,k}$ represent the set of inputs. Then efficiency for the k-th DMU can be defined by

$$\text{efficiency} = \frac{\text{weighted sum of outputs}}{\text{weighted sum of inputs}} = \frac{u^T Y_{\cdot,k}}{v^T X_{\cdot,k}}.$$

Under DEA, the inputs (X) and outputs (Y) are data; the weights (u, v) are variables. For each DMU, a different set of weights may be used; this allows for different operational organization and/or different valuation of inputs and outputs [9]. Obviously, each DMU would want to choose its most favorable set of weights. This raises the questions of (1) determining each DMU's best weights, and (2) comparing DMUs effectively based on different choices of weights.

Initially, we take $X, Y > 0$ — so that every DMU takes in every input and produces every output. Then, to determine the most favorable set of weights for the k-th DMU, we define the general fractional CCR model (named for Charnes et al. for their formulation in [5, 6]) by:

$$\begin{array}{ll}\max\limits_{u,v} & h = \frac{u^T Y_{\cdot,k}}{v^T X_{\cdot,k}} \\ \text{subject to} & \frac{u^T Y}{v^T X} \leq 1 \\ & u, v > 0\end{array} \tag{2}$$

In this model, we maximize the efficiency score of DMU k subject to the constraint that the efficiency scores for all DMUs are less than or equal to 1 (maximum efficiency). Here, u and v are restricted to be positive so that every input and every output is considered in the analysis [6]. In order to compare DMUs based on efficiency, a problem of this form must be solved for *each* DMU in the set.

To convert the general fractional CCR model to a linear programming model, we invoke positive homogeneity, set the denominator of h equal to a constant and multiply the first constraint through by its denominator to remove any fractional portions. Because fractional portions are removed, the initial assumptions on X, Y can be relaxed to assuming that every DMU takes in some positive input and produces some positive output: $X, Y \geq 0$ with $X_{\cdot,k} \neq 0, Y_{\cdot,k} \neq 0 \ \forall i$. Typically, X, Y are dense matrices. Because there is no longer the possibility of undefined fractions, we also relax the last constraint, requiring that $u, v \geq 0$ only (in doing so we find weak efficiency [8]). These modifications result in what is commonly referred to as the dual DEA model:

$$\begin{array}{ll}\max\limits_{u,v} & u^T Y_{\cdot,k} \\ \text{subject to} & v^T X_{\cdot,k} = 1 \\ & u^T Y \leq v^T X \\ & u, v \geq 0\end{array} \tag{3}$$

In this model, we look to maximize efficiency by directly manipulating the weights.

Taking the dual of (3), we get the primal DEA model:

$$\begin{array}{ll}\min\limits_{\theta,\lambda} & \theta \\ \text{subject to} & \theta X_{\cdot,k} \geq X\lambda \\ & Y\lambda \geq Y_{\cdot,k} \\ & \lambda \geq 0\end{array} \tag{4}$$

Here, rather than manipulating the weights directly, we look for a composite DMU (with inputs $X\lambda$ and outputs $Y\lambda$) that produces just as much output as DMU k but that uses at most a fractional amount ($\theta X_{\cdot,k}$) of input.

DEA models fit the definition of slice models: a complete DEA model consists of multiple linear programs (one for each DMU), each of which has the same structure but different data. In fact, these models are even more closely related than just structurally — a core set of data remains the same in each program. For the dual model (3), only one row and the objective function change (resulting from the condition that all of the efficiency scores must be less than or equal to 1); in terms of model (1), this implies that

$$A^k = \begin{bmatrix} Y_{\cdot,k}^T & 0 & -1 \\ 0 & X_{\cdot,k}^T & 0 \end{bmatrix},\ b^k = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

and

$$\mathcal{X} = \left\{ \begin{pmatrix} u \\ v \\ \theta \end{pmatrix} \middle| u^T Y \le v^T X; u, v \ge 0 \right\}.$$

For the primal model (4), one column and the right-hand-side change; in terms of model (1), this implies that

$$A^k = \begin{bmatrix} X & I & 0 & -X_{\cdot,k} \\ Y & 0 & -I & 0 \end{bmatrix},\ b^k = \begin{bmatrix} 0 \\ Y_{\cdot,k} \end{bmatrix} \tag{5}$$

and

$$\mathcal{X} = \left\{ \begin{pmatrix} \lambda \\ s_x \\ s_y \\ \theta \end{pmatrix} \middle| \lambda, s_x, s_y \ge 0 \right\}. \tag{6}$$

Here, the slices contain most of the data and the core contains very little. If we introduce two auxiliary variables for $X\lambda$ and $Y\lambda$, then (5) and (6) become:

$$A^k = \begin{bmatrix} 0 & I & 0 & -X_{\cdot,k} & I & 0 \\ 0 & 0 & -I & 0 & 0 & I \end{bmatrix},\ b^k = \begin{bmatrix} 0 \\ Y_{\cdot,k} \end{bmatrix} \tag{7}$$

$$\mathcal{X} = \left\{ \begin{pmatrix} \lambda \\ s_x \\ s_y \\ \theta \\ f_x \\ f_y \end{pmatrix} \middle| \begin{matrix} f_x = X\lambda; f_y = Y\lambda \\ \lambda, s_x, s_y \ge 0 \end{matrix} \right\}. \tag{8}$$

The data in each slice is then reduced to $3 * |DMU|$ non-zeros and the core part of the model contains all the significant data.

