

TK Solver Laws

Spring 2007 Edition

The purpose of the following material is to contribute to the collective understanding of the essence of TK Solver. It does not answer the question, “What is TK?”, and it does not attempt to catalog or explain TK techniques and tricks to any extent. It is a set of problem solving laws which are good to know, respect and live by.

The “laws” are supplemented with a sample model FRUSTUM in the Appendix. It deals with the frustum of a cone and it has 18 variables and 15+ rules. It is an extension of the cone model frequently used in TK Solver demos and training materials.

The **first law** of TK (and also, frequently, the user’s first problem):

*As many independent equations are needed
as there are unknown variables to be solved for.*

Hence the first thing one should do when designing or auditing a TK model is to check the balance of the number of variables, input variables and equations.

$$\text{equations} = \text{unknowns} = \text{total} - \text{input}$$

Problems arise with the concept of *independent equations*. Here we must depend on our education, insight and discipline to methodically perform the check.

The condition stipulated by this first law is necessary but not sufficient: Solving n independent equations doesn’t automatically yield the values of n unknowns. A real solution may not exist for certain values of some input variables, or for any values of some input variables, or for any values of any input variables. An example of a problem of the last category is given in the “Direct Solver” chapter of the TK documentation (“Existence of Solution” section).

Dependence/Independence Exercise

The appreciation for the concept and role of independent equations is as important as the technique of adding redundant equations to a model, as advocated in other parts of this document. Lack of this appreciation may lead to serious confusion as may be demonstrated with the following simple example of a right-angle triangle model (confusion may grow proportionally to the square of the model size). Note: It may be hard (i.e. boring) to read what follows without trying it in TK.

Here are four sets of equations with three equations in each set:

$a^2 + b^2 = c^2$	$a^2 + b^2 = c^2$	$\alpha + \beta = 90$	$\alpha + \beta = 90$
$\text{sind}(\alpha) = a/c$	$\text{sind}(\alpha) = a/c$	$\text{sind}(\alpha) = a/c$	$\text{sind}(\alpha) = a/c$
$\text{cosd}(\alpha) = b/c$	$\text{sind}(\beta) = b/c$	$\text{sind}(\beta) = b/c$	$\text{cosd}(\alpha) = b/c$

Each of the last three sets represents a complete system of three independent equations linking five variables, a, b, c, α, β . Each of them can be used for evaluation of any three unknowns

given the input values of the remaining two (by the Direct Solver in some cases, and with help of the Iterative Solver in other cases). The first set is different, although all its equations appear in the other sets. Firstly, there is no *beta*, but, more importantly, only two of the three equations are independent (if you square each of the last two equations and add them up, you will get the 1st equation). For these reasons, the first set of equations solves for three variables given a value of the fourth with iteration required in every case.

Consider the effects of cancelling the last equation of the first set and using the first two equations for evaluation of two unknowns, such as *a,b* or *a,alpha* or *b,c* or *b,alpha* or *c,alpha* . If *b,alpha* are given and the unknowns are *a* and *c*, one of the latter has to be guessed and the Iterative Solver invoked. Alternatively, the 3rd equation may be uncanceled and used as a redundant equation so that the Direct Solver can do its job.

Suppose *a,c* are unknowns, the third rule is canceled, there is an input value *b*=.8 (but no value assigned to *alpha*), and you assign guess values *a*=.6 and *c*=1. You press F9 and find the guess values of *a* and *c* moved to the output fields and *alpha* evaluated as 36.87 degrees. Well, that was a “lucky guess” because $.6^2 + .8^2$ happened to be equal to 1^2 . But try guesses *a*=.5 and *c*=1.1. You will be rewarded by the dreaded message “Iteration: Too many guesses”. The *a* and *c* stay as guess variables and their values change a bit as the Iterative Solver tries to find its bearings (which it failed to do); an interim value *alpha*=27 shows up in the output field.

