Debugging Mathematical Programming Models: Principles and Practical Strategies

Article *in* Review of Marketing & Agricultural Economics · February 1996

Source: RePEc

CITATIONS
11

READS
62

3 authors:

David J. Pannell
University of Western Australia
330 PUBLICATIONS **11,013** CITATIONS
SEE PROFILE

Ross Kingwell
University of Western Australia
150 PUBLICATIONS **2,186** CITATIONS
SEE PROFILE

Steven Schilizzi
University of Western Australia
146 PUBLICATIONS **1,687** CITATIONS
SEE PROFILE

Debugging Mathematical Programming Models: Principles and Practical Strategies

David J. Pannell, Ross S. Kingwell and Steven Schilizzi*

Bugs are an unavoidable aspect of mathematical programming (MP) modelling. In this paper we discuss the prevention and diagnosis of bugs in MP models. The topic is rarely addressed in the literature but is crucial to the success of modelling projects, especially for large models. We argue that finding a bug and understanding unexpected results (whether or not due to a bug) are very closely related activities. We identify different types of bugs and suggest practical strategies for dealing with each. Adopting procedures for prevention of bugs is essential, especially for large models. We outline the prevention strategies we have adopted and found successful for the MIDAS and MUDAS models.

1. Introduction

Validation of mathematical programming (MP) and other types of models has been discussed by a number of authors (e.g. Gass; McCarl). Debugging is one component of the validation process but it has generally been given a cursory treatment in the validation literature. For example, McCarl's "procedure for model validation" includes the following: "If the model has failed, discover why. ... Repair the model and go to step 2" (p.161). Readers who have constructed large MP models will know how difficult and time consuming it can be to obey these simple instructions.

Debugging has also been neglected in the many texts which deal with construction and solution of MP models. We hope, in this paper, to counter some of this neglect. Our ideas are based on experience over the past decade developing, using and actively maintaining the MIDAS whole-farm model (e.g. Morrison *et al.* 1986; Kingwell and Pannell, 1987) and more recently MUDAS, a much larger version including seasonal variation (Kingwell *et al.* 1992). Our aims in this paper are: to identify some principles of bugs and debugging, to discuss some implications of these principles, to identify types of bugs and their symptoms, and to create checklists of strategies for (a) debugging and (b) preventing bugs.

2. Guiding Principles

In this section we identify and discuss some principles of bugs and debugging which are relevant to diagnosing, curing and preventing bugs. Our principles are:

(a) Prevention is better than cure.

This is an old but apt adage which applies as much in debugging MP modelling as anywhere. We discuss prevention of bugs at some length later.

(b) An unexpected model result is due to a bug unless you can convince yourself otherwise.

This implies a conservative approach to interpreting model results. It means that you do not accept a plausible explanation for a result without examining and testing the alternative explanation that the result is at least partly due to a bug. In our experience, it is easy to invent a more or less convincing explanation for almost any result in a complex model. Such explanations should not be accepted uncritically. On the other hand, you can't prove there is no bug (in much the same way as in the dominant paradigm for science it is possible to disprove a hypothesis, but not to prove it. See Magee). In the end it comes down to the modeller taking "reasonable" care, a matter of subjective judgement.

(c) Maintaining a bug-free (or at least low-bug) model requires discipline.

Most importantly, the modeller needs discipline to pursue every unexpected result or suspicious looking matrix coefficient to the point where you are convinced that it is or isn't due to a bug. It's all too easy to go on to the more interesting task of applying the model. Discipline is also required to keep model documentation up to date. This leads to:

* Agricultural and Resource Economics, University of Western Australia, Nedlands 6907.

Review coordinated by K. Parton and the Editor.

(d) A well documented model is easier to debug and maintain.

This is possibly the most obvious but also the most frequently ignored of our principles.

(e) Knowledge of the model is essential for debugging.

Knowledge of the model's assumptions and structure is essential for uncovering some bugs. Further, detailed knowledge of the real-world system being modelled is also useful, particularly for recognising model runs which are symptomatic of a bug.

(f) In a well maintained model, the number of bugs in a matrix decreases over time as they are discovered and fixed, but not to zero.

After a long period of model use and maintenance, any remaining bugs are unlikely to be serious, or at least to conflict with expectations. Such bugs can remain undetected for a long time. For example, we recently found 15 wrong coefficients in the EWM version of MIDAS which had been there for several years. MIDAS is an intensively maintained and used MP model and it is not especially large as MP matrices go, yet it still harboured at least 15 undiscovered bugs for several years. Fortunately they weren't serious.

Long-standing bugs like these will generally only be discovered if the model is used in a new and innovative way, model input or output is examined in different ways or a new person joins the modelling team. The 15 bugs mentioned above were discovered when a new format for output was created. However these new approaches to model use rapidly dissipate their capacity for revealing bugs as they too become routine.

(g) The number of bugs in a matrix increases rapidly with the size of the matrix.

For example, suppose there is a one in 10,000 chance of an error in any coefficient chosen at random (including coefficients which should be zeros). In a small 100 x 100 matrix, the expected number of bugs in coefficients is 1, the probability of no bugs is 0.37 and the probability of five or more coefficients being wrong is 0.0037. Table 1 shows how these probabilities change as matrix size increases. The probability of there being at least one bug in the matrix increases to be over 99 percent for a matrix with 50,000 coefficients. Even more worrying, the probability of there being five or more errors in the matrix is almost 20 percent for a 30,000 coefficient matrix, which would not be considered particularly large.