Because many real-world DEA problems have numerous DMUs and multiple inputs and outputs, the evaluation process can be very data-intensive and thus very time consuming under general linear programming solvers. Thinking of DEA models as slice models enables us to keep the core data constant between individual problems. Defining individual problems then consists of just adding specific data. This approach reduces problem generation time, and also allows us to easily include previous solution information (such as basic rows or columns) for the core data.

Specialized DEA solvers, which take advantage of the common structure between programs, do exist (for example, see [17]). However, these solvers assume knowledge of the DEA models (models (3) and (4), as well as some other common ones) and only take in the data X, Y. This works well when the model to be solved is one of the assumed DEA models, but sometimes the model is an extension or variation of an assumed form instead. For example, Banker and Morey [3] explored incorporating into DEA models non-discretionary variables, which affect output but which the DMUs have no control over. Banker and Morey [4] also explored the idea of categorical variables, which allow DMUs to be classified and compared only to others in the same class. When the DMUs can control their classification, the resulting model becomes a mixed-integer model [4, 14]. Allen et al. [1] explored some examples where model-specific restrictions and constraints were added to general DEA models in order to include "prior knowledge or accepted views" in the model. These examples show that the ability to not only define the data but also the model is of importance for DEA applications, thus suggesting the use of a modeling language.

3 The Slice Interface

Solving slice models involves looping over the data slices, including specific slices in the model. Even if there is no or very little core data, the structure of each program is the same and so can be held constant. Often with general solvers, no mechanism is available to store the structure and core data; each program must be generated and solved from scratch, ignoring completely the common ties that the programs have with each other. For example, the DEA model implementations [16, 13] for the GAMS modeling language make use of the GAMS loop structure within the expression of the model. This loop structure defines the particular programs which are passed the general solver and includes the solver call. As a result, multiple calls are made to the general solver, each accompanied by a new model generation. Some optimal basis information may be passed on (depending upon option settings), but the structure and data are entirely regenerated, resulting in huge solution times for models with many DMUs.

As an alternative, we suggest that similarities between the programs be considered. To do this, we have built an interface between the GAMS modeling language and the CPLEX callable library. This slice interface uses the GAMS dictionary to identify the changes between programs and the CPLEX problem modification routines to make the changes — all automatically. In this way, we are able to reuse the common structure and core data without having to regenerate them for each program.

3.1 Implementation

Unlike the general slice model implementations in [13, 16], *all* of the information for all of the programs is passed into the interface at once. The interface then separates this information into program structure and core rows ($\mathcal{X}$), and program-specific rows (A^k and b^k), based upon the modeler's instructions. Once the interface has labeled the constraints and received the data from GAMS, the core rows and structure are read into a CPLEX problem object. At this stage, we solve the slice model. This is done in a loop (replacing the loop previously written in GAMS) over the slice set. At each iteration, specific rows are added to the core program object. The resulting program is then solved and its results are printed to a solution file. Next, the specific program is returned to its core state while some solution information (basis information for linear programs and starting point information for mixed integer programs) is passed on to the next solve. Once the last solve is done, we return the last solution to GAMS. In this way, the slice interface enables us perform the multiple solves needed by slice models, while at the same time keeping program structure constant and using advance starting information.

To instruct the interface on which constraints are core constraints and which are program-specific, the modeler needs to do very little. Figure 1 compares the general GAMS implementation to the GAMS slice implementation for the dual DEA model (3), and shows that very little must be changed. First, the set of DMUs (`n`) which are being evaluated must be identified. This is done by associating a special name (`slice`) with the set through an alias command. Then, the constraints which contain slice data must be identified (`objfcn` and `denom`). This is done by *declaring* these constraints over the special alias name; these constraints can then be defined over any subset of the DMU set. Core constraints (`lime`) are declared and defined normally. Note that we do not have to worry about changing objective functions: in GAMS the objective function is given as an objective variable — the data associated with the objective function is actually stored in the constraint matrix (`objfcn`).