What happens if we respond to the above mentioned error message by using the 3rd redundant equation. The Iterative Solver drifts unpredictably and lands on a solution which is correct, as it is one of the infinite number of solutions of an under-determined problem of solving two independent equations for three unknowns. However, it most likely doesn’t have anything in common with your intention and assignment. The latter was flawed to start with, but TK cannot tell you because it doesn’t directly recognize independence of equations. It generates two error terms in the 1st and 3rd equations matching the two guess variables, and iterates happily until the misleading end. The remedy: be aware of the number of independent equations and unknowns to solve for and don’t try to trick TK into yielding more by reckless or naive choices of guess variables.

Curiously, TK is not as defenseless in the case of linear equations or linear relationships. The shortage of independent linear equations matching the number of guesses (or, viewed from the other side, the presence of equations which are linear combinations of others) is flagged by an error message “Iteration: Dependency error” which is quite helpful in finding the culprit and fixing the problem.

The **second law** of TK (a unique TK feature which should be a more frequent subject of study and promotion):

A set of TK rules equivalent to m independent equations linking n variables provides a framework for solving all feasible combinations of input and output variables.

The total number of combinations of unknowns m out of the total number of variables n equals

$$C(m,n) = C(n-m,n) = n!/(m!(n-m)!).$$

The number of feasible combinations is smaller because there are almost always some combinations that lead to overdeterminacy (inconsistency) in one part of the model and to underdeterminacy in another part. For instance, all the combinations of input variables which contain all variables appearing in any particular equation, are infeasible.

The **third law** of TK (and a source of misunderstanding of historic proportions):

A TK model can be solved for a certain set of unknowns by the Direct Solver if the reduced VEC-matrix can be triangulized by reshuffling of rows and columns. Every diagonal element points to a variable and an equation resolvable in that variable.

This is like $E=mc^2$, which doesn't need to be on top of the minds of ordinary people all the time, but it holds and is good to know and use when planning interstellar travel.

Variable-Equation-Coincidence (VEC) matrices are a way of visualizing what's going on in TK, much like flowcharts are used in procedural programming. People should use flowcharts and VEC matrices when in trouble. There is one specific difference. Flowcharts reflect an obvious and intuitive flow of information like water flowing from Minnesota to the Gulf of Mexico. In contrast, VEC matrices reflect TK's multidirectional workings.

UTS uses VEC matrices when dealing with all kinds of tech support calls and projects, big and small. There is a VB wizard which automates the entire process, showing which equation is solved for which variable and how many times the Direct Solver has to scan through the Rule Sheet to solve the model. The wizard will also suggest which variables to guess.

The Appendix includes all three of these VEC matrices for the model FRUSTUM.TKW. The real use of VEC matrices commonly occurs in conjunction with subject-related interpretation, on-line experimentation, development, and verification during model design, auditing and trouble-shooting. The exercise described in the previous paragraph can be performed with the VB wizard. Current efforts at UTS are aimed at simplifying this process for users on all platforms.

The **fourth law** of TK (a source of trouble for the Direct Solver and a reason for having the Iterative Solver):

Simultaneous equations break down the propagation of solution by the Direct Solver.

Simultaneousness (yes, there is such word in Webster) of a group of linear or nonlinear equations means that these equations apply to a group of unknown variables *simultaneously* or at the same time, and there is no way of solving them for one unknown after another. In other or TK words, their VEC matrix cannot be triangulized.

Here is a trivial example: The father is five times and the mother is four times older than the daughter (that is, $5 * D = F$ and $4 * D = M$); the father just graduated from the kindergarten when the mother was born ($F - M = 6$).

Eq-ns:	1	2	3
<i>D</i>	x	x	

Eq-ns:	1	2	3
M	x		x
F		x	x

The VEC matrix cannot be triangulized and pressing F9 leaves the Output fields blank. One has to assign any guess value to any of the unknowns; pressing F9 then invokes the Iterative Solver which returns the solution $D=6$, $M=24$ and $F=30$.

