On the Evolution of Optimization Modeling Systems

Robert Fourer

2010 Mathematics Subject Classification: 90-04
Keywords and Phrases: Optimization, mathematical programming, modeling languages, matrix generators

After a promising start in the 1950s, enthusiasm for the practical potential of linear programming systems seemed to fade. By the end of the 1970s it was not unusual to encounter sentiments of the following sort:

> We do not feel that the linear programming user's most pressing need over the next few years is for a new optimizer that runs twice as fast on a machine that costs half as much (although this will probably happen). Cost of optimization is just not the dominant barrier to LP model implementation. The process required to manage the data, formulate and build the model, report on and analyze the results costs far more, and is much more of a barrier to effective use of LP, than the cost/performance of the optimizer.
>
> Why aren't more larger models being run? It is not because they could not be useful; it is because we are not successful in using them ... They become unmanageable. LP technology has reached the point where anything that can be formulated and understood can be optimized at a relatively modest cost. [13]

This was written not by a frustrated user, but by the developers of an advanced LP system at one of the major computer manufacturers. Similar sentiments were expressed by others who were in a position to observe that the powerful techniques of computational optimization were not translating to powerful applications, at least not nearly as readily as expected.

Advanced software for optimization modeling was a response to this malaise and a key factor in bringing mathematical programming to a new period of enthusiasm. This article is intended as a brief introduction and history, particularly as reflected in writings by some of the pioneers and in my own early experiences. A detailed survey appears in [14], and extensive observations on the subject by many of the major participants have been collected in [11] and [12].

The history of optimization modeling systems can be viewed roughly as beginning with *matrix generators* and then expanding to *modeling languages*, and this account is organized accordingly. At the end I add a few reflections on more recent developments. In giving a historical account it is hard to avoid the use of "mathematical programming" to refer to what has since become more straightforwardly known as "optimization," and so these terms appear more-or-less interchangeably in my account. On the other hand "linear programming" or "LP" is still the term of choice of the special case of linear objectives and constraints.

Matrix generators

Almost as soon as computers were successfully used to solve linear programming problems, communication with the optimization algorithms became a bottleneck. A model in even a few kinds of variables and constraints, with perhaps a half-dozen modest tables of data, already gave rise to too many coefficients, right-hand sides, and bounds to manage by simply having a person enter them from a keyboard of some kind. Even if the time and effort could be found to key in all of these numbers, the process would not be fast or reliable enough to support extended development or deployment of models. Similar problems were encountered in examining and analyzing the results. Thus it was evident from the earliest days of large-scale optimization that computers would have to be used to create and manage problems as well as to solve them.

Because development focused initially on linear programming, and because the greatest work of setting up an LP is the entry of the matrix of coefficients, computer programs that manage optimization modeling projects became known as *matrix generators*. To make good use of computer resources, LP algorithms have always operated on only the nonzero coefficients, and so matrix generators also are concerned not with an explicit matrix but with a listing of its nonzero elements. The key observation that makes efficient matrix generators possible is that coefficients can be enumerated in an efficient way:

> Anyone who has been taught that linear programming is a way to solve problems such as Minimize Minimize $x_1 + 2x_2 + 4x_3 + x_4 + 3x_5$ ... may wonder how any computer program can help to assemble such a meaningless jumble of coefficients. The point is that practical linear programming problems are not like this. Although the range of problems to which mathematical programming is applied is very wide and is continuing to expand, it seems safe to claim that there is some coherent structure in all applications. Indeed, for a surprisingly wide class of applications the rows (or constraints) can be grouped into five categories and the columns (or variables) into three categories ... When a problem has been structured in this way, one can see how a computer program can be devised to fill in the details from a relatively compact set of input data. [1]

This explanation comes from Martin Beale's paper "Matrix Generators and Output Analyzers" in the proceedings of the 6th Mathematical Programming Symposium, held in 1967. Already at that point much had been learned about how best to write such programs. In particular Beale describes the practice of building short character strings to uniquely identify variables and constraints. These encoded names, typically 8 characters or less, were a central feature of the (nearly) standard MPS format adopted for the representation of linear programs.