Although a direct translation from the general GAMS implementation is easy to produce for the GAMS slice implementation, it is not always the best formulation to use. Under our approach, the entire problem is generated and passed to the slice interface initially. This works fine for models like model (3) because the extra constraints that we must generate are few in comparison to the whole model. However, in other models like model (4), almost all of the constraints change and so many more constraints are passed initially to the solver. This results in very high problem generation times, which can destroy any time-savings the slice interface achieves during the actual solves. Dealing with this issue involves adjusting the model to fit the way the solver works: models like model (4) change by column slices (variables and right-hand-side values); the slice interface, on the other hand, makes use of row slices (constraints and objective function coefficients). In modeling using column slices, many constraints change overall, but the changes within individual constraints are minor. As a result, we end up regenerating pieces of the model — exactly one of the problems that we hoped to eliminate by using the slice interface. However, by storing the pieces of these constraints that remain constant in auxiliary variables, as presented in equations (7) and (8), we are able to eliminate much of this regeneration: we still must generate many more constraints, but

```
sets  n    ‘DMUs’,
      k(n) ‘selected unit’,
      i    ‘inputs’,
      o    ‘outputs’;



equations
objfcn(n) ‘efficiency defn’,
denom(n)  ‘weighted input’,
lime(n)   ‘output/input<1’;

objfcn(k).. sum(o, u(o)*Y(o,k))
            =e= eff;

denom(k)..  sum(i, v(i)*X(i,k))
            =e= 1;

lime(n)..  sum(o, u(o)*Y(o,n)) =l=
           sum(i, v(i)*X(i,n));


set nloop(n) ‘DMUs to analyze’;
* run over all DMUs
nloop(n) = yes;

k(n) = no;
loop(nloop,
  k(nloop) = yes;
  solve dea using lp max eff;
  k(nloop) = no;
);
```

(a)

```
sets  n    ‘DMUs’,
      k(n) ‘selected units’,
      i    ‘inputs’,
      o    ‘outputs’;
alias(n,slice);


equations
objfcn(slice) ‘efficiency defn’,
denom(slice)  ‘weighted input’,
lime(n)       ‘output/input<1’;

objfcn(k).. sum(o, u(o)*Y(o,k))
            =e= eff;

denom(k)..  sum(i, v(i)*X(i,k))
            =e= 1;

lime(n)..   sum(o, u(o)*Y(o,n)) =l=
            sum(i, v(i)*X(i,n));



* run over all DMUs
k(n) = yes;




solve dea using lp max eff;
```

(b)

Figure 1: DEA model implementation for (3) using (a) the general GAMS interface and (b) the GAMS slice interface.

```
equations
di(slice,i) ‘input constr’,
do(slice,o) ‘output constr’;



di(k,i)..  theta*X(i,k) =g=
           sum(n, X(i,n)*lambda(n));

do(k,o)..  sum(n, Y(o,n)*lambda(n))
           =g= Y(o,k);
```

(a)

```
equations
di(slice,i) ‘input constr’,
do(slice,o) ‘output constr’,
dfx(i)      ‘aux input var’,
dfy(o)      ‘aux output var’;

di(k,i)..  theta*X(i,k) =g= fx(i);


do(k,i)..  fy(o) =g= Y(o,k);


dfx(i)..   fx(i) =e=
           sum(n, X(i,n)*lambda(n));

dfy(i)..   fy(o) =e=
           sum(n, Y(o,n)*lambda(n));
```

(b)

Figure 2: Column-based slice model (4) under (a) direct translation into GAMS and under (b) auxiliary variables.

these constraints now have few variables in them. Although this increases the number of variables in the model, these extra variables can be eliminated during presolve, once they are no longer useful. Figure 2 compares the direct GAMS translation of model (4) to the improved model formulation with auxiliary variables (`fx` and `fy`). Using auxiliary variables can significantly reduce problem generation time, since the data X and Y are only generated once.

3.2 Technical Issues

To implement the slice procedure, we first must be able to identify the slices. This is done using the GAMS dictionary file. Under GAMS, additional information about the model is available through the dictionary file. Contained within the dictionary is the unique element list, which lists and indexes the unique element names that have been used in the model. Inside this list, alias names are listed separately, but linked to the index for their corresponding element. In a similar manner, equation names are linked to the elements which they are declared over. So, program-specific constraints can be identified by searching equation declarations for the slice alias. Because constraints are only generated for sets which they are declared over, we are able to determine which DMUs (if any) are to be skipped in the analysis simply by examining which constraints are actually generated.

After implementing our slice interface, we encountered scaling problems within our solution loop. Under default CPLEX scaling, the entire program is scaled when it is initially read in. However, it is not fully rescaled when the program is changed. As a result, we encountered numerical errors in some solutions due to poor scaling. To correct this, we unscale the problem prior to changing it. This enables us to return the problem to its true core state and guarantees that when scaling is done (for instance, during the solve), it is done to the problem being solved and not just pieces of the problem.

