Intelligent Mathematical Programming Software: Past, Present, and Future*

(* This article appears concurrently in the Spring issue of the INFORMS Computing Society Newsletter)

JOHN W. CHINNECK
Systems and Computer Engineering
Carleton University
Ottawa, Ontario K1S 5B6
Canada
chinneck@sce.carleton.ca
http://www.sce.carleton.ca/faculty/chinneck.html

HARVEY J. GREENBERG
Mathematics Department
University of Colorado at Denver
Denver, Colorado 80217-3364
U.S.A.
hgreenbe@carbon.cudenver.edu
http://www.cudenver.edu/~hgreenbe/

Introduction

The practice of mathematical programming has been profoundly affected in recent years by two key trends: the proliferation of powerful and inexpensive computing platforms, and greatly improved solution algorithms. These trends allow experienced mathematical programmers to solve models of much greater scale and complexity than previously possible. But at the same time, these trends have introduced a host of novices to mathematical programming, notably by the solvers embedded in popular spreadsheet software.

From expert to novice, the bottleneck in the mathematical programming process is often not the numerical computation of a solution. Instead the critical tasks are creating, analyzing, understanding, and communicating models and instances, with clear implications for decision support. Some form of automated assistance is desirable, even necessary, to handle the scale and complexity of modern mathematical programs. This realization has spurred the development of a foundation for an *Intelligent Mathematical Programming System* (IMPS) [9] over the last two decades.

Formally, we define an IMPS as software that reduces the complexity inherent in the *mathematical programming process*. We define this process more precisely in the next section as a breakdown of the tasks one performs to build, use and maintain a decision-support system whose core is a mathematical program. Our definition of an IMPS is a bit broader than the usual definition given for computer intelligence in that we include aids that do not necessarily involve reasoning; an important example is our inclusion of visualization aids.

The development of software tools supporting the process of mathematical programming has been an active area of research over many years, particularly in recent times. A recent bibliography [13] gives more than 500 citations referring to some aspect of an IMPS over the period 1953-1996, of which more than 80 pertain specifically to software.

Our goal in this article is to summarize the kinds of tasks that might be undertaken by an ideal IMPS, as suggested by the work undertaken over the past two decades. We also comment briefly on the current state of the art. We are in the process of compiling a survey of existing IMPS software – if we have not already contacted you, you are invited to contribute a description of software that we might otherwise miss (see Appendix). To summarize briefly, there is a gratifying volume of IMPS software, yet there remain numerous parts of the modeling process that are in need of suitable tools. This is a fertile area for research.

1. The Mathematical Programming Process

Figure 1 is a simplified diagram of the main objects in the mathematical programming process. A *mathematical model* is extracted from the *problem domain* by making appropriate approximations of relationships (objectives and constraints). As data is collected, various *instances* of the model, which differ according to the specific data used, can be formulated and solved. The solver results for various instances can be collected as *case studies*. The loop is closed when the case study results are compared to the problem domain. Unexpected or erroneous results, or other questions raised by the case study results, may lead to modifications of the mathematical model or the data, and the modeling/analysis cycle repeats.