This law applies to a single equation as well. Suppose one needs a segment of a circle where the segment perimeter is k -times larger than the chord, i.e.

$$r \cdot \varphi = k \cdot 2 \cdot r \cdot \sin(\varphi/2)$$

The TK rule, after canceling the radius r , may be written as $\mathbf{phi=2*k*sin(phi/2)}$. If phi is known, the corresponding VEC matrix

Eq-ns:	1
k	x

is naturally triangulized and the Direct Solver can solve the equation for k . If k is known and phi is not, the VEC matrix

Eq-ns:	1
phi	#

is equally simple, but it is not triangulized. The symbol # is used to show that the i th variable appears in j th equation, but the equation cannot be resolved for that variable by any sequence of direct and inverse operations. In our example, the Direct Solver cannot extract the value of phi from the above equation, and the Iterative Solver must be used for solving the one-rule model.

The **fifth law** of TK (which may be of great help and is largely ignored):

The simultaneous equations for TK's Iterative Solver to be concerned with are those left after resolving the triangulized head and removing the reverse-triangulized tail of the model.

Strictly speaking, any system of n independent equations in n unknowns represents a set of n simultaneously applied equality constraints. A naive approach is to guess the values of the n unknowns and use some iterative algorithm which would manipulate these values until all n constraints are satisfied with a desired accuracy. In the FRUSTUM model with $n=15$, this would amount to determining 15 reasonable guesses and searching for the solution satisfying 15 constraints in a 15-dimensional space.

A less naive (but still grossly inefficient) way is to “presolve” the system and reduce the set of simultaneous equations a little by evaluating those unknowns that are defined by simple formulas with the unknown variables on the left and evaluable expressions on the right-hand side of the equal sign respectively. Then, in order of increasing sophistication come methods which, depending on the distribution of given and unknown variables, use specific routes of symbolic

manipulation to broaden the presolvable part of the system and reduce the number of simultaneous equations.

TK's technique of solution propagation automatically lets the Direct Solver progress up to the point where there are no single equations resolvable in a single unknown. In other words, the Direct Solver triangulizes the VEC matrix as far as it can. It is like going through the equations or the columns of the VEC matrix one after another and if there is a lone "x", circle it and check off the entire column (since the equation is solved), and entire row (since the variable is no longer unknown). The columns and rows with encircled "x's" constitute the *triangulized head*.

We can apply the reverse process to the rest of the VEC matrix (hence the term *reversed triangulization*). We take one row after another, and, if there is a lone "x", circle it and also check off the entire row (since the variable does not appear in any other unchecked equation and is of no use anymore) and the entire column (since the equation is solved). The columns and rows with encircled x's should be placed in the lower right corner of the VEC matrix; they constitute the *triangulized tail*. The tail represents those equations and variables which can be solved only after the body of simultaneous equations is solved.

This trick may not be easy to follow. Here is the rationale: There must be a last variable evaluated at the end of the solution process. This variable cannot appear in any other equation because there are no other equations to be solved. Also, all other variables in the last equation must be evaluated because otherwise the last variable could not be evaluated from that equation. However, the evaluation of the last variable from the last equation is equivalent to the situation whereby the last variable and last equation would have never existed. Then we could apply the same consideration and treatment to the second last variable and equation, and so on.

A partially triangulized VEC matrix of the FRUSTUM model in the Appendix reflects a problem with the given values of *th*, *AC* and *r1*. It shows the *head* and *tail*, and the *body* of the simultaneous system sandwiched between the two, consisting of only three rows (variables *s1,r1,h1*) and columns (equations 1,2,4). As may be concluded from the 3x3 VEC matrix of the simultaneous system, any one of the three unknown variables needs to be assigned a guess value in order to let the Iterative Solver solve the problem.

In problems requiring the use of the Iterative Solver, the head or the tail or both may be missing (trivial examples: the D-M-F and the circle-segment problem in the commentary to the fourth law of TK above). TK Solver does not take advantage of the possible 3-way split of the model when applying the Iterative Solver. Instead, it rushes through the whole model, processing every rule at every iteration step. This does not matter in small and medium-sized models and may be remedied at a model-design level when dealing with large models using conditional function calls, and/or the built-in function SOLVED().