Besides these implementation issues, we must also deal with the issue of solution availability. Under the GAMS loop implementation, solution information for each program is accessible from within GAMS because each program's solution is returned to GAMS. Under the GAMS slice implementation, only the solution of the last slice problem is returned to GAMS. However, solution information for all of the programs is written to a solution file. To make this solution information accessible, the solution file is formatted so that the information is written to GAMS parameter values, indexed by slice. Refer to Figure 3 for an example of a partial solution file generated by solving model (3). Under defaults, the model status (`modelstat`), solver status (`solvestat`) and objective value (`objval`) are given. Other variable and constraint values can also be given, and are indexed not only by slice, but also by their type (`'prim'` for primal variable, `'dual'` for dual variable, `'slack'` for slack, and `'rc'` for reduced cost) and any model dependencies. The parameter names are built by appending "val" to the original name found in the model. This enables us to read the solution information directly back into GAMS for further analysis. Time and solution statistics are also given as GAMS comments at the end of the file.

3.3 DEA Applications

To test the slice interface, we consider a DEA application: measuring the efficiency of 104 German hospitals from [15]. In this application, the inputs included the number of beds and the annual cost of care, while the outputs included the number of cases per hospital department (for the 1992 data used in [15], 18 hospital departments were considered). To determine efficiency for each hospital, we used model (4), which resulted in 49 efficient hospitals and 2 very inefficient (efficiency scores less than 10%) hospitals. The time comparison between the GAMS model under CPLEX 6.6 and the GAMS model under the slice interface are given in the first row of Table 1. As can be seen from these results, we were able to solve the DEA model is less time.

Another application truly shows the power of the slice interface. For this application, we use a variation on the basic DEA models to solve two different sized problems provided by Walden [22]: a small one involving 60 DMUs and a large one involving 4888 DMUs. Results comparing the GAMS model under CPLEX and the GAMS model under the slice interface are given in the second and third rows of Table 1. In the small problem we see some time improvement, just as we did for the hospital application. But, in the large problem, we see a significant improvement in time: 16 hours versus approximately 12 minutes. This result especially shows the power of the slice interface for DEA models: we

```
parameter modelstat;
parameter solvestat;
parameter objval;
parameter vval;
parameter uval;
parameter effval;
parameter objfcnval;
parameter denomval;
parameter limeval;

modelstat('1') = 1;
solvestat('1') = 1;
objval('1') = 0.82038345;
vval('prim', '1', 'input1') = 0.29969728;
uval('prim', '1', 'output1') = 0.00524723;
effval('prim', '1') = 0.82038345;
uval('rc', '1', 'output3') = -4.83451060;
limeval('dual', '1', '12') = 0.50353179;
objfcnval('dual', '1', '1') = 1.00000000;
denomval('dual', '1', '1') = 0.82038345;
limeval('slack', '1', '1') = 0.17961655;

modelstat('2') = 1;
solvestat('2') = 1;
objval('2') = 0.94174174;
vval('prim', '2', 'input1') = 0.35675676;
uval('prim', '2', 'output1') = 0.00624625;
effval('prim', '2') = 0.94174174;
uval('rc', '2', 'output3') = -1.45585586;
limeval('dual', '2', '12') = 0.77537538;
objfcnval('dual', '2', '2') = 1.00000000;
denomval('dual', '2', '2') = 0.94174174;
limeval('slack', '2', '1') = 0.21381381;


*Time to read data: 0
*Calculation time: 0.03
*Time since initialization call: 0.03

*Number of scaling changes for unscaled infeasibilities: 0
*Maximum unscaled (scaled) primal infeasibility: 0 (0)
*Maximum unscaled (scaled) dual   infeasibility: 2.13163e-14 (9.23706e-14)
*Maximum unscaled (scaled) primal residual:      2.77556e-16 (1.38778e-17)
*Maximum unscaled (scaled) dual   residual:      3.55271e-15 (3.55271e-15)
*Maximum scaled basis condition number: 622.855
```

Figure 3: Partial solution file for model (3) written by the slice interface.

Application Problem	General Solver (CPLEX)	Slice Interface
Hospital Model (104 DMUs)	7.02 sec	0.51 sec
Small Model (60 DMUs)	4.63 sec	0.43 sec
Large Model (4888 DMUs)	16 hours	717.01 sec

Table 1: Time comparisons between the general GAMS implementations and the GAMS slice implementations.

Application Problem	Advanced Basis	Presolve
Hospital Model (104 DMUs)	0.51 sec	0.84 sec
Small Model (60 DMUs)	0.43 sec	0.45 sec
Large Model (4888 DMUs)	717.01 sec	1520.23 sec

Table 2: Time comparisons with advanced basis information (no presolve) and with presolve (no advanced basis).

can solve general DEA models with many DMUs much more efficiently.

3.4 Solution Options

Under GAMS, solver behavior can be modified through the use of an options file. Many options are available for the slice interface. One is the choice of solution algorithm. For linear programs, the slice interface initially chooses the dual simplex method for minimization models and the primal simplex method for maximization models. These defaults were chosen based upon the performance of our interface under test problems, but can be changed in the options file. Mixed integer problems automatically use the CPLEX MIP solver.