A skilled programmer could get quite good at writing matrix generators. In the same article Beale states:

> I should like to dispel the illusion that a FORTRAN matrix generator is necessarily a very cumbersome affair by pointing out that I once wrote one before breakfast one Sunday morning. (Although it did contain one mistake which had to be corrected after going on the computer.)

The inclusion of such a disclaimer suggests that this activity did pose challenges to some modelers of optimization problems. In fact matrix generators are inherently difficult to write, and that difficulty derives most significantly from the challenges of debugging them. The following account describes procedures that persisted through much of the 1970s:

> ... the debugging process ... was basically the same one that had been used since the introduction of mathematical programming (MP) systems. When a model run was completed, the complete solution was printed along with a report. The output was examined to determine if the run passed the "laugh test", that is, no infeasibles and no "outrageous" values. If the laugh test failed, the solution print would be examined by paper clip indexing and manual paging. Frequently, the solution print was not enough to determine the problem and the matrix had to be printed. For large mathematical programs, the two printouts could be 6 inches thick. Nevertheless, the information needed to detect and correct the error took no more than a page. The trick was to know where to look and have facility with 6 inches of printout. [15]

This account, from a project at the U.S. Federal Energy Administration, suggests the kinds of difficulties that prompted the malaise described out the outset of this article. With computers becoming more powerful and attempts at optimization modeling becoming correspondingly more widespread and ambitious, the supply of sufficiently skilled debuggers — and debugging time — could not keep up.

A direct solution, pursued by the FEA project, was to get the computer to do some of the work of paging through the printout. This led to the development of progressively more sophisticated systems known as PERUSE and ANALYZE

[9] that worked with information from the 8-character names and searched for patterns associated with errors and infeasibility.

Another approach was based on making matrix generators more reliable. The essence of the debugging problem can be viewed as a gap between representations: a high-level, structured concept of the optimization problem, which is natural for human modelers to work with, is replaced by a computer program whose output is a list of coefficients in a form suitable for fast processing by a solver's algorithms. It is understandably hard for a human analyst to tell from looking at the coefficient list whether the program is running correctly, or why the results are wrong. So if the matrix generator can be written in a higher-level language that deals more directly with the concepts of LP formulation, then at least the chances of errors due to low-level programming bugs will be reduced. Indeed because such a program deals in terms closer to the modeler's original conception, one can expect that it will be easier to write, verify, maintain, and fix over the lifetime of the model.

The same proceedings in which Beale describes matrix generators programmed in a general-purpose language (Fortran) contain this abstract of a talk on a special-purpose matrix-generation language:

> The approach used in MaGen is based on a recognition that mathematical models consist of activities and constraints on these activities, and that both the activities and constraints can be grouped into classes. The generation of the matrix is carried out by FORM VECTOR statements under control of a DICTIONARY which defines the classes and provides mnemonic names for use in the model, and a Data section which provides the numerical information. [10]

Languages like MaGen, here described by its creator Larry Haverly, did much to structure the matrix generation process. They supported the small tables of data from which LPs were built, and incorporated intuitive syntactic forms for creation of unique 8-character names by concatenation of table row and column labels.

My own introduction to matrix generators was through one of these languages. In 1974 I joined the Computer Research Center set up in Cambridge, Massachusetts by the National Bureau of Economic Research (NBER). Although the center's focus was on statistical and data analysis software, it had recently brought in Bill Orchard-Hays to lead a development effort in the rather different area of linear programming. Orchard-Hays had taken the unusual (for the time) job of programmer at the RAND corporation in the early 1950s, shortly before George Dantzig's arrival gave impetus to an effort to program machines to do linear programming. Out of this collaboration came practical implementations of Dantzig's simplex method, initially on a card-programmed calculator and then on the first IBM scientific computer.