The **sixth law** of TK (offering light at the end of the tunnel):

A set of simultaneous equations can always be solved by the Direct Solver, after assigning input values to some of the unknowns and dealing with ensuing inconsistencies, by editing error terms into affected rules.

Strictly speaking, the error terms can be always obtained, but the solution may be produced only if it exists; more about that in the commentary to the thirteenth law.

The purpose of TK computation, of course, is not to produce error terms, but to produce problem solutions by means of model resolution. The importance of error terms rests in the ability of TK's Iterative Solver to bring them gradually close enough to zero during the repeated launching of the Direct Solver and adjustment of the extra input values (also called guess values). Once that is accomplished, the equations hold, the rules become satisfied and the problem solved.

The Iterative Solver uses a modified Newton-Raphson procedure for modifying the guess values towards obtaining the solution. It is a brute force method which, at every step, examines local changes of error terms with respect to changes in guess values, and shoots. If the search space is curved and broken, it may overshoot and diverge. One can avoid this by moving the targets (error terms) gradually rather than setting them to zero right away.

In the majority of practical situations one marks the variables with extra input values as guess variables, and lets TK generate the error terms and bring them down to zero automatically.

When it comes to solving systems of linear equations, TK's Iterative Solver works with absolute reliability and predictability, independently of guess values. These are directly mimicked by the Newton-Raphson procedure and are usually solved within a single iteration step.

The **seventh law** of TK (which could save people a lot of trouble):

There is always a minimum number of unknown variables to be assigned input (guess) values in order to make the set of simultaneous equations resolvable by the Direct Solver.

The last paragraph of the commentary to the fifth law already addressed the technicalities of finding the alternatives with a minimum number of guess variables. So, we will concentrate here on the broader issue of the *reasons* for reducing the number of guess variables.

The fewer the extra input values or guess variables, the less worry about the values securing safe and fast convergence to the solution, and the faster the iteration steps (the time taken by each step is proportional to the number of guess variables, with some parts of the iteration algorithm depending on *square* of this number).

The potential to identify and use the minimum number of guess variables (and the ease of doing so) is one of the greatest advantages of TK. This advantage goes far beyond the solution speed-up. Here are words from the popular 'Numerical Recipes' (Press et al., 1992): "We make an extreme but wholly defensible, statement: There are *no* good, general methods for solving systems of more than one nonlinear equation. Furthermore, it is not hard to see why (very likely)

there *never will be* any good, general methods.” The authors then go on about how hard it is to work with two guess variables and two error terms. You imagine each error term $err1$ and $err2$ to be a function of two variables, say, x and y . Both functions form separate surfaces that penetrate the x,y plane along separate lines or sets of lines. The solutions exist in all points where the lines $err1=0$ and $err2=0$ cross (or nowhere if they never cross). These points may be hard to find, but they at least may be easily described and imagined. “For problems in more than two dimensions,” the authors continue, “we need to find points mutually common to N unrelated zero-contour hypersurfaces, each of dimension $N-1$.”

Well, don’t yawn yet, and let us finish the quotation, although the rest of it will be more relevant to the 13th (and the last) TK law than to the present one. “You see that root finding becomes virtually impossible without insight! You will have to use additional information, specific to your particular problem, to answer such basic questions as, ‘Do I expect a unique solution?’ and ‘Approximately where?’”.

TK offers a two-pronged strategy in attacking the problems: On one hand, it helps to substantially reduce the number of guess variables and dimensionality of the search space (compared to alternative techniques) and abate the troubles outlined above. It does this by letting you apply the technique of quick propagation of solution through the system of rules/equations (the Direct Solver) without touching the equations. On the other hand, it lets you handle the search (i.e. the repetitive calls of the Direct Solver) in the reduced search space automatically by a robust built-in trial and error mechanism (the Iterative Solver).