Table 1: Probabilities of Errors in Coefficients if the Probability of an Error in a Randomly Chosen Coefficient is 0.0001 (N = number of errors)

Matrix size (rows x cols)	E(N)	P(N=0)	P(N=5)	P(N=10)
10,000	1	0.37	0.0037	*
20,000	2	0.14	0.053	*
30,000	3	0.050	0.18	0.0011
40,000	4	0.018	0.37	0.0080
50,000	5	0.0067	0.56	0.030

* $p<0.001$

The probabilities in Table 1 are based on the assumption that the probability of an error is independent of the size of a matrix. In reality, the probability may increase with matrix size due to the fact that it is more difficult to examine larger matrices and more difficult to recognise a bug when you see one. Table 2 is similar to Table 1 except that it is based on the assumption that the probability of an error in a randomly chosen coefficient is proportional to the matrix size (0.0001 for a 10,000 coefficient matrix, 0.0002 for 20,000, etc.). This assumption has a big effect on the probabilities. A 30,000 coefficient matrix now has a 94 percent chance of containing five or more errors and a 41 percent chance of 10 or more errors. These probabilities are based on simple assumptions and should be viewed as illustrative only. However they do highlight the great risk of bugs in large matrices especially considering that the matrix sizes used here are by no means large. Matrices with millions of coefficients are certainly in use.

With modern software and hardware, human ability to maintain and debug models is the only factor limiting their size and complexity. It does pose real limits which need to be recognised and respected. It is difficult to say how big is too big due to the repetitiveness of some models. MIDAS has little repetition. With allocation of adequate resources (0.3 to 0.5 person years per year) and strict adherence to the sorts of bug prevention strategies suggested here, its 400 x 300 matrix can be maintained with only occasional errors.

Table 2: Probabilities of Errors in Coefficients if the Probability of an Error in a Randomly Chosen Coefficient is Proportional to Matrix Size (N = number of errors)

Matrix size (rows x cols)	E(N)	P(N=0)	P(N=5)	P(N=10)
10,000	1	0.37	0.0037	*
20,000	4	0.018	0.37	0.0081
30,000	9	*	0.94	0.41
40,000	16	*	0.9996	0.96
50,000	25	*	0.9999	0.9998

* $p<0.001$

MUDAS is much bigger (1500 x 1300) but it is also more repetitive. Nevertheless we feel that a matrix of this size is right at the limit of what can realistically be maintained in a usable and fairly error-free state.

Taking money to build models of the size one sometimes hears about (tens of thousands of columns) may be self deluding if not outright dishonest and can reflect badly on modellers, on the agricultural economics profession and on the individuals involved. There is little or no prospect of satisfactorily debugging such a model, so any results from them must be subject to grave doubts.

(h) Maintaining a large MP model in a fairly bug-free state requires a large commitment of human resources.

Rarely are sufficient resources provided. We consider it likely that many large MP models in active use contain important bugs.

(i) Bugs you thought you had fixed can easily come back to haunt you.

Anyone who has been responsible for debugging a large model over a long period of time will be well acquainted with this principle. Later in our discussion of prevention strategies we suggest model naming and updating strategies which should avoid this problem.

(j) When you are actively searching for one bug, you are quite likely to discover other, unsuspected bugs.

The act of searching for a bug requires a high degree of mental acuity. It may also require critical examination of aspects of the model which have previously been neglected. Both of these factors can lead to the modeller stumbling onto previously unsuspected bugs.

(k) Good hardware and software can make debugging much less of an onerous task.

Debugging often involves making numerous changes to the model and conducting several model runs. Obviously, using the fastest available hardware minimises the response time for tests of hypotheses. Software can probably make an even bigger difference, not just with speed of solution, but even more so with the tools now available to edit matrices (e.g. Pannell 1988), generate solutions and summarise output (e.g. Pannell and Bathgate 1991).

3. Symptoms of Bugs

The more serious bugs are usually detected through one of the following symptoms appearing in the model solution (a) an unlikely model solution, (b) no feasible solution, or (c) an unbounded solution. The majority of these symptoms are observed during the model development and testing phase but they can also occur when the model is changed for a particular analysis. The change may introduce a new bug or it may allow an existing bug to express itself.

Infeasible or unbounded solutions are clearly indicated in the output from the computer program, but identification of unlikely solutions requires a degree of subjective judgement. An unlikely solution can be blatantly obvious or very subtle. In general an unlikely solution is one in which one of the elements of the solution is outside the range within which the modeller judges it should fall. The suspect element may be the level of an activity, the shadow cost (marginal or dual value) of an activity, the level of slack for a constraint or the shadow price for a constraint. Examples of various types of unlikely solution include:

- an activity is selected at a level judged to be too high;
- an activity which you judge should be included in the solution at non-zero level is not included;
- an activity which you judge should not be included in the solution at non-zero level is included;

- the shadow price of a constraint is very different from the range within which you judge it should fall;
- the shadow cost of an activity is very different from the range within which you judge it should fall;
- a constraint which you judge should be binding in the solution has a non-zero slack value;
- there are two similar copies of a model which you believe should give the same basic solution but they do not;
- the solution seems consistent with the relationships and constraints included in the model, but an expert in the system being modelled advises that the solution is not consistent with the real world.

Later we suggest strategies for determining what type of bug, if any, is causing these symptoms. The symptoms listed above all relate to problems with the model solution. However many bugs are too subtle or minor to have a detectable effect on the solution. It may be that a bug affects the levels of several activities, but that the resulting levels are plausible, even though they are incorrect. Alternatively the model user may have no prior expectation about which of a range of activities will be included in the optimal solution. In this situation a mis-typed coefficient could dramatically alter the optimal solution without arousing suspicion.

There are two ways of dealing with these more subtle bugs. One is to detect them through careful examination of the model's coefficients, checking their consistency with the underlying assumptions of the model. The potential for tedium in this task is great, especially in large models. The other weapon against subtle or minor bugs is to prevent them occurring in the first place, as we discuss later.

4. Strategies for Debugging

Once a symptom has been detected, the next step is diagnosis of the cause. An important element of the diagnosis is an awareness of the full range of possible causes of the observed symptom. Table 3 shows one way of categorising the range of possible diagnoses.

Here we are concerned with symptoms which occur in a model solution, not with bugs which are initially detected through checking of model inputs. There is no fool- proof strategy which will lead directly to the diagnosis of a bug and it is difficult to generalise about the best strategy. However a methodical approach is bound to be more productive than a random search. A useful analogy can be drawn between debugging and scientific research. The practice of science and efficient debugging involve similar elements.