The early days of linear programming were an exciting time to be working with computers:

> mathematical programming and computing have been contemporary in an almost uniquely exact sense. Their histories parallel each other year by year in a remarkable way. Furthermore, mathematical programming simply could not have developed without computers. Although the converse is obviously not true, still linear programming was one of the important and demanding applications for computers from the outset. [17]

These comments are from a detailed retrospective article in which Orchard-Hays describes implementing a series of progressively more ambitious mathematical programming systems over a span of nearly three decades. By the time that our paths crossed, however, he had more the outlook of a former revolutionary, as this excerpt from the same article suggests:

> ... the nature of the computing industry, profession, and technology has by now been determined – all their essential features have existed for perhaps five years. One hopes that some of the more recent developments will be applied more widely and effectively but the technology that now exists is pretty much what will exist, leaving aside a few finishing touches to areas already well developed, such as minicomputers and networks.

This is perhaps a reminder that some fundamental aspects of computing and of optimization have hardly changed since that time, though in other respects today's environment is unimaginably different. The Mathematical Programming (now Mathematical Optimization) Society later fittingly named its prize in computational mathematical programming after Beale and Orchard-Hays.

I was fortunate to learn linear programming from Orchard-Hays's book [16] in which it was described how the simplex method was implemented for computers. Had I read one of the standard textbooks I would have learned a quite impractical version that was motivated by a need to assign little LPs for solution by hand. Among the components of the Orchard-Hays system that I encountered was a matrix generation and reporting language; working with two analysts at the U.S. Department of Transportation, I used it to develop a network LP application involving the assignment of railroad cars to a train schedule [6].

Modeling languages

The logical alternative to making matrix generation programs easier to debug was to make them unnecessary to write, by instead designing a kind of language that expressed the human modeler's formulation of an optimization problem directly to a computer system. The result was the concept of a *modeling language.*

Just as there are diverse ways to conceive of an optimization problem, there are potentially diverse designs for modeling languages. However for general-purpose modeling – not tied to any one application type or area – the one most widely implemented and used approach is based on the variables and equations familiar to any student of algebra and calculus. A generic optimization problem may be viewed as the minimization or maximization of some function of decision variables, subject to equations and inequalities involving those variables. So if you want to

$$\text{Minimize } \sum_{j=1}^{n} c_j x_j$$

where each x_j the quantity of one n of things to be bought, and c_j is its unit cost, then why not present it to the modeling software in a similar way, only using a standard computer character set? In the resulting *algebraic* modeling language, it could come out like this:

```
minimize TotalCost:  sum j in 1..n c[j] * x[j];
```

Of course for input to computer software one must be quite explicit, so additional statements are needed to declare that `n` and the `c[j]` are data values, while the `x[j]` are variables on an appropriate domain — since they represent things to buy, most likely nonnegative values or nonnegative integers.

Early, less ambitious modeling language designs called for linear expressions to be written in a simpler syntax, which might express an objective as

```
min 2.54 x1 + 3.37 x2 + 0.93 x3 + 7.71 x4 + 7.75 x5 + 2.26 x6 + ...
```

Although superficially this is also algebraic, it is no different in concept from the aforementioned MPS file or any listing of nonzero coefficients. What most importantly distinguishes the previous description of `TotalCost` is that it's symbolic, in that it uses mathematical symbols to describe a general form of objective independently of the actual data. Whether `n` is 7 or a 7 thousand or 7 million, the expression for `TotalCost` is written the same way; its description in the modeling language does not become thousands or millions of lines long, even as the corresponding data file becomes quite large.

The same ideas apply to constraints, except that they express equality or inequality of two algebraic expressions. So if in another model one wants to state that

$$\sum_{p \in P} (1/a_{ps}) y_p \leq b_s \quad \text{for all} \quad s \in S$$

it could be written, after some renaming of sets, parameters, and variables to make their meanings clearer, as

```
subject to Time {s in STAGE}:
    sum {p in PROD} (1/rate[p,s]) * Make[p] <= avail[s];
```

Constraints usually occur in indexed collections as in this case, rather than individually as in our example of an objective. Thus the advantage of a symbolic description is even greater, as depending on the data one constraint description can represent any number of constraints, as well as any number of coefficients within each constraint.