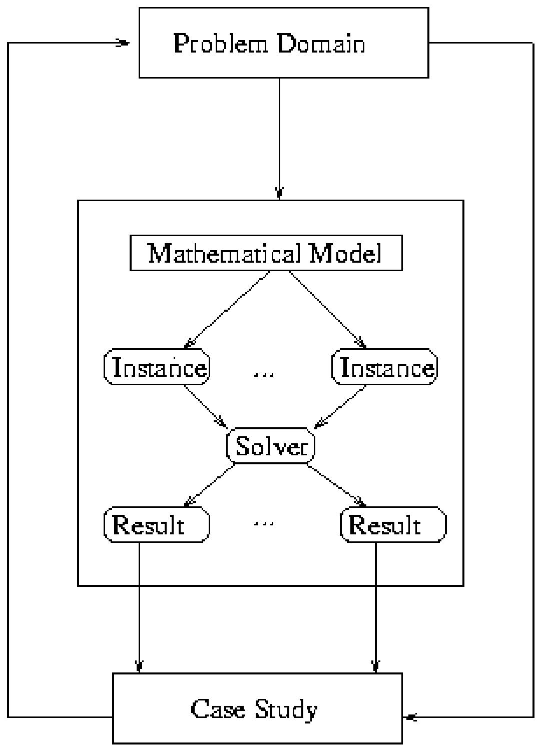

Figure 1. **A Simplified View of the Mathematical Programming Process**

The process in Figure 1 is idealized. Many modeling projects are neither top-down (i.e. starting with model structures and progressing towards data acquisition) nor bottom-up (i.e. starting with a rich database and inferring appropriate relationships). The flow is often middle-out, or hopscotch.

The mathematical programming process itself can be viewed as a cycle of functions applied to the objects in Figure 1. The following describes the seven main functions in the process, all of which are involved in any modeling project, and all of which can benefit from automated assistance.

Expressing the Model. Especially for large and complex models, it is vital to have a convenient way of expressing the model such that errors and omissions can be detected. Some algebraic modeling languages and graphical interfaces are in this category. There are two related functions: *documentation*, or recording a description for others to know the model, and *verification*, or

ensuring that what you think is in the computer-resident model is actually there.

Viewing the Model and Instances. Different views of the model and its instances include algebraic, schematic, graphical, and natural language. Individual modelers may find some views more natural, assisting their understanding of the model or solution. Even the same individual may want different views for insights, so an IMPS must be able to accommodate a broad range of *cognitive differences*. *Reporting* is a related function that may include interactive query, generating internal files for later analysis, or creating overhead slides for presentation.

Simplifying the Model. Automatic simplification of models to improve solver performance is well established, for example in the pre-solve routines in LP solvers of commercial quality. Perhaps more important is the idea of simplification to improve human understanding, for example automatic rewriting of modeling language source code to give a simpler model expression, such as finding an underlying netform.

Debugging. This is the process of removing the mechanical errors in an instance, and perhaps in the model itself. Correcting models that are mechanically correct, but that are faulty representations of reality, is the province of the general analysis tools, treated separately. Various kinds of errors are common in mathematical programs, sometimes leading to infeasibility, unboundedness, and non-viability. Debugging the model can be approached via debugging of model instances, or can be approached directly at a symbolic level.

Data Management. A large volume of data is collected and generated in many mathematical programming processes. Database techniques are commonly used to manage the information, and may lend themselves in the future to the application of related technologies, such as data mining. We make only limited observations in this category because the subject is vast and merits special attention.

Scenario Management. Typically, many instances of a model are solved, each representing a *scenario*. A *case study* is a collection of related scenarios, and cases could overlap (i.e. have some of the same scenarios). The large volume of information generated in this way must be processed and filtered so that important conclusions are readily apparent to the analyst.

General Analysis. There are numerous general questions about any mathematical model. For example: Are there specific relationships among the data? What are the nonlinear functional shapes? What drives the price of a certain variable? General analysis tools help in answering such questions by providing the means to probe the model in various ways. *Validation*, or determining how well the model represents reality, is a related function. For example, is a linear programming model an adequate representation of the system, given the decisions that it is designed to support? As another example, consider *redundancy analysis:* is the constraint redundant because other constraints are more restrictive, or due to economic preference?

The environment in which the modeling project is carried out can add to the complexity, thereby increasing the need for automated assistance. For example, a model may be eclectic, with different people responsible for different parts of the model (e.g. the National Energy Modeling System [20]). It is also typical that a model will be applied or modified by people other than those who built it. Intelligent aids are indispensable in both cases.

2. How Software Can Assist: Past, Present and Future

Past IMPS software shows clearly that the greatest challenge in developing systems that will last over the long term is in coping with the undermining effect of rapid changes in software technology. Our survey so far indicates that many systems are no longer active because of an implementation decision tying the software to a development system that no longer exists or that would require ongoing heavy maintenance to keep up with the host platform. Systems built to run on many platforms tend to survive provided they continue to offer competitive capabilities.