Firstly, there is the requirement that the scientist (or modeller) immerse herself or himself in the problem. This involves detailed study of the general field of research (or of the model and its assumptions). The first stage will be already partially complete since, presumably, the person doing the debugging is thoroughly familiar with the model. The other element of this phase is to become familiar with the behaviour of the bug. Extra model runs may be needed to reveal circumstances in which the bug does and does not occur.

The second element is identification of a range of possible explanations for the problem being addressed. It is sometimes easy to narrow the range of reasonable diagnoses. Table 4 shows a checklist of how the range of possible diagnoses can be narrowed for particular symptoms apparent in the model solution. Where the table indicates that a diagnosis can be ruled out, it means it can be ruled out as the cause of the symptom indicated. Clearly it doesn't necessarily mean that the problem is completely absent from the matrix. Some diagnoses cannot be ruled out altogether but are quite unlikely to cause the indicated symptom.

Notice that most diagnoses indicated in Table 4 fall under heading 2: the model is consistent with the underlying assumptions. It is usually impossible to rule out the converse diagnosis (that the model is not consistent with the underlying assumptions) without further checking of the matrix structure and contents.

In the third stage, specific hypotheses are formulated and tested in experiments. This is discussed in detail in the next section. Fourthly if the process is successful, information from the experiments is integrated with information about the general field to provide an understanding of the problem. Finally, the new understanding may allow improved management of the system being studied (or correction of the bug).

Just like science (Koestler) debugging cannot be a cold, calculating and linear process. Both involve essential elements of inspired guesswork, hunches and sudden flashes of insight which cut through the mist.

Table 3: Possible Diagnoses of a Suspected Bug

1. The model is not consistent with the underlying assumptions.
   - 1.1 A coefficient is incorrect.
     - 1.1.1 A coefficient has been mistyped.
     - 1.1.2 A coefficient has been given the wrong sign.
     - 1.1.3 Inconsistent units of measurement have been used when deciding on the value for a coefficient.
     - 1.1.4 A coefficient has been miscalculated.
     - 1.1.5 A coefficient has been omitted from the matrix.
     - 1.1.6 A coefficient has been placed in the wrong place in a matrix.
   - 1.2 A constraint is incorrect.
     - 1.2.1 A constraint is operating in the wrong direction (e.g. as a "greater than" when it should be a "less than").
     - 1.2.2 A needed constraint is omitted.
     - 1.2.3 A constraint is ill-conceived (e.g. coefficients omitted or in the wrong activities).
     - 1.2.4 The model is over-constrained; an extra, unnecessary constraint has been included.
   - 1.3 An activity is incorrect.
     - 1.3.1 An activity is ill-conceived (e.g. coefficients omitted or placed in the wrong constraints).
     - 1.3.2 A needed activity is omitted.
2. The model is consistent with the underlying assumptions.
   - 2.1 The underlying assumptions are consistent with the real world.
     - 2.1.1 The unexpected result is a new insight about the real world.
     - 2.1.2 The model result is correct but is being misinterpreted (e.g. the level of an activity may be interpreted using incorrect units of measurement).
     - 2.1.3 There is a bug in the software used to solve the model.
     - 2.1.4 Bad or inadequate control instructions were given to the software used to solve the model (e.g. instruct program to maximise the objective function when it should be minimised).
     - 2.1.5 The model is badly scaled, resulting in an accumulation of rounding errors when the model is solved.
   - 2.2 The underlying assumptions are not consistent with the real world. The model may need new constraints or activities or changes in the values of some coefficients. needed constraint is omitted.

Table 4: Diagnoses From Table 3 Which Can be Ruled Out as the Cause of Particular Symptoms or are Unlikely to be the Cause

Symptom	Ruled out Diagnoses[a]	Unlikely Diagnoses
No feasible solution	2.1[b], 2.2, 1.2.2	2.1.3, 2.1.4
Unbounded solution	2.1[b], 2.2, 1.2.4	2.1.3
The solution includes elements which are not possible in the real world system being modelled	2.1.1, 2.2	
The solution algorithm includes an automatic facility for scaling a matrix and this facility is switched on		2.1.5

[a] Ruling out a diagnosis at one level also rules out all diagnoses at a lower level. For example, if diagnosis 2.3 is ruled out then so too are diagnoses 2.3.1 and 2.3.2.

[b] Although unlikely if a reputable solution algorithm is used, diagnoses 2.1.3 and 2.1.4 should not be completely ruled out. Occasionally adjustments to the feasibility tolerance used in a package will cure the problem of not being able to find a feasible solution in a perfectly valid and feasible model.

Also, neither process will proceed neatly and linearly through the stages described above. There will be overlap between stages and possibly feedback of information to an earlier stage.

5. Techniques for Testing Hypotheses

The third element of the debugging process listed above is to formulate and test hypotheses. There are many techniques for looking at a model's inputs and/or outputs or of manipulating and comparing model solutions which can help to test for particular problems. We will examine various techniques categorised according to the diagnoses in Table 3. Then we will suggest the order in which the different techniques should be applied, depending on the symptoms observed.

Diagnosis 1

Testing for an error in the matrix coefficients is, conceptually, quite simple. It requires examination of coefficients in the appropriate region to ensure that they are consistent with the underlying assumptions of the model. In practice the problem is deciding which coefficients to examine. Several of the techniques suggested here are designed to help narrow the focus of the search for bugs. Searching for a bug generally starts by employing a technique to identify a suspicious section of the matrix. Greenberg (1993) calls this first stage "isolation".

For now, suppose that the hunt has been narrowed to a section of the matrix: a row or column or small block of coefficients. At this stage there is no alternative to a visual examination of all coefficients in the suspect region. It is often possible to use a text editor to look at data in the format used by the computer algorithm, but it is probably more productive to examine the data *in situ* in the matrix. This provides additional visual cues (presuming that the model has been thoughtfully and consistently constructed) which can make a difference in recognising a problem. A matrix editor like GULP (Pannell 1988) is invaluable for this purpose as it allows the modeller to see coefficients in context without having to print out the matrix. Clearly the person undertaking the examination needs to be thoroughly familiar with the modelling technique and the model's assumptions so that he or she can recognise an incorrect coefficient. The modeller also needs to be aware of all the ways in which a coefficient, constraint or activity can be in error, as listed in Table 3, and check thoroughly for each. Identification of an error must, in the end, involve an examination of this type. Greenberg (1993) calls this stage the search for an "explanation". Isolation usually helps in reaching an explanation. Once the problem has been fully diagnosed, selection of a treatment to correct the problem is usually straightforward.