A well-written matrix generator also has the property of data independence, but the advantages of modeling languages extend further. Most important, a modeling language is significantly closer to the human analyst's original conception of the model, and further from the detailed mechanisms of coefficient generation:

> Model building in a strategic planning environment is a dynamic process, where models are used as a way to unravel the complex real-world situation of interest. This implies not only that a model builder must be able to develop and modify models continuously in a convenient manner, but, more importantly, that a model builder must be able to express all the relevant structural and partitioning information contained in the model in a convenient short-hand notation. We strongly believe that one can only accomplish this by adhering to the rigorous and scientific notation of algebra. ... With a well-specified algebraic syntax, any mode representation can be understood by both humans and machines. The machine can make all the required syntactical and semantic checks to guarantee a complete and algebraically correct model. At the same time, humans with a basic knowledge of algebra can use it as the complete documentation of their model. [2]

This introduction by Bisschop and Meeraus to the GAMS modeling language reflects a development effort begun in the 1970s, and so dates to the same period as the quote that led off this article. Although its focus is on the needs of optimization applications that the authors encountered in their work at the World Bank, its arguments are applicable to optimization projects more generally.

I also first encountered modeling languages in the 1970s, while working at NBER. I do not recall how they first came to my attention, but as the Computer Research Center's mission was the design and development of innovative modeling software, ideas for new languages and tools were continually under discussion; naturally the younger members of the linear programming team began to consider those ideas in the context of LP software:

> Popular computer packages for linear programming do not differ much in concept from ones devised ten or twenty years ago. We propose a modern LP system – one that takes advantage of such (relatively) new ideas as high-level languages, interactive and virtual operating systems, modular design, and hierarchical file systems.

> Particular topics include: computer languages that describe optimization models algebraically; specialized editors for models and data; modular algorithmic codes; and interactive result reporters. We present specific designs that incorporate these features, and discuss their likely advantages (over current systems) to both research and practical model-building. [7]

This was the abstract to a report on "A Modern Approach to Computer Systems for Linear Programming," which I had begun writing with Michael J. Harrison by the time that I left for graduate school in 1976. Algebraic modeling languages played a prominent role in our proposals, and an example from a prototype language design was included.

"A Modern Approach ..." was completed at NBER's Stanford office and appeared in the M.I.T. Sloan School's working paper series. After completing my PhD studies at Stanford and moving to Northwestern, an attempt to submit it for publication made clear that some of its central assertions were considerably less obvious to others than they had been to me. In particular we had started off the description of our modeling language by stating that,

> Models are first written, and usually are best understood, in algebraic form. Ideally, then, an LP system would read the modeler's algebraic formulation directly, would interpret it, and would then generate the appropriate matrix.

Reviewers' reactions to this claim suggested that there were plenty of adherents to the traditional ways of mathematical programming, who would settle for nothing less than a thorough justification. Thus I came to write a different paper, focused on modeling languages, which investigated in detail the differences between modeler's and algorithm's form, the resulting inherent difficulties of debugging a matrix generator, and many related issues. Additionally, to confirm the practicality of the concept, I collected references to 13 modeling language implementations, with detailed comparisons of the 7 that were sophisticated enough to offer indexed summations and collections of constraints. Most have been forgotten, but they did include GAMS, which remains one of the leading commercial modeling language systems, and LINDO, which gave rise to another successful optimization modeling company.

The publication of this work as "Modeling Languages versus Matrix Generators" [3] was still not an easy matter. As I recall it was opposed by one referee initially and by the other referee after its revision, but never by both at the same time ... and so a sympathetic editor was able to recommend it, and after a further examination the editor-in-chief concurred. It appeared in a computer science journal devoted to mathematical software, which at the time seemed a better fit than the journals on operations research and management science.

Subsequently a chance encounter led to my greatest adventure in modeling languages. I had known Dave Gay when he was an optimization researcher at NBER, but by the time we met at the 1984 TIMS/ORSA conference in San Francisco he had moved to the Computing Sciences Research Center at Bell Laboratories. The Center's researchers had developed Unix and the C programming language among many innovations, and were given a free hand in initiating new projects. Dave graciously invited me to spend a sabbatical year there without any particular commitments, and as it happened my arrival coincided with the completion of Brian Kernighan's latest computer language project. A fresh attempt at designing an algebraic modeling language seemed like a great fit for the three of us.