Advanced starting information is also important to the slice interface. When an advanced starting basis is used, CPLEX (version 6.6) skips the presolve stage. So, whether an advanced basis is used or not greatly affects how the individual programs are solved. Under our tests on general linear DEA models, using advanced basis information (and skipping the presolve stage) in subsequent solves can greatly improve solution time. Table 2 shows the solution times for the DEA models from Section 3.3 using advanced basis information and using presolve. These results show time improvement in all three cases, with significant improvement for the large model, where using advanced basis information cut the solution time in half. Based on these results, the default option for linear programs under the slice interface is to use advanced basis information. Because changing a program destroys advanced starting information, we explicitly copy the advanced basis from one program to the next in the interface code. The copy, though, is ignored if use of the advance basis is turned off in the options file.

Besides affecting the solver, options can also affect the way in which the programs are defined. For the slice interface, a special option, model type, does this. DEA models require that slices be added to the core model, but other types of slice models exist. One type, cross-validation models, works differently. Cross-validation measures a prediction model's error using the only the available data set to train and test the model. In k-fold cross-validation [10], the available data set is divided into k (generally about equal) pieces. Then the model is trained k times, each time leaving one of the k pieces out of the training set and using it as the testing set instead. Performing cross-validation on prediction models involving mathematical programming can be regarded as another example of slice modeling because the resulting models are programs with the same structure but different data. But, unlike DEA models in which slices are added to the core data, cross-validation models require that slices be deleted from the core (training) data: the k-th individual program can be defined by deleting the k-th piece. In this case, the "slicing" is done by elimination: everything *except* the particular slice is included in the program. Under the slice interface, this is achieved by setting the model type to deletion.

As an example of cross-validation, we consider the feature selection models of Ferris and Munson [12], who use 10-fold cross-validation to select the number of features with the best predictive capability. Under their approach, 10 mixed integer programs are generated each time they perform cross-validation, each resulting in 10 separate problem generations and solver calls. To define the training and testing data, they define dynamic sets inside the GAMS loop, immediately prior to the solve statement. In order to implement this under the slice interface, we modify their model to remove the dynamic sets, instead using the slice set, the number of folds (`p`), in the constraints. This allows us to generate all of the constraints at once. Figure 4 shows the main changes between the original formulation and the slice formulation. In the original formulation (a), the training sets (`a_trai` and `b_trai`) and testing sets (`a_test` and `b_test`) are defined inside the GAMS loop. In the slice formulation (b), the testing sets are indexed by the slice set (`p`) and are completely defined (in `a_test` and `b_test`) prior to the solve statement. In effect, for each slice in `p`, the set `a_test` extracts some of the rows from the full set `a` into a testing set. Because the model makes use of cross-validation, we use the deletion model type and define the sets by what gets *deleted* (the testing sets), so we no longer need the training sets. The sum over the training sets in the objective function (`c_def`) is replaced by a sum over the full set; any variables which appear in the objective function and which are related to the testing set will be eliminated by presolve because they will be defined no where else in the specific program. In addition, the cardinality calls appearing in the objective function are replaced by scalars (`a_card` and `b_card`), which are also defined prior to the solve.

For comparison, we ran the cross-validation models on the galaxy data set with 6, 7, and 8 features. Each model achieved the same objective value and selected the same features. After solving, we read the solutions from the slice interface back into GAMS in order to evaluate the misclassification error on the testing data. The average expected misclassification error is given in Table 3. Table 4 shows the solution times for both the original formulation and the

```


set a_test(a) ‘a testing set’;
set a_trai(a) ‘a training set’;
set b_test(b) ‘b testing set’;
set b_trai(b) ‘b training set’;



c_def..
c =e= (sum(a_trai,a_class(a_trai))/
           card(a_trai) +
       sum(b_trai,b_class(b_trai))/
           card(b_trai));

a_def(a_trai)..
-sum(o, a_data(a_trai,o)*weight(o))
  + gamma + 1 =l= a_class(a_trai);

b_def(b_trai)..
 sum(o, b_data(b_trai,o)*weight(o))
  - gamma + 1 =l= b_class(b_trai);
```

(a)

```
alias(p,slice);

set a_test(a,p) ‘a testing set’;

set b_test(b,p) ‘b testing set’;



c_def..
c =e= sum(a,a_class(a))/
          a_card +
      sum(b,b_class(b))/
          b_card;

a_def(a,p)$a_test(a,p)..
-sum(o, a_data(a,o)*weight(o))
  + gamma + 1 =l= a_class(a);

b_def(b,p)$b_test(b,p)..
 sum(o, b_data(b,o)*weight(o))
  - gamma + 1 =l= b_class(b);
```

(b)

Figure 4: Cross-validation comparison for (a) dynamic set model and (b) deletion slice model.

Features	Expected Misclassification
6	1.102041%
7	1.224490%
8	1.183673%

Table 3: Average expected misclassification for the feature selection problem under the galaxy data set with 6, 7, and 8 features.

slice interface under various options. In all cases, the slice interface is faster by around 40-50 seconds.