Consider now the possible methods of isolating a problem.

(a) Often, the symptom observed in the model solution provides valuable clues. Be sure to make the most of any clues which are provided. If the problem area is not obvious, examine the solution for logical inconsistencies in the relative levels of different activities or for unrealistic shadow costs of non-basic activities or shadow prices of binding constraints. If this leads to questioning a particular section of the matrix, then proceed to a detailed examination of the coefficients in that section.

(b) Conduct model runs to determine circumstances where the bug[1] does and does not occur. Does it always have an impact on the solution or does it only express itself when some parameters take particular values? For example if a bug occurs in a coefficient of activity A which causes the selection of unrealistic levels of activity B, the bug will only be apparent when activity A is included in the optimal solution. A series of runs in which a key parameter is varied over a wide range is a good way of examining the behaviour of the bug. Try to use information about its behaviour to determine which constraints and which activities are the root of the problem.

(b) If you suspect that a bug occurs in a particular activity but are unable to identify the specific problem, a potentially useful technique is to compare the solutions of two very similar models: one with the activity constrained to zero level and the other with the activity constrained to a low level (e.g. 1 unit). Then calculate the difference between the solutions in the level of each

[1] In general, reference to a "bug" in this section should be interpreted as a "hypothesised bug".

activity and the degree of slack for each constraint. This reveals all the direct and indirect impacts of the activity on other activities and constraints. This can sometimes reveal a linkage between the suspect activity and another activity which should not be occurring, leading you to examine the matrix for unintended links. Undertaking such a comparison can be a very tedious operation without some computerisation. Options to do this include using a custom-written computer program or a spreadsheet package.

(c) If you have a recent previous version of the model in which the unexpected result does not occur, conduct a comprehensive comparison of the data for the two versions. This may reveal a bug which has been introduced inadvertently. It is possible that the bug was present in the previous version without manifesting itself in the model solution. In this case the bug will not be revealed directly by the comparison of data. However, the comparison will at least show which coefficients have changed, allowing you to search for the change which has caused the bug to reveal itself. Such information should give clues as to the location of the actual bug. If the data are stored in a text file (e.g. in MPS format), importing the two models into a spreadsheet package can greatly ease comparison of data files.

(d) Another technique is to delete sections of the matrix (groups of rows or columns) and see if the problematic result still occurs. This is only possible in some circumstances. Care is needed when deciding which parts to delete as it is easy to introduce new problems by removing a crucial constraint or activity. The safest approach is to limit such deletions to discrete and fairly self contained sections of the model. For example if a model includes several different regions, it will probably be possible to delete one of the regions without disturbing the functionality of the other regions.

If the model data is stored in MPS format, deletion of a constraint with more than a couple of coefficients is a very tedious task. Coefficients for each activity are grouped together but this means that coefficients for any one constraint can be distributed throughout the data file. The solution is to use a matrix editor (such as GULP), which makes deletion or addition of constraints a simple task.

Greenberg (1993) outlines some isolation techniques for diagnosis of infeasible models. These are outlined further below.

Tests of hypotheses which do not involve bugs in the matrix tend to be quite specific to particular diagnoses:

Diagnosis 2.1.1

Testing a hypothesis that an unexpected result is correct and that the model is free of bugs is, unfortunately, impossible. McCarl (p.157) states that:

> Models can never be validated, only invalidated. ... The outcome of a model validation process is either a model that has been proved invalid or a model about which one has an increased degree of confidence.

Although this is strictly true, it is possible to indirectly test the validity of a particular result. Suppose that you have searched thoroughly for a bug without finding one but are unable to convince yourself that a particular unexpected result is valid. Even if you do have a plausible explanation, some results clash so strongly with prior expectations that any attempt to publicise them without very convincing supporting arguments will threaten your credibility. One approach is to try to reproduce the result using a different technique; try using a different modelling approach (e.g. dynamic programming, simulation) or a much simpler MP model[2]. If you can independently reproduce the result it at least gives you confidence that the result is correct and it may also provide a convincing explanation for the result.

Apart from this, one is limited to validation through absence of invalidation. If you do have a plausible explanation, conduct additional model runs to attempt to falsify it. This can be done by preventing the mechanism for your plausible explanation from operating. For example, suppose you have two similar models but there is an unexpected difference in the level of activity A between the two solutions. You hypothesise that this is due to changes in the level of activity B. Try constraining the level of activity B to be the same in both solutions. If the difference in activity A then disappears, this lends support to your hypothesis about the mechanism and helps dispel doubts that it may be simply due to a bug.

[2] Thanks to Brian Hardaker for this suggestion.

Diagnosis 2.1.2

This is simply a matter of careful checking. For example if the level of an activity seems wrong, refer to the model documentation and check that the coefficients for that activity are consistent with the unit of measurement you are using to interpret the result. Also check that the puzzling solution is reported by the software as being optimal. It may be that the activity levels are so strange because the solution is infeasible or unbounded.

Diagnosis 2.1.3

Implementing an accurate and reliable computer package for mathematical programming is notoriously difficult. Even the most highly reputed packages are not immune from bugs. For example Tice and Kletke (1984) reported a serious bug in a version of MPSX, a powerful and widely used package for mainframe computers. After a period of experience with a model the modeller may gain confidence that the software is in fact correctly finding optimal solutions. However, in the development phase the possibility of errors associated with an algorithm should not be ruled out.

Diagnosis 2.1.4

Occasionally, a problem of bizarre and puzzling model solutions can be resolved by correctly informing the algorithm that the objective is maximisation or minimisation.