The **eighth law** of TK (the most underused one):

The number of guess variables may be reduced, or the use of the Iterative Solver avoided altogether for some output variables, by presolving the subsystems of simultaneous equations symbolically and adding redundant equations to the set of rules, or by using the techniques of local root finding or iteration.

The first frustum problem in the Appendix (angle of opening, height and radius of base given) was one of the 111 directly resolvable problems in the original model with 18 variables and 15 independent equations. There are, in total, 816 combinations of three input variables out of 18. If we ignore 96 infeasible combinations, there still remains $816-96-111=609$ combinations of inputs and outputs which would have to be solved with help of the Iterative Solver. It’s no wonder that the next problem with the given values of angle of opening, base area and curved area of the frustum falls into this category. Several single variables could have been chosen as guesses; we selected $r1$ and TK converged quickly to the desired solution.

In the next section of the Appendix an alternative technique was tried: We took equations 1, 2, and 4, eliminated the variables $s1$ and $r1$, added the resulting redundant (i.e. “not independent”) equation to the model, and were able to solve the second problem by the Direct Solver without worrying about the selection of a guess variable and choice of a guess value.

The new equation links the variables $AC1$, $h1$ and th . We could have felt encouraged to take advantage of the similarity of the top and complete cones, and, without additional algebra effort, add to the model another redundant equation linking the variables $AC2, h2$ and th . That would

increase the number of directly resolvable input/output combinations further to 178. We will return to the FRUSTUM model with some more radical suggestions in the commentary to the next, 9th law of TK.

Pre-solving linear equations

There may be some spot on the Rule Sheet (or two remote spots) with the rules

$$a*x - c = g*y \quad ; \quad b*y - d = h*x$$

TK sails smoothly through these rules when there is only one unknown to evaluate in each. If both x and y are unknown, the Direct Solver gets stuck on the simultaneity of two linear equations in x and y . If the model is to be used frequently to solve for x and y , it would be inefficient to use the Iterative Solver for resolving the impasse each time. Even the use of the First Guess (default guess) feature would be wasteful and might complicate the use of the model as a whole. The remedy: presolve the system of equations algebraically and add the results to the model. Thanks to the TK solution propagation technique, the pre-solution doesn't need to be complete (i.e. you don't need to derive and add to the model the formulas for $x=f1(a,b,c,d,g,h)$ and $y=f2(a,b,c,d,g,h)$). It is enough to take the subset of n simultaneous equations in n unknowns and eliminate $n-1$ unknowns in the simplest way, making the VEC matrix triangulizable. In the above case, eliminating y and adding the resulting equation

$$x*(a-g*h/b) = d*g/b + c$$

will do the job. The trick may be applied to larger subsystems of linear equations, say up to 4-5, depending on the sparsity of the equivalent matrix (i.e. the number of unknowns appearing in each of the equations and how hard it may be to eliminate what needs to be eliminated). Also, it is not strictly limited to linear equations. You may have product equations

$$x^a / c = y^g \quad ; \quad y^b / d = x^h$$

that, by applying logarithmic transformation, will turn to linear equations similar to the ones above. This also yields a similar redundant equation,

$$\ln(x)*(a-g*h/b) = \ln(d)*g/b + \ln(c)$$

that can make the model resolvable by the Direct Solver. In this case, we avoid the possibility of the Iterative Solver straying into the negative x or y area and causing an Exponentiation error.

Pre-solving polynomial equations

Another technique for saving the Iterative Solver for better things: Identifying polynomial equations in your model and solving them by tools in the ROOTS subdirectory of the TK Library. Sometimes the polynomial equations appear explicitly in the model and sometimes they don't. Take, for instance, the following three equations:

$$a/(x-b) = y*z \quad ; \quad c/(y-d) = z*x \quad ; \quad e/(z-f) = x*y$$

These equations would require guessing two variables out of three (x,y,z), with only a moderate success rate. It is not immediately obvious that there is an equivalent cubic equation