The performance of mixed integer programs can depend a great deal upon the options that are set. For these types of programs, starting points can be important: the availability of even a partial starting point can improve solution time. If a starting point is provided and used, CPLEX checks to see if the starting point provides an integer feasible solution prior to starting the solve. If it does, the solver then starts immediately from a feasible point. In table 4, the times listed include those for solving the models with starting points (SP)(arising as the solution of the previous problem in the sequence) and without starting points (NSP). For both the original formulation and the slice formulation, the use of starting points improved the solution times. These results suggest that starting points are beneficial to general slice models. Because of this, the slice interface by default copies the solution values for integer variables to the next program to use as a (partial) starting point. Although we get improved solution times, starting points should not be used for cross-validation models in order to make the solves independent.

Cutoff values can also affect the solution performance. In table 4, the CV times are the times for solving the models without starting points but using problem-specific cutoff values. The cutoff values were taken to be the corresponding objective value for the feature selection problem with one less feature. By using appropriate cutoff values, we were able to obtain times that were around or better than the times obtained when using starting points (SP). Further, because the cutoff values came from a different model and were determined prior to starting the solution process for the current model, all of the solves were done independently.

4 Beyond DEA: Sensitivity Analysis

Because of the time improvements over general GAMS implementations, slice model solutions can be extended through the application of other procedures to include additional information like confidence intervals. In finding efficiency scores under DEA, we use the provided DMU data to determine feasible input-output points (both actual DMU points and composite DMU points). DEA efficiency scores are then determined based upon these feasible points. Thus, changes in the DMU data result in changes to the feasible points and to the

Features		Original Formulation	Slice Formulation
6	NSP	487.22 sec	433.05 sec
	SP	229.62 sec	189.94 sec
	CV	223.26 sec	176.30 sec
7	NSP	306.73 sec	258.39 sec
	SP	229.51 sec	191.61 sec
	CV	215.73 sec	171.16 sec
8	NSP	251.39 sec	210.07 sec
	SP	191.07 sec	151.36 sec
	CV	196.04 sec	154.71 sec

Table 4: Time comparisons between the original (dynamic set) formulation [12] and the slice formulation for the feature selection problem under the galaxy data set with 6, 7, and 8 features. Times are given for the models without starting points (NSP), with starting points (SP) and with cutoff values (no starting points) (CV).

efficiency scores. To determine how sensitive the efficiency scores are to changes in data, we would like to obtain additional feasible input-output points against which we can measure the efficiency scores for the original DMUs. Statistical methods offer some ways in which this can be done.

In [19, 20, 21] Simar and Wilson make use of bootstrap methods for obtaining additional input-output points by repeatedly resampling from the original data set. Under their approach, a smooth bootstrap is used to build an estimate for the probability density function of the original DEA efficiency scores. From this estimate, new efficiency scores are drawn and are used to obtain corresponding input-output data. Using this new data, the DEA model for the DMUs is re-solved. Repeatedly applying this approach results in sampling distributions for the DMUs' efficiency scores, which can be used to determine confidence intervals.

Modeling languages offer advantages when applying Simar and Wilson's bootstrap approaches. Under a modeling language, all of the models can be built from one program specification by using parameters in the program to allow for the changing data. Further, there is no need to go outside in order to obtain the new data for each model: we can actually build this data inside the model specification. In building the probability density function estimate for the original DEA scores, we use a smoothing parameter in the bootstrap; this smoothing parameter is obtained by maximizing a log-likelihood function, which can be done under GAMS using one of its non-linear solvers. To actually draw the samples from the estimated function, we use the algorithm from [18]; this too can be done in GAMS using its predefined probability density functions. The new input-output data is defined in terms of the old data and the samples; this turns into simple parameter re-definitions in GAMS. In this way, not only can the resulting models be solved, but they can also be defined in GAMS.

In their examples from [19, 20], Simar and Wilson draw 1000 samples; however, they note in [21] that 2000 or more samples may be needed for accurate confidence intervals. Note that each sample involves a DEA loop consisting of a sequence of linear programs, one for each DMU. Thus, in order to obtain confidence intervals on the efficiency scores, the complete DEA model must be solved 1000 or more times, each time with different data. The slice interface can be used for each of these 1000 samples, making the overall model solution much more efficient. This significantly reduces the solution time and allows us to analyze the scores for DEA models involving large numbers of DMUs.

We applied the techniques from [19] to the hospital model from Section 3.3, considering only hospitals at least 10% efficient. Taking 1000 samples under the slice interface took 463.41 sec (approximately 8 minutes). Without the slice interface, the time would have been close to 2 hours, even for this small example. The results are displayed in Table 5. Of the 49 efficient hospitals, only 6 had tight confidence intervals, contained within 90%–100%. 29 others had very wide confidence intervals, with 17 of these spanning over half the efficiency range. The remaining efficient hospitals had confidence intervals around 75%–100%. These results suggest that an efficient rating is very data-dependent for many hospitals in this application. We conjecture that the use of slice modeling within these applications will make such analyses commonplace.