Some MP computer packages allow adjustment of the "tolerances" used to test whether a given solution is feasible or optimal. A feasibility tolerance is a small number (e.g. 1×10^{-8}) which gives the maximum sum of infeasibilities for all constraints before the basis is considered to be feasible. If the package is reporting that it cannot find any feasible solution but you are unable to find any problem with the model's structure or coefficients, adjustments to the feasibility tolerance may solve the problem. For example, try relaxing the tolerance to 1×10^{-6}. The documentation for the algorithm may give guidance about which values to try.

The optimality tolerance is the minimum improvement to the objective function which an activity must make before it will be brought into the basis. If the computer package appears to be getting stuck in a loop so that it never reaches the optimal solution, adjustments to the optimality tolerance may solve the problem.

Diagnosis 2.1.5

A badly scaled matrix is one in which there is a big difference in the magnitudes of coefficients used. A badly scaled matrix has a greater chance of failing to solve because of the accumulation of rounding errors which occur in every mathematical operation on real numbers in a computer. Such rounding errors are exacerbated by poor scaling. Symptoms of accumulated rounding errors can include an unbounded solution, an infeasible solution or an apparently optimal solution which is actually not consistent with the constraints of the model. There is no hard and fast rule about how bad scaling can be before serious rounding errors occur. Many packages include warning messages based on a rule of thumb regarding the ratio between the largest and smallest coefficients in the matrix.

If a change in scaling is needed, it is simply a matter of using different units of measurement for some rows and/or columns. Converting the units of measurement entails multiplying all the coefficients in a row or column by the same value. This can be done for as many rows or columns as necessary to ensure that coefficients are not too different. It is wise to use scaling multipliers which do not make the interpretation of output too difficult.

Diagnosis 2.2

Sometimes you will come to believe that the model is correct within itself but that it is failing to capture some aspect of the real world. Typically you may feel the need for new constraints or for distinguishing between similar but slightly different activities or constraints. This requires interaction with an expert in the biological or technical system being modelled. Such interaction should be viewed as part of the ongoing process of model development (e.g. Morrison, 1987). Your expert may volunteer suggestions that such changes are needed. A thorough and up-to-date documentation is very helpful for facilitating productive interaction with outside experts.

6. Matching Hypotheses to Symptoms

Having surveyed the available techniques and tools, let us now consider how one should approach particular symptoms. In what order should hypotheses be tested and these techniques applied? The suggestions

which follow are certainly not exhaustive in their coverage of the full range of symptoms. They also cannot be applied in an unthinking "cookbook" manner; as we have observed, successful debugging requires careful thought as well as creativity and inspiration.

There are two considerations when deciding on the order in which hypotheses should be tested: the relative likelihood of alternative hypotheses and the ease with which they can be tested. Commonly there are several hypotheses which could equally well explain the symptom and which are about as likely as each other to be true. However the difficulty of testing different hypotheses can vary widely, so we recommend that, in the absence of any reason to suspect a particular type of bug, ease of testing should initially be the main criterion used.

No Feasible Solution

For infeasible and unbounded solutions, one part of the diagnosis problem requires no effort: there clearly *is* something wrong with the matrix. Clues about where to start looking are provided; one or more constraints is indicated as being infeasible or one of the activities is listed as unbounded. Unfortunately, the clue rarely leads to easy identification of the cause of the problem. This applies particularly to infeasible models, where the cause of the problem may lie in a constraint which is not reported as being infeasible.

Start by observing which rows are reported as being infeasible in the program output. If there are not too many infeasible rows, carefully check their coefficients. Coefficients being entered with the wrong sign (i.e. positive when they should be negative) are a possible cause of infeasibility.

The next step, if needed, is to check that all constraints are operating in the correct direction. Are there any "less than" constraints which should actually be "greater than" constraints, and vice versa?

Next see if there are any "equals" constraints in the model which can be relaxed to "less than" or "greater than" constraints. Some modellers are prone to overuse "equals" constraints and this can easily lead to unnecessary infeasibilities.

After exhausting these simple approaches you must resort to more time consuming techniques. Greenberg (1993) outlined several techniques which can help to isolate an infeasibility. One is based on a theorem by Dantzig which says that if an LP is infeasible, there exists an infeasible one-constraint LP formed by adding up the constraints, weighted by their shadow costs in the infeasible solution. The one-constraint LP potentially has the same number of activities as the original, although some coefficients are likely to be zero. The coefficient for an activity in the one-constraint model is calculated by multiplying each coefficient of the activity by its corresponding shadow price in the infeasible solution and adding up the results. Greenberg observed that:

> This has great appeal for diagnosis formation because, on the surface, it seems that explaining a one-constraint infeasibility is easy, at least compared to the original LP. (Greenberg, 1993, p. 124).

The main useful information from this technique is which activities have non-zero coefficients in the aggregated constraint, not necessarily the numerical values of their coefficients. If the number of non-zero coefficients in the aggregated constraint is low, identifying the cause of the infeasibility may be straightforward. On the other hand, there may be no non-zero coefficients, which provides no new information, or a very large number of them, which does not help to clarify the issue very much.

A second technique is to find a smaller subset of the matrix which is still infeasible. Ideally you seek the "irreducible infeasible subsystem" which is no longer infeasible if any of its constraints are dropped (Greenberg, 1993). This method was first suggested by Debrosse and Westerberg. Chinneck and Dravnieks (1991) have developed several different methods for identifying the irreducible infeasible subsystem. Start by removing all constraints having zero-shadow prices in the infeasible solution. Then remove activities which have no coefficients in any of the remaining constraints. The resulting model is still infeasible but it may be further reducible. To test this, one of the methods involves dropping another constraint at random (and any further activities with no coefficients). If the model is still infeasible, leave that constraint out and proceed to drop another constraint at random. When the model becomes feasible, add back in the constraint which made the difference (and any activities you dropped with the constraint). Keep this constraint in the model but drop each of the remaining constraints one at a time. If dropping a constraint does not make the model feasible, leave it out. After testing

all of the remaining constraints, the resulting subsystem is irreducibly infeasible.