The shape and capabilities of future IMPSs are inextricably linked with developments in computing, both hardware and software. The progression to faster and better user interfaces offers great opportunities for research towards better exploitation of that technology for improved IMPSs. The explosive growth of the World Wide Web is already impacting the teaching of mathematical programming; for example, Java codes that enable interactive computing in real time are available. We can expect further collaborative development and solution of immense models via the web. These will require the creation of intelligent assistants to help manage the process.

We now briefly summarize some of the specific ways in which the functions in the mathematical programming process are currently assisted by software.

Expressing and Viewing Models. There are numerous ways to express complex mathematical programs, including algebraic modeling languages, constraint logic programming languages, graphical interfaces, and spreadsheets. See [15] for a survey.

The expression of a model must be a complete description that captures sufficient relevant details for decision support. The idea of views is broader, and includes any representation that provides a useful insight, including views that suppress or omit details. Model expression and viewing are part of the *discourse* component [12] of an IMPS (i.e. how we and the computer communicate information about the model and its instances).

In discourse, language styles may differ, such as algebraic or procedural. *Network modeling languages* can use a graphical user interface or can be a specialized variant of algebraic modeling languages (e.g. Proflow [5]). *Constraint logic programming* languages offer yet another way to express, as well as to view, logical relations. Some systems can provide a more natural language discourse in support of model analysis.

One tabular expression is a *block schematic*, which represents the model with special cell syntax [1,15,18]. We also use tables to view data with different *slices* and *aggregations*, common in most algebraic languages. Process views can be constructed as a combination of these [1]. S*preadsheets* are a special kind of tabular discourse, where cells contain model logic and data intermixed. *Graphics* can be iconic or diagrammatic. *Matrix graphics* can provide useful views, for example a very large matrix can be displayed with a single pixel lighted for any cell having a nonzero entry, or positive entries can be shown by a + sign and negative entries by a – sign. See Greenberg and Murphy [14] for a general analysis of structures and a survey of systems in which graphics and matrix graphics are used.

Some views are not standard in mathematical programming, but stem from database concepts, such as *dependency relations* – e.g., which functions and variables depend on a particular set. The MODLER [11] system provides views of this type.

In the ideal IMPS, the modeler should have a choice of ways of expressing the model, and should be able to switch easily between a wide variety of views as needed. The vast majority of current

systems provide a single means of expressing the model, and usually no other views. MIMI [1] is perhaps the most sophisticated commercial system, allowing several modes of expressing and viewing the model. MProbe [7] also offers various tabular views as well as profile plots of nonlinear functions of many dimensions, and an algebraic view via a link to AMPL. MODLER [11] provides multiple views, but it requires algebraic input; ANALYZE [10] offers the ability to choose between various tabular, network graphic, matrix, and algebraic views.

There is a need for basic research on views that provide insight for model forms beyond linear. One promising avenue for research is animation; see the pioneering work by Jones [17].

Simplifying Models. Most commercial solvers, and some modeling languages, include a pre-solve capability, but they are designed as "black boxes" to speed up optimization, not as analysis tools. However, there are a few tools that reveal simplifications of the model aimed at improving human understanding. MProbe [7] allows the examination of nonlinear functions of many dimensions to determine whether they are candidates for simplification (e.g. linearization). ANALYZE [10] allows exploration of the model for simplifications in LPs and MILPs. GAMSCHK [19] reads GAMS language input and gives a report on ways in which the model can be simplified. Redundancy analysis, via a general analysis tool such as ANALYZE, can also be used to simplify a model, though few practitioners understand how and why this is done.

Debugging Models. Classical software provides for the detection of non-optimal states, but not the underlying errors. Modern software provides assistance in diagnosing a cause with suggestions for remedy and prevention.

Infeasibility debugging has a long history. Methods of successive bound tightening have been included in LP pre-solve routines for many years, and are sometimes capable of detecting infeasibility and providing useful feedback through the traceback of the sequence of reductions (see the REDUCE command in ANALYZE, for example). However, in recent years most large commercial LP solvers have moved to the isolation of an *Irreducible Infeasible Subsystem* (IIS) [4,6,0]. An IIS is often a tiny fraction of the constraints in the model and has the property that it is infeasible, but any proper subset is feasible; thus every constraint in the IIS contributes to that infeasibility. Finding an IIS helps to focus the diagnostic effort. See Chinneck [6] for a recent survey of the state of the art in infeasibility analysis for all forms of mathematical programs.