5 Conclusion

In this paper, we have discussed slice modeling. Using a slice model formulation, we have shown how defining individual programs from the model by data slices suggests a new solution technique. By implementing this technique in the slice interface, we have been able to take advantage of the fact that program structure and core data stay the same between programs. This has lead to faster solution times, as we have demonstrated on real-world examples from both DEA and cross-validation. In addition, we have been able to extend the results of DEA models to include confidence intervals by the application of the computationally-intensive bootstrap.

Because the slice interface has been built for a general modeling language, we have been able to solve very different slice models, from linear DEA models to mixed integer cross-validation models. But, nothing in the slice interface limits us to only these types of models; any linear or mixed integer slice model can make use of the slice interface. For example, determining the effects of particular scenarios on a robust solution in the framework of stochastic optimization can be formulated as a slice model and solved under the slice interface (provided the scenarios are made available in advance for the initial model generation). This allows us to extend the efficiency we achieve under DEA and cross-validation models to other areas.

Hospital	Efficiency	95% CI		Hospital	Efficiency	95% CI	
1	1.0000	0.8009	0.9961	53	1.0000	0.9467	0.9959
2	1.0000	0.4072	0.9958	54	0.8937	0.8434	0.8909
3	0.0759	–	–	55	0.8780	0.7222	0.8743
4	1.0000	0.5843	0.9958	56	0.9418	0.8663	0.9378
5	1.0000	0.4315	0.9961	57	0.6271	0.5932	0.6253
6	1.0000	0.7119	0.9962	58	0.8471	0.7971	0.8437
7	1.0000	0.4951	0.9963	59	0.9943	0.9010	0.9901
8	0.9766	0.9103	0.9719	60	0.8953	0.7968	0.8915
9	1.0000	0.6372	0.9958	61	0.9196	0.7825	0.9152
10	0.8486	0.8092	0.8453	62	1.0000	0.8116	0.9960
11	0.9301	0.8588	0.9259	63	1.0000	0.6499	0.9961
12	1.0000	0.7779	0.9963	64	0.9449	0.8958	0.9406
13	1.0000	0.3235	0.9957	65	0.9690	0.9391	0.9665
14	0.9847	0.9378	0.9804	66	0.7712	0.7391	0.7680
15	0.9862	0.8601	0.9822	67	0.9753	0.9327	0.9715
16	1.0000	0.3925	0.9958	68	0.6963	0.6473	0.6944
17	1.0000	0.7746	0.9956	69	1.0000	0.6602	0.9949
18	0.9106	0.8650	0.9082	70	0.9042	0.8670	0.9000
19	1.0000	0.3968	0.9959	71	1.0000	0.4671	0.9960
20	0.6225	0.6017	0.6209	72	0.9485	0.9009	0.9442
21	1.0000	0.8734	0.9960	73	0.6970	0.6411	0.6950
22	0.9045	0.8307	0.9004	74	0.9620	0.9003	0.9578
23	1.0000	0.7786	0.9960	75	1.0000	0.8009	0.9954
24	1.0000	0.9516	0.9957	76	0.9452	0.7538	0.9422
25	0.9834	0.8755	0.9793	77	1.0000	0.6386	0.9960
26	0.9895	0.9144	0.9852	78	1.0000	0.5638	0.9956
27	0.7987	0.7554	0.7960	79	0.5121	0.4424	0.5103
28	0.9049	0.8636	0.9012	80	1.0000	0.8819	0.9959
29	1.0000	0.4157	0.9955	81	0.8067	0.7121	0.8041
30	0.8633	0.7644	0.8598	82	0.9497	0.8396	0.9457
31	0.8462	0.8103	0.8440	83	1.0000	0.2864	0.9955
32	1.0000	0.8894	0.9961	84	0.8727	0.8037	0.8689
33	0.9343	0.8551	0.9300	85	0.8717	0.8154	0.8677
34	1.0000	0.7809	0.9956	86	0.9104	0.8256	0.9063
35	0.7360	0.7039	0.7330	87	1.0000	0.9501	0.9960
36	0.9118	0.8672	0.9081	88	0.8791	0.8363	0.8752
37	1.0000	0.4090	0.9959	89	1.0000	0.5280	0.9960
38	1.0000	0.3226	0.9960	90	0.8148	0.7044	0.8115
39	1.0000	0.4643	0.9962	91	0.8410	0.7944	0.8376
40	0.8321	0.7487	0.8291	92	1.0000	0.6551	0.9958
41	0.9381	0.8564	0.9345	93	1.0000	0.9044	0.9954
42	0.9379	0.8132	0.9339	94	1.0000	0.8561	0.9958
43	1.0000	0.8813	0.9957	95	0.9522	0.8991	0.9485
44	0.2054	0.1991	0.2049	96	0.7874	0.7172	0.7844
45	1.0000	0.3591	0.9956	97	1.0000	0.2958	0.9962
46	1.0000	0.5923	0.9955	98	0.7959	0.7604	0.7938
47	1.0000	0.3534	0.9964	99	1.0000	0.4211	0.9956
48	0.0473	–	–	100	1.0000	0.8133	0.9956
49	1.0000	0.8959	0.9962	101	1.0000	0.3027	0.9958
50	0.6349	0.5398	0.6324	102	1.0000	0.6236	0.9960
51	0.9664	0.9046	0.9621	103	1.0000	0.9010	0.9955
52	1.0000	0.7878	0.9957	104	1.0000	0.3961	0.9957