Like the one-constraint approach, while this can isolate the infeasibility, it does not necessarily provide an easy explanation of its cause. Nevertheless it is now possible to search for that explanation in what is almost certainly a much smaller model than the original

Another way to reduce the model which may be useful in some circumstances is to add "artificial activities" or "artificial variables" to the model. For each less-than constraint, add an activity with a coefficient of -1 in the row and zero coefficients in every other row except the objective function. For each greater-than, add a similar activity with a coefficient of 1 in the row. For each equals constraint, add two artificial activities, one of each type. Include an unfavorable coefficient in the objective function of each artificial activity, such as 1000 in a minimization problem or a -1000 in a maximization. These artificial activities will not be selected unless they have to be to achieve feasibility, but they allow the package to find a feasible solution when it would otherwise be impossible. Observe which artificial activities are selected at non-zero levels and delete them. Attempt to solve the model again. Delete any new non-zero artificial activities. Repeat this until the model becomes infeasible again. At this point, the constraints with deleted artificial activities contain the cause of the infeasibility. The resulting infeasible subsystem is not necessarily irreducible, but the technique is simple to apply and does not use the shadow prices from the infeasible solution, which are not provided by some LP packages. If desired, one of Chinneck and Dravnieks' (1991) techniques (such as the one described above) can be applied to the subsystem to achieve irreducibility.

Note that there is a danger in this approach of causing the model to become unbounded. This will occur if the objective function coefficient of one of the artificial activities is not sufficiently unfavorable to counteract a benefit generated elsewhere in the matrix. To avoid this, use very large unfavorable objective function coefficients in the artificial activities.

The third technique outlined by Greenberg (1993) is "successive bounding". This is not described in detail here because it is only a practical option if the LP software being used has the capacity to conduct it automatically. The advantage of successive bounding is that, when it does work, is provides not just an isolation, but an explanation of the cause of the infeasibility.

If you have been unable to find the cause of an infeasibility and suspect that there is actually nothing wrong with the matrix, try adjusting the feasibility tolerance in your computer package. Alternatively some packages will save details of the basis for later use. Instruct the package to save the basis which is nearest to being feasible. Then instruct it to solve again, starting from this basis. The different accumulation of rounding errors may allow the package to correctly identify that the model is feasible. This is a suggestion which applies to relatively large models. If that does not work, test for a failure of the LP software by making substantial changes to key objective function coefficients and trying to solve the model. If it does correctly solve it means that the original model is not infeasible, since changing only objective function coefficients cannot affect the feasibility of a model. In this case, it may be possible to find the optimal solution for the original model by solving it starting from the optimal basis for this revised model.

If you have tried all these techniques and are still unable to obtain an optimal solution, it may be that your model correctly represents the problem but the problem has no feasible solution.

Unbounded Solution

Start by checking that the direction of optimisation (maximisation or minimisation) used by the computer program is consistent with the model.

Secondly identify the unbounded column from the program output. The problem is that there is nothing preventing this activity from being selected at an infinite level. Thus the modeller should work through all constraints of the model and check whether one of them should be affecting the activity but is not. Possible reasons for the problem include: (a) coefficients with the wrong sign, (b) constraints operating in the wrong direction, (c) coefficients missing or in the wrong place, (d) an omitted constraint.

Solution Conflicts with Expectations

It is common to be surprised by a result obtained from a large MP model. Such surprise should be met with scepticism and followed by a careful search for causes. We suggest that the search proceed in the following steps, which are in order of increasing difficulty: (a)

check your interpretation of output, (b) check for errors in computer commands used and for obvious errors made by the computer algorithm, (c) check for bugs in the model and (d) check hypothetical explanations why the result may be correct.

Checking interpretation of output includes checking that the solution is reported as being optimal, not infeasible or unbounded. If so, check the units of measurement you are using to interpret the solution.

Technical problems to check for include the direction in which the model was optimised and other problems with the control instructions given to the program. Bugs with the computer package may leave obvious symptoms (e.g. a mix of positive and negative shadow costs of activities). If you suspect that the solution is not truly optimal, check that the tolerances being used by the algorithm are consistent with any instructions given in program documentation.

If you have not identified the problem by this stage, there is no alternative to searching for bugs in the matrix. In practice you are likely to investigate alternative hypotheses in the order suggested by the particular symptoms observed. However we put forward the following suggested order in which to test hypotheses in cases where the modeller is uncertain how to proceed.

If the problem is apparently in a particular activity or constraint, examine it for obvious errors in coefficients: typing errors, coefficients with the wrong sign, coefficients missing or in the wrong place, inconsistent units of measurement or an error of calculation. If appropriate, check that constraints are operating in the right direction.

Unexpected solutions are often associated with one or more activities being selected at levels outside the range judged to be reasonable. (For convenience let us call these "target activities"). If you don't initially find any problem with the coefficients or constraints of the model, add a new constraint which forces the target activity to be selected at a level which corresponds to your prior expectations. It may be that there is some error which is forcing a high or low level of the activity. This will be revealed either by an infeasible solution or by behaviour of the constrained model.

If the modified model is infeasible, it means that the original model contains a "forcing substructure" (Greenberg, 1994). This is where an activity is forced to take a particular value by the model's constraints, rather than as a result of its impact on the objective function. If an activity is forced to its lower bound of zero in every feasible solution, it is said to be "nonviable" (Chinneck, 1992) and if this is not intentional it probably indicates a bug. An activity's level may be determined by a forcing substructure even if the level varies in different model solutions. Forcing substructures are not necessarily bugs, but even if they are not, detecting and explaining them is valuable in understanding the model's results. One method of searching for forcing substructures is to use the successive bounding technique to identify redundant constraints on individual activities (Greenberg, 1994). For example if an activity must have a positive value in every feasible solution, its non-negativity constraint is redundant. Greenberg (1994, p.125) argued that "seeking redundancies reveals forced levels that deepen our understanding of a solution by separating that which is forced by implication and that which is determined by economic trade off".