$$\begin{aligned}
\mathbf{A} \cdot \mathbf{x}^3 + \mathbf{B} \cdot \mathbf{x}^2 + \mathbf{C} \cdot \mathbf{x} + \mathbf{D} &= 0, \quad \text{where} \quad \mathbf{A} = \mathbf{a} \cdot \mathbf{d} \cdot \mathbf{f} \\
\mathbf{B} &= \mathbf{a} \cdot (\mathbf{c} + \mathbf{e} - \mathbf{a} - \mathbf{b} \cdot \mathbf{d} \cdot \mathbf{f}) - \mathbf{c} \cdot \mathbf{e} \\
\mathbf{C} &= \mathbf{b} \cdot (\mathbf{c} \cdot (2 \cdot \mathbf{e} - \mathbf{a}) - \mathbf{a} \cdot \mathbf{e}) \\
\mathbf{D} &= \mathbf{b}^2 \cdot \mathbf{c} \cdot \mathbf{e}
\end{aligned}$$

The tool CUBICG.TKT can be used for computing all three roots of equations of this form. Similar equations can be written for y and z . The derivation of the cubic equation is somewhat involved and can be done best with help of one of the computer algebra programs.

There is an easier to derive alternative which reduces the number of guesses to just 1. Divide the 1st equation by the 2nd; this way z is eliminated, and the resulting equation may be written in terms of two unknowns, x, y : $\mathbf{y} \cdot (\mathbf{x} \cdot \mathbf{a} - \mathbf{x} \cdot \mathbf{c} + \mathbf{b} \cdot \mathbf{c}) = \mathbf{d} \cdot \mathbf{x} \cdot \mathbf{a}$. Using this as a redundant equation reduces the number of guess variables to one and enhances the odds of quickly finding solutions.

Identifying polynomial equation(s) in your model and using the Library tools for their solutions, analytical for up to 4th order and numerical (Bairstow) for higher orders, requires doing something about multiple and/or complex roots. It is a blessing in disguise: on one hand it is extra worry and work, but on the other hand you can get more out of your model and you can prevent getting into a bigger trouble by missing an important (or even the only meaningful) solution, or prevent a futile search for a real solution when none exists. (More about this in the commentary to the thirteenth law of TK.)

Local iteration techniques

Occasionally, you may wish to avoid using the Iterative Solver for solving a small subset of simultaneous equations or even a single equation that is unsolvable by the Direct Solver. Let's consider, for example, the Colebrook formula for flow in pipes:

$$\frac{1}{\sqrt{f}} = -0.869 \cdot \ln \left(\frac{e/D}{3.7} + \frac{2.523}{R\sqrt{f}} \right)$$

and let's assume that f needs to be evaluated now and then for known e , D and R . It is not a polynomial and there is no way to derive an equivalent equation directly resolvable in f . There are several iterative algorithms for single guess variables in the TK Library (and one, Newton's method, for two guess variables). We can use Newton's method with numeric differentiation (tool NEWTON-N.TKT) after trying the Iterative Solver and making sure that it works fine starting with an initial guess from the simpler Blasius formula:

$$f = 0.316 / R^{1/4}$$

The tool is set up for using the built-in function APPLY to call the function to be iterated on but it also can be modified so that it does the whole job on its own:

```

===== PROCEDURE FUNCTION: Colebrook =====
Comment:          f by Colebrook formula, Newton's method
Input Variables:  e,D,R
Output Variables: f
S Statement-----
  dx = 1e-6
  a = .869
  b = e/D/3.7
  c = 2.53/R
  x = R^.125/sqrt(.316)      ; Blasius formula, f = .316/R^(1/4)
Loop:
  y = x + a*ln(b+c*x)        ; substitution x=1/sqrt(f) used
  if abs(y) < 1e-8 goto End
  x = x - y/((x*(1+dx)+a*ln(b+c*x*(1+dx)) - y) / (x*dx))
goto Loop
End: f = 1/x^2

```

Evaluation of f can now be initiated from anywhere in the Rule Sheet or Function Subsheets by referring to the function **Colebrook**. To preserve full backsolvability and to prevent re-evaluation of an already known f , the incorporation of local iteration in this and other situations may be accomplished by a rule such as

```

if known(`f) then 1/sqrt(f)=-.869*ln(e/D/3.7+2.523/R/sqrt(f))
                                     else f=Colebrook(e,D,R)

```

The measures stipulated in this law may be a matter of necessity or convenience. In the latter case it is a question of a compromise between ease of design and ease of use which, in turn, may be affected by the purpose of the model and its prevalent usage.