Unboundedness debugging can theoretically be done by applying infeasibility debugging to the dual (at least in LP, and with some caveats in NLP and MIP), but examples show this is not effective. Underlying causes of unboundedness can be very different, such as a negative cycle in a network model. Such causes might be better sought directly, rather than rely on a technique aimed at primal infeasibility.

Nonviability is a property of a network model in which at least one arc flow variable is forced to zero by the structure of the network model (not by upper bounds on the arc flows); this can happen in all forms of network models, including *processing networks* [2]. Because it is unlikely that the modeler intends an arc that cannot support flow, such a mechanical error should be flagged prior to model solution. Chinneck [2,3] has developed methods for identifying and isolating nonviability, and these have been implemented in academic prototype software. It is interesting to note that any general LP can be viewed as a process network from its activity I/O structure [14].

A generalization of nonviability refers to any variable that is forced to a fixed value by the constraints in the model. Given this interpretation, methods and software remain to be developed for all non-network forms of mathematical programs.

General Analysis. General analysis software is responsible for providing a suite of tools for probing the model in various ways to answer questions and provide insights. As examples, consider two tools created by the authors.

ANALYZE provides computer assistance for analyzing linear mathematical programs and their solutions. There are three levels of use. The lowest level provides convenient interactive query and, combined with MODLER, enables syntax-directed report writing [8]. The second level provides procedures to assist analysis in a variety of ways. Standard sensitivity questions, such as *What if …?, Why …?* and *Why not …?*, can be answered with easy access to information about the solution. In addition, diagnostic analysis, such as debugging an infeasibility, can be resolved efficiently either by internal routines or links with external information, such as an IIS obtained from MINOS(IIS) [4]. The third level is an artificially intelligent environment that includes translations of objects into English and rule-based reasoning.

MProbe provides estimates of nonlinear function shape (e.g. convex, concave or both), the extent of the shape, function range, slope analog, plotting of a function between two arbitrary points in n-space, model navigation, and estimates of constraint effectiveness. It also provides estimates of the shape of the constrained region (convex or nonconvex) and the likelihood of finding a global optimum (based on the shape and sense of the objective function, and the shape of the constrained region). User-defined sorting can be used to isolate classes of constraints that are likely to cause MILP difficulties.

Data Management. Many modeling languages allow direct connection of the mathematical model with common or proprietary database formats. Access to large databases has been an important aspect of mathematical programming systems since their inception. Interfacing with spreadsheets is a more recent development, proving to be just as important, though for different users.

Scenario Management. For generations, this has been done by model managers, using control tables in a way that made the management process very difficult. MathPro [18] is an example of a system built specifically to improve scenario management. Cross-scenario analysis is currently a human endeavor with minimal computer assistance. It is an open question how to use information from a set of scenarios advantageously when solving (or specifying) a new scenario. The IMPS responsibility is to provide tools to enable such analysis. The tool could be as simple as a link to some other module, like a statistics package.

3. Conclusions

The process of mathematical programming can be quite complex, and it is becoming even more so in step with advances in computing hardware and software and improved algorithms. We are in an era that is reminiscent of the early days of general computer programming, when there was little assistance for the programmer, who was forced to use a simple text editor to create programs, and was forced to pore through core dumps to debug errors. Now the general programmer has a host of computer assistants: code editors that highlight reserved words, flag errors as they are typed, and can jump to variable definitions and show calling hierarchies, on-line documentation, debuggers, profilers, etc. Mathematical programming is at an early stage of developing its own specialized tools to assist in the process of creating, exploring, and maintaining useful models. Actually solving model instances is no longer the major barrier to effective mathematical programming; the ability to cope with the inherent complexity is. Intelligent mathematical programming systems aim to break through this barrier.

APPENDIX: ADDING TO THE SOFTWARE SURVEY

If you have knowledge of software that exhibits some aspect of an IMPS system, please contact one of the authors to provide the following information:

- Name of software system.
- Contact names and addresses for the developers or informants (including email).
- Sources of information about the software: published papers, technical reports, books, web sites, etc.
- Comments on how the software contributes to each of the seven functions in the mathematical programming process.