If the unusual activity level is not due to a forcing substructure, it could be that the activity is having a larger or smaller beneficial impact on the objective function than you expect. To test this, use the technique described earlier for comparing two solutions in which the level of an activity is constrained to differ by a small amount (say one unit). First constrain the activity to a low level (e.g. zero) and then to a slightly higher level. The difference in objective function values can easily be calculated but it is also possible to determine which factors are contributing to the difference. Do this by (a) calculating the difference between the two solutions in the level of each activity and (b) multiplying differences by the objective function value for that activity. The sum of these values gives the net difference in the objective function value. You may find unexpected indirect effects on the objective function which explain the unusual result; check that these are not due to bugs.

Possibly the reason for the high or low level of the target activity is a problem with an alternative activity which competes with the target activity for resources. Check for activities which compete with the target activity and in each case examine them for bugs. If the level of the target activity seems too low, search for bugs which bias the model toward high levels of the alternative activity.

It is still possible to suspect a bug even if all activity levels conform to prior expectations. Inappropriate values for shadow prices, shadow costs or constraint slacks may be symptomatic of a bug which will affect activity levels if the model is altered (e.g. in sensitivity analysis).

A very high shadow price can be investigated by comparing two solutions in which the constraint limit (right hand side term) is varied by a small amount (say one unit). Calculate all differences in activity levels. Unexpected large differences may indicate a bug.

A very large shadow cost for an activity is fundamentally the same situation as an activity being selected at a lower level than expected. Investigate it using the strategy described above for unusual activity levels.

A large constraint slack may indicate (a) an error in the right hand side term for that constraint, (b) a low or missing positive coefficient or an erroneous negative coefficient in an activity (for "less than" constraints) or (c) a problem with low availability or high usage of a resource represented in another constraint.

Diagnosis of an unusual model result is sometimes particularly elusive. In these cases adopt the techniques described earlier for locating the bug (or other explanation) within the matrix: conduct model runs to determine situations where the bug does and does not occur; if possible delete sections of the matrix and see if the problem still occurs. A deletion which cures the symptoms of a bug may indicate that the bug occurs in the deleted section.

If no bug has been found by now, it may be that there is no error in the matrix coefficients or solving algorithm. Instead the problem may be a failure to correctly represent the system being modelled. Some aspect of the system may have been incorrectly excluded from the model or included in the model in a way which fails to accurately represent its nature. Alternatively it may be necessary to look for hypotheses which explain why the result is, in fact, correct. If possible, conduct tests to attempt to refute these hypotheses.

When a model is first constructed, it is advisable to conduct runs for the purpose of trying to reveal bugs hidden in the matrix. One strategy, suggested by McCarl and Apland (1986), is to restrict the values of all activities (using constraints or bounds) to a set of values observed in the real world. Then try to solve the model to check whether the real-world solution is feasible within your model. If the solution is not feasible, you need to find out why and correct the problem. If the model does solve, check for unusual activity levels or constraint slacks. If a lack of data means that you cannot constrain every activity in the model to a real-world value, it can be valuable to constrain sub-sections of the model for which you do have information about levels. Check for feasibility and for any unusual side effects.

Another useful approach is to run wide ranging sensitivity analysis, varying key parameters through plausible (or even implausible) values and observing how the model behaves. Often errors in the model are revealed most starkly when it includes unusual or unlikely combinations of parameter values.

7. Strategies for Preventing Bugs

Debugging is difficult, frustrating and time consuming. The discovery of a bug after a set of model results has been publicised is potentially quite damaging to the credibility of the model and its developers. Many bugs go undetected for a very long time, possibly forever. For these reasons, prevention of bugs is crucial to the success of a modelling project. Key elements of bug prevention are discipline and care, but there is also a range of relatively simple strategies which can contribute to prevention of bugs. In our experience with the MIDAS and MUDAS models, the following are useful elements of an overall bug prevention strategy.

(a) During model development and construction, proceed in small steps. Thoroughly test and debug the model before adding the next component or the next level of complexity.

(b) For a particular LP package, if you have a choice between entering the data with a text editor or with a matrix editor, use the matrix editor. This will dramatically reduce the risk of typing errors and save considerable time. Also you are more likely to spot existing errors if you are working with a matrix format than in awkward formats like "MPS" which is used by a number of major packages. Use of a text editor to change MPS data should be limited to small and simple changes.

(c) Use a "macro" to automate repeated key strokes when entering or editing a model. Some pro-

grams (e.g all spreadsheets) have a macro facility built in, or it can be made available in a separate package. Macros not only save time but also reduce the likelihood of errors being introduced through typing fatigue.

(d) For models with an intended long life, develop computerised data entry systems. We have created user-friendly spreadsheet templates which allow users to view and change model assumptions without them needing to be familiar with the matrix of the model. A simple example is given by Pannell and Falconer (1987) and a full listing of current spreadsheets is given by Pannell and Bathgate (1991). These spreadsheets allow users to view parameters in a format and with units of measurement to which they can relate easily. The spreadsheets perform arithmetic operations on the parameters to calculate the required matrix coefficients. Of course there is a risk that the formulae entered in the spreadsheet themselves contain errors but at least these errors only need to be detected and corrected once. Without such a system, coefficients must be calculated by hand and are more prone to error.

The GAMS system provides an alternative approach to data entry which also may contribute to bug prevention. Data are presented to the package in tables and algebraic inequalities rather than the usual matrix format. This provides some of the advantages of the spreadsheet approach described above.

(e) Occasionally print out and inspect part or all of the matrix showing numbers. Some computer packages include the facility to print out very compact summaries of the matrix using symbols to represent coefficients of different magnitudes. While this can be very convenient and useful for checking the consistency of a model's structure, it can also mask errors which would be obvious from an examination of the complete matrix.

(f) Have one person with ultimate responsibility for changes to the model and for ensuring that it is up to date and free of bugs. Personal responsibility is very important. We are aware of a research institution at which a major model was generating impossible solutions but because of a lack of individual responsibility, the problems went undiagnosed and unresolved. This general problem is compounded by the general lack of recognition among research administrators of the importance and resource requirements of model maintenance and debugging. There is often little incentive for individuals to take on this role; they can earn more kudos in other activities.