The **ninth law** of TK (which should have been introduced much earlier, sorry):

The invertibility of the operations and functions is the name of the game with TK.

The proverbial chicken-egg problem: this point deserves to be, and could have been made into, “the second law” which would lead to reversing the order of other items. This is to say that the invertibility of operations and functions is a cornerstone of TK. It falls closely behind the multidirectional solution propagation and the ultimate backsolving capability, and also behind the reduction of guesses when using the Iterative Solver.

The TK Direct Solver naturally takes advantage of subtraction, division and raising to power of $1/a$ to be, respectively, inverse operations to addition, multiplication and raising to power of a . The same concept holds for the selection and implementation of built-in functions. For instance, $y=\exp(x)$ is equivalent to saying $x=\ln(y)$ and $y=\operatorname{erf}(x)$ may be solved for both unknown y and x . Some built-in functions (e.g. **sin, cos, tan, sqrt**) offer “imperfect inverses” or so-called principal values. You may have a rule $y=\sin(x)$ which for $x=8$ returns $y=.989$; you assign the latter as input, solve for x , and obtain 1.425; this is because \sin is a periodic function and it returns an inverse from the interval $-\pi/2 \leq x \leq \pi/2$.

The inverses of the above trigonometric functions are also imperfect from the viewpoint of floating-point arithmetic, which is a partial reason for having the functions **sind, cosd, tand**

and others with arguments in degrees. For instance, `asind(180)` returns correct 0, whereas the resulting value of `asin(pi())`, 1.225E-16, is loaded with a small, but often unacceptable error. Some built-in functions (e.g. `abs`, `int` or `Bessel`) are by definition non-invertible. They halt the propagation of solution in situations when their arguments are unknown. `Bessel` is a smooth function with continuous derivatives so that you can use guessing and iteration for backsolving. That, unfortunately, is not the case with `abs`, `int`, `mod` and other non-invertible functions. You are advised to check the TK help utility or the manual on the invertibility of the built-in functions you wish to use.

Invertibility of custom-designed (also called user-defined) functions

As far as the invertibility of custom-designed rule functions and the invertibility or resolvability of every rule and the model as a whole are concerned, you are on your own.

Consider this equation linking a, b, c, x, y :

$$\frac{x}{x-a} = b \cdot y + \frac{c^2 - y^2}{c+y}$$

If you transcribe it into a TK rule literally as `x/(x-a) = b*y + (c^2-y^2)/(c+y)` it will be directly resolvable in `a` and `b` and nothing else. A little streamlining of the left- and right-hand side yields an equivalent rule `1/(1-a/x) = y*(b-1) + c`, which is directly resolvable with respect to each of the five variables.

TK procedure functions are in essence procedural programs or subroutines with fixed sets of input and output variables and with precisely defined flow of computation. That makes the concept of inversion and invertibility inapplicable. The backsolving of procedure functions can be achieved only through guessing and iteration or by (rarely justified) design of specialized procedures with built-in backsolving features.

It is shown in the Appendix how to develop a fully invertible function for direct evaluation of the elements of a cone with acute opening angle, given input values any two variables out of seven. Two calls to such functions (one for cut-off cone and one for the complete cone) may replace the equivalent 10 rules in the original model FRUSTUM. As a consequence, the model looks better and, more importantly, the Direct Solver will be able to handle 426 combinations of inputs and outputs instead of the original 111 combinations (out of the 816 total or 719 feasible ones).