Thus did AMPL get its start. We aimed to make it a declarative modeling language in a rigorous way, so that the definition of a variable, objective, or constraint told you everything you needed to know about it. In a constraint such as Time above, you could assign or re-assign any parameter like `rate[p,s]` or `avail[s]`, or even a set like `STAGE`, and the resulting optimization problem would change implicitly. A lot of our initial work went into the design of the set and indexing expressions, to make them resemble their mathematical counterparts and to allow expressions of full generality to appear anywhere in a statement where they logically made sense.

The naming of software was taken very seriously at Bell Labs, so the choice of AMPL, from A Mathematical Programming Language (with a nod to APL), came well after the project had begun. By the late 1980s the concept of modeling languages had become much more established and a paper on AMPL's design [4] was welcomed by *Management Science.* The referees did object that our reported times to translate sophisticated models were often nearly as great as the times to solve them, but by the time their reports came in, the translator logic had been rewritten and the times were faster by an order of magnitude.

AMPL had a long gestation period, being fundamentally a research project with a few interested users for its first seven years. Bell Labs provided an ideal environment for innovation but not a clear path for disseminating the resulting software. There was a strong tradition of disseminating written work, however, so we proposed to write an AMPL book [5] that happened to have a disk in the back. It started with a tutorial chapter introducing a basic model type and corresponding language forms, which expanded to a four-chapter tutorial covering a greater range of model types and language features. At that point there seemed no good reason to abandon the tutorial approach, and subsequent chapters eventually introduced all of the more advanced features using progressively more advanced versions of the same examples. This approach paid off in popularizing the modeling language approach beyond what a straightforward user's manual could have done.

The AMPL book's design was commissioned by the publisher as part of a projected series in which volumes on different software systems would be associated with different animals, but beyond that we have no specific explanation for the cat that appears on the cover.

Reflections

Algebraic modeling languages have long since become an established approach rather than a "modern" departure. Four general-purpose languages – AIMMS, AMPL, GAMS, MPL – and their associated software have been in active development for two decades or more, each by a small company devoted to optimization. The similarity of their names notwithstanding, the stories of how these language came about are all quite different; and although based on the same underlying concept, they differ significantly in how the concept is presented to users. Moreover a comparable variety of algebraic modeling languages have developed for dedicated use with particular solvers.

Freedom from programming the generation of matrix coefficients has indeed proved to be a powerful encouragement to applied optimization. Modeling languages have lowered the barrier to getting started, particularly as the population of technically trained computer users has expended far beyond the community of practiced programmers. Applications of optimization models have spread throughout engineering, science, management, and economics, reflected in hundreds of citations annually in the technical literature.

Modeling languages' general algebraic orientation also has the advantage of allowing them to express nonlinear relations as easily as linear ones. The benefits of avoiding programming are particularly great in working with nonlinear solvers that require function values and derivative evaluations, which modeling language systems can determine reliably straight from the algebraic descriptions. In fact the advent of efficiently and automatically computed second derivatives (beginning with [8]) was a significant factor in advancing nonlinear solver design.

And what of matrix generators? They have by no means disappeared, and will surely maintain a place in optimization modeling as long as there are talented programmers. They have particular advantages for tight integration of solver routines into business systems and advanced algorithmic schemes. And modeling languages have greatly influenced the practice of matrix generation as well, with the help of object-oriented programming. Through the creation of new object types and the overloading of familiar operators, it has become possible to use a general programming language in a way that looks and feels a lot more like a modeling language declaration. Even the symbolic nature of a model can be preserved to some degree. Thus the process of creating and maintaining a generator can be made more natural and reliable, though difficulties of disentangling low-level programming bugs from higher-level modeling errors are still a powerful concern.