(g) Have only one master copy of the matrix to which changes can be made. (Of course keep back-ups of this). Failure to do this is the usual cause of bugs which return after you thought you had fixed them.

(h) Have a meaningful and consistent system for naming model data files according to model version. MIDAS model versions are named something like EWM91-4. The EWM indicates the region represented, 91 is the year and the 4 indicates that this is the fourth version of the model to be released during the year. It is essential to assign a new version name after every change or set of changes to the model. On DOS microcomputers, consistent use of file extensions is also helpful. All our MPS data files are allocated the extension "MPS".

(i) Have a meaningful and consistent system for naming rows and columns. Make sure that a legend is included in the model documentation and that the documentation is readily available.

(j) Use intuitively obvious units for rows and constraints. Do not worry about scaling unless necessary. Record units of measurement within the legend of row and column names.

(k) Structure of the matrix (i.e. order of rows and columns) can be important. Group related rows and columns together and be consistent about the order used so that the visual pattern of coefficients can help highlight a coefficient out of place or with the wrong sign.

(l) Have a good system for reporting bugs or problems or suggested changes to the person responsible for the model. One approach is to distribute "bug report sheets" with appropriate headings and questions to all model users.

(m) Have a system for recording all changes to a model. Keep a file or log book showing the date, the reason and the substance of each change and

the revised name of the new version produced. It can also be helpful to record the sources of information used to make the change.

(n) If a system for summarising and condensing output is used, be sure to examine a complete model solution occasionally. The reason is the same as for the summarised (symbolic) matrix printout.

(o) Employ someone who is unfamiliar with the model and have them go over the entire matrix in detail, checking calculations and questioning the logic of matrix structure. The aim should be to convince themselves that they understand the reasoning behind and derivation of every coefficient. This should occur whenever a new person is employed to work on an existing model. As well as giving them a thorough knowledge of the model, even quite subtle bugs in the model can be detected given such detailed attention. It is also helpful to have as many people as possible examine the assumptions and logic of the model, even if each only covers a small subsection of the model.

8. Concluding Comments

How many bugs are too many? In practice there is a need to equate the marginal cost of reducing bugs with the marginal benefit of their exposure. With large models the cost of debugging is high, but presumably the value of the information is also high otherwise resources would not have been put into developing the model. It comes down to a difficult judgement about the probability of bugs, importance of the information, size of the model, personal reputation, etc. On the other hand, fear of potential (but unknown) bugs should not prevent use of the model. All reasonable care is care enough.

Keeping a large model up to date and bug-free is a difficult, thankless, under-recognised task. No large MP modelling project should be contemplated without recognition of and adequate allowance given for debugging and maintenance. Given the rapid increase in probability of bugs in large models, "adequate" probably means more than is usually allocated.

References

CHINNECK, J.W. (1992), Viability analysis: A formulation aid for all classes of network models, *Naval Research Logistics* 39, 531-543.

CHINNECK, J.W. and DRAVNIEKS, E.W. (1991), Locating minimal infeasible constraint sets in linear programs, *ORSA Journal on Computing* 3, 157-168.

DANTZIG, G.B. (1963), *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ.

DEBROSSE, C.J. and WESTERBERG, A.W. (1973), A feasible-point algorithm for structured design systems in chemical engineering, *American Institute of Chemical Engineering Journal* 19, 251-258.

GASS, S.I. (1983), Decision-aiding models - validation. assessment and related issues, *Operations Research* 31(4), 603-631.

GREENBERG, H.J. (1993), How to analyze the results of linear programs - Part 3: Infeasibility diagnosis, *Interfaces* 23(6), 120-139.

GREENBERG, H.J. (1994), How to analyze the results of linear programs - Part 4: Forcing substructures, *Interfaces* 24(1): 121-130.

KINGWELL, R.S. and PANNELL, D.J. (Eds) (1987), *MIDAS, A Bioeconomic Model of a Dryland Farm System*, Pudoc, Wageningen.

KINGWELL, R.S., MORRISON, D.A. and BATHGATE, A.D. (1992), The effect of climatic risk on dryland farm management, *Agricultural Systems* 39, 153-175.

KOESTLER, A. (1964), *The Act of Creation*, Hutchinson, London.

MAGEE, B. (1973), *Popper*, Fontana/Collins, London.

McCARL, B.A. and APLAND, J. (1986), Validation of linear programming models, *Southern Journal of Agricultural Economics* 1986, 155-164.

MCCARL, B.A. (1984), Model validation: an overview with some emphasis on risk models, *Review of Marketing and Agricultural Economics* 52(3), 153-173.

MORRISON, D.A. (1987), Background to the development of MIDAS, In: R.S. Kingwell and D.J. Pannell (Eds). *MIDAS, A Bioeconomic Model of a Dryland Farm System*, Pudoc, Wageningen, 5-14.

MORRISON, D.A., KINGWELL, R.S., PANNELL, D.J. and EWING M.A. (1986), A mathematical programming model of a crop-livestock farm system, *Agricultural Systems* 20(4), 243-268.

PANNELL, D.J. (1988), An integrated package for linear programming, *Review of Marketing and Agricultural Economics* 56(2), 234-5.

PANNELL, D.J. and BATHGATE, A. (1991), *MIDAS, Model of an Integrated Dryland Agricultural System, Manual and Documentation for the Eastern Wheatbelt Model Version EWM91-4*, Miscellaneous Publication 28/91, Department of Agriculture, Perth, Western Australian, 162 pp.

PANNELL, D.J. and FALCONER, D.A. (1987), Solution, interpretation and revision of MIDAS, In: R.S. Kingwell and D.J. Pannell (Eds). *MIDAS, A Bioeconomic Model of a Dryland Farm System*, Pudoc, Wageningen, 55-63.

TICE, T.F. and KLETKE, M.G. (1984), Reliability of linear programming software: an experience with the IBM Mathematical Programming System series, *American Journal of Agricultural Economics* 66(1), 104-7.