References

1. Chesapeake Decision Sciences, *MIMI User Manual*, New Providence, NJ, 1988.
2. J.W. Chinneck, Formulating Processing Networks: Viability Theory, *Naval Research Logistics* 37 (1990), 245-261.
3. J.W. Chinneck, Viability Analysis: A Formulation Aid for All Classes of Network Models*, Naval Research Logistics* 39 (1992), 531-543.
4. J.W. Chinneck, MINOS(IIS): Infeasibility Analysis Using MINOS*, Computers & Operations Research* 21:1 (1994), 1-9. Software available at http://www.sce.carleton.ca/faculty/chinneck/minosiis.html.
5. J.W. Chinneck, Processing Network Models of Energy/Environment Systems, *Computers and Industrial Engineering* 28:1 (1995), 179-189.
6. J.W. Chinneck, Chapter 14: Feasibility and Viability, in *Advances in Sensitivity Analysis and Parametric Programming*, T. Gal and H.J. Greenberg (eds.), International Series in Operations Research and Management Science, vol. 6, Kluwer Academic Publishers, Boston, MA, 1997.
7. J.W. Chinneck, *Analyzing Mathematical Programs Using MProbe*, Technical Report SCE-98-03, Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1998. Software available at http://www.sce.carleton.ca/faculty/chinneck/MProbe.html.
8. H.J. Greenberg, Syntax-Directed Report Writing in Linear Programming*, European Journal of Operations Research* 72:2 (1994), 300-311.
9. H.J. Greenberg, An Industrial Consortium to Sponsor the Development of an Intelligent Mathematical Programming System, *Interfaces* 20:6 (1991), 88-93.
10. H.J. Greenberg, A Computer-Assisted Analysis System for Mathematical Programming Models and Solutions: A User's Guide for ANALYZE, Kluwer Academic Press, Boston, MA, 1993. DOS and linux versions are available at http://www.cudenver.edu/~hgreenbe/imps/software.html.
11. H.J. Greenberg*, Modeling by Object-Driven Linear Elemental Relations: A User's Guide for MODLER*, Kluwer Academic Press, Boston, MA, 1993. DOS and linux versions are available at http://www.cudenver.edu/~hgreenbe/imps/software.html.
12. H.J. Greenberg, Intelligent User Interfaces for Mathematical Programming, Proceedings of the Shell Conference: *Logistics: Where Ends have to Meet*, C. Van Rijgn (ed.), Pergamon Press, 1989, 198-223.
13. H.J. Greenberg, A Bibliography for the Development of an Intelligent Mathematical Programming System, *Annals of Operations Research* 65 (1996), 55-90. A 1997 version is on the World Wide Web, http://www.cudenver.edu/~hgreenbe/imps/impsbib/impsbib.html.
14. H.J. Greenberg and F.H. Murphy, Views of Mathematical Programming Models and Their Instances, *Decision Support Systems* 13:1 (1995), 3-34.
15. H.J. Greenberg and F.H. Murphy, A Comparison of Mathematical Programming Modeling Systems, *Annals of Operations Research* 38 (1992), 177-238.

16. H.J. Greenberg and F.H. Murphy, Approaches to Diagnosing Infeasibility for Linear Programs, *ORSA Journal on Computing* 3:3 (1991), 253-261.
17. C.V. Jones, *Visualization in Optimization*, Kluwer Academic Press, Boston, MA, 1995.
18. MathPro, Inc., *MathPro Usage Guide: Introduction and Reference*, Washington, DC, 1989.
19. B.A. McCarl, *GAMSCHK User Documentation*, Department of Agricultural Economics, Texas A&M University, College Station, TX, 1998. Software available at http://agrinet.tamu.edu/mccarl/.
20. *National Energy Modeling System Integrating Module Documentation Report*, DOE/EIA-MO57(95), Energy Information Administration, Washington, DC, 1995. This, and related documents, are available at http://www.eia.doe.gov/bookshelf/docs.html.