Whatever the choice of language, it seems clear that developments over four decades have realized much of the vision of letting people communicate optimization problems to computer systems in the same way that people imagine and describe optimization problems, while computers handle the translation to and from the forms that algorithms require. And still, anyone who has provided support to modeling language users is aware that the vision has not been entirely realized, and that modelers even now need to do a certain amount of translating from how they think of constraints to how modeling languages are prepared to accept them. Replies that begin, "First define some additional zero-one variables ...", or "You could make the quadratic function convex if ...", remain all too common; the conversions implied by these statements have been addressed to some extent in some designs, but not yet in a truly thorough manner applicable both to a broad range of models and a variety of solvers.

In conclusion it is reasonable to say that optimization modeling is considered challenging today just as it was in the 1970s, but that he experience of creating an application has changed for the better. Just as in the case of solver software, improvements in modeling software have occurred partly because computers have become more powerful, but equally because software has become more ambitious and sophisticated. The malaise of earlier times seems much less evident, and there is arguably a better balance between what can be formulated and understood and what can be optimized.

References

[1] E.M.L. Beale, Matrix generators and output analyzers, in: Harold W. Kuhn (ed.), *Proceedings of the Princeton Symposium on Mathematical Programming*, Princeton University Press, 1970, pp. 25–36.

[2] J. Bisschop and A. Meeraus, On the development of a general algebraic modeling system in a strategic planning environment, *Mathematical Programming Studies* 20 (1982) 1–29.

[3] R. Fourer, Modeling languages versus matrix generators for linear programming, *ACM Transactions on Mathematical Software* 9 (1983) 143–183.

[4] R. Fourer, D.M. Gay and B.W. Kernighan, A modeling language for mathematical programming, *Management Science* 36 (1990) 519–554.

[5] R. Fourer, D.M. Gay and B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, 1993.

[6] R. Fourer, J.B. Gertler and H.J. Simkowitz, Models of railroad passenger-car requirements in the northeast corridor, *Annals of Economic and Social Measurement* 6 (1977) 367–398.

[7] R. Fourer and M.J. Harrison, A modern approach to computer systems for linear programming, Working paper 988-78, Sloan School of Management, Massachusetts Institute of Technology (1978).

[8] D.M. Gay, More AD of nonlinear AMPL models: Computing hessian information and exploiting partial separability, in: M. Berz, C. Bischof, G. Corliss and A. Griewank (eds.), *Computational Differentiation: Techniques, Applications, and Tools,* SIAM, 1996, pp. 173–184.

[9] H. Greenberg, A functional description of ANALYZE: A computer-assisted analysis system for linear programming models, *ACM Transactions on Mathematical Software* 9 (1983) 18–56.

[10] C.A. Haverly, MaGen II, in: Harold W. Kuhn (ed.), *Proceedings of the Princeton Symposium on Mathematical Programming,* Princeton University Press, 1970, pp. 600–601.

[11] J. Kallrath (ed.), *Modeling Languages in Mathematical Optimization,* Kluwer Academic Publishers, 2004.

[12] J. Kallrath (ed.), *Algebraic Modeling Systems: Modeling and Solving Real World Optimization Problems,* Springer, 2012.

[13] C.B. Krabek, R.J. Sjoquist and D.C. Sommer, The APEX systems: Past and future, *SIGMAP Bulletin* 29 (1980) 3–23.

[14] C.A.C. Kuip, Algebraic languages for mathematical programming, *European Journal of Operational Research* 67 (1993) 25–51.

[15] W.G. Kurator and R.P. O'Neill, PERUSE: An interactive system for mathematical programs, *ACM Transactions on Mathematical Software* 6 (1980) 489–509.

[16] W. Orchard-Hays, *Advanced Linear-Programming Computing Techniques,* McGraw-Hill, 1968.

[17] W. Orchard-Hays, History of mathematical programming systems, in: H.J. Greenberg (ed.), *Design and Implementation of Optimization Software,* Sijthoff and Noordhoff, 1978, pp. 1–102.

Robert Fourer
Northwestern University
2145 Sheridan Road
Evanston, IL 60208-3119
USA
`4er@northwestern.edu`