Table 5: Confidence intervals for hospitals at least 10% efficient.

Acknowledgements

We would like to thank Alex Meeraus for introducing us to DEA, and Steven Dirkse for providing us with additional dictionary routines to facilitate our interface. In addition, we would like to thank Stefan Scholtes and John Walden for providing us with real-world DEA applications.

References

[1] R. Allen, A. Athanassopoulos, R. G. Dyson, and E. Thanassoulis. Weights restrictions and value judgements in Data Envelopment Analysis: Evolution, development and future directions. *Annals of Operations Research*, 73:13–34, 1997.

[2] T. Anderson. A Data Envelopment Analysis (DEA) home page. HTML document, June 1996. http://www.emp.pdx.edu/dea/homedea.html (1 September 1999).

[3] R. D. Banker and R. C. Morey. Efficiency analysis for exogenously fixed inputs and outputs. *Operations Research*, 34(4):513–521, July–August 1986.

[4] R. D. Banker and R. C. Morey. The use of categorical variables in Data Envelopment Analysis. *Management Science*, 32(12):1613–1627, December 1986.

[5] A. Charnes, W. W. Cooper, and E. Rhodes. Measuring the efficiency of decision making units. *European Journal of Operational Research*, 2:429–444, 1978.

[6] A. Charnes, W. W. Cooper, and E. Rhodes. Short communication: Measuring the efficiency of decision-making units. *European Journal of Operations Research*, 3:339, 1979.

[7] A. Charnes, W. W. Cooper, and E. Rhodes. Evaluating program and managerial efficiency: An application of Data Envelopment Analysis to Program Follow Through. *Management Science*, 27(6):668–697, June 1981.

[8] W. W. Cooper, L. M. Seiford, and K. Tone. *Data Envelopment Analysis: A Comprehensive Text with Models, Applications, References and DEA-Solver Software*. Kluwer Acaemic Publishers, Boston, 2000.

[9] R. G. Dyson, E. Thanassoulis, and A. Boussofiane. A DEA (Data Envelopment Analysis) tutorial. HTML document, July 1995. http://www.warwick.ac.uk/~bsrnb/pages/links/dea.htm (1 September 1999).

[10] B. Efron and R. J. Tibshirani. *Introduction to the Bootstrap*. Chapman and Hall, New York, 1993.

[11] M. J. Farrell. The measurement of productive efficiency. *Journal of the Royal Statiscal Society, Series A (General)*, 120(3):253–290, 1957.

[12] M. C. Ferris and T. S. Munson. Modeling languages and condor: Metacomputing for optimization. *Mathematical Programming*, 88(3):487–505, September 2000.

[13] GAMS Development Corporaton. Data Envelopment Analysis — DEA (DEA,SEQ=192). GAMS model. Obtained from the GAMS library call 'gamslib dea' (13 September 1999).

[14] W. A. Kamakura. A note on "The use of categorical variables in Data Envelopment Analysis". *Management Science*, 34(10):1273–1276, October 1988.

[15] L. Kuntz and S. Scholtes. Measuring the robustness of empirical efficiency valuations. *Management Science*, 46(6):807–823, June 2000.

[16] O. B. Olesen and N. C. Petersen. A presentation of GAMS for DEA. *Computers and Operations Research*, 23(4):323–339, 1996.

[17] H. Scheel. Data Envelopment Analysis (DEA) page. HTML document. http://www.wiso.uni-dortmund.de/lsfg/or/scheel/doordea.htm (23 October 2000).

[18] B. W. Silverman. *Density Estimation for Statistics and Data Analysis.* Chapman and Hall, London, 1986. pp. 52–53.

[19] L. Simar and P. W. Wilson. Sensitivity analysis of efficiency scores: How to bootstrap in nonparametric frontier models. *Management Science*, 44(1):49–61, January 1998.

[20] L. Simar and P. W. Wilson. A general methodology for bootstrapping in non-parametric frontier models. *Journal of Applied Statistics*, 27(6):779–802, 2000.

[21] L. Simar and P. W. Wilson. Statistical inference in nonparametric frontier models: The state of the art. *Journal of Productivity Analysis*, 13:49–78, 2000.

[22] J. B. Walden. Private communication.