How do we represent procedure functions and rule functions in VEC matrices so that their role in the model resolution process and their invertibility (or lack of it) is reflected properly? A procedure function is executed only when the values of all the parameter and input variables are known and it yields values of all output variables. Consequently, a call to a procedure function is equivalent to as many equations as there are output variables, with an “x” for each in “its own” equation column. In each of these columns, all the cells corresponding to the parameter or input variables should be marked with “#” meaning that the procedure is not resolvable or invertible with respect to these variables, but their values are needed for producing a value of variable marked by “x”.

It is different with rule functions which can be resolved for any argument or result variables and which can be entered many times during the solution process. Rule functions should be considered and should appear in VEC matrices as an extension of the Rule Sheet.

The **tenth law** of TK (the most obvious):

The lexical analysis (with VEC matrices and other tools) helps us to use TK efficiently. It may assist with, but it never substitutes for, the necessary mathematical and subject-related analysis.

There are always three interrelated but distinct aspects of TK model design and use. The “laws” deal primarily with the question of what to expect (and what not to) from TK as a declarative computer language, how to interpret its behavior, and how to use its features efficiently.

There are also mathematical aspects. The approach to the problems on hand, the resulting model design and the model use must be mathematically sound. We approached the concept of mathematical soundness in the first law but there are many other points, too many to summarize in a discussion of this sort. They include the whole of mathematics, including numerical analysis. Issues like what happens when some variables acquire zero value or fall into a singularity trap and how to prevent it; multiplicity of solutions; iterative solutions straying into an area where the arguments of arcsine or arccosine are larger than 1 or arguments of square roots are smaller than 0; a total absence of a real solution; devastating propagation and amplification of errors, and other liabilities of floating point arithmetic including such minute details as preferring binary to decadic fractions.

An aside: Floating point arithmetic blues

The end of the previous paragraph touches on occasional confusing manifestations of the use of the 64-bit IEEE floating-point arithmetic (f.p.a.) standards used for numeric computations in TK. To follow on this issue, compare the middle five values resulting from the “Add step” option of a TK list fill starting at -3.0 through 3.0, step 0.6 and 0.75 or, respectively, hexadecimal 3FE3333333333333 and nicely rounded 3FE8:

```

===== TABLE: test =====
Element
      Increment:0.6= "3FE3333333333333
      You expect   You get
. . . . .
4      -1.2        -1.2
5      -.6         -.60000000000000001
6      0           -1.110223024625157E-16
7      .6          .5999999999999999
8      1.2         1.2
. . . . .
      Increment:0.75="3FE8
      You expect and get
. . . . .
4      -1.5
5      -.75
6      0
7      .75
8      1.5
. . . . .

```

There are 11 values in the first series and 9 in the second. More importantly, the hexadecimal/binary approximation of 0.6 causes the floating-point noise here and there in the 1st list, whereas the exact binary representation of 0.75 produces well-rounded-off values in the 2nd list.

The TK Library model IEEE64 and tool HEXADEC.TKT in the MISCUTIL subdirectory offers a workbench for investigating potential f.p.a. mischief. The TK real numbers are represented in the computer by 16-digit hexadecimal numbers. The first 3 digits represent the sign and the exponent, and the last 13 digits or 52 bits are the mantissa. There is, actually, an additional 53rd leading bit in the mantissa, which is always assumed to be 1 and which always counts but never shows. Without delving into the meaning of the first 3 hexadecimal digits, we just say that the exponent for both .6 and .75 is -1. According to these definitions and conventions the 3FE3333333333333 in hexadecimal represents precisely the following decadic number:

$$0.59999999999999997779553950749686919152736663818359375$$

You can see that it is almost 0.6, but not quite, there is an absolute difference of 2.22...e-17. So, why not increase it by adding 1 to the hexadecimal mantissa, i.e. to work with 3FE3333333333334? The decimal equivalent would be

$$0.600000000000000088817841970012523233890533447265625$$

which is even worse; with an absolute difference of 8.88...e-17. So, if we insist on 0.6, we have to buy and live with the previous representation as the best possible. If we work with simple binary fractions, such as $0.5 = 2^{-1} = 3FE$ or $0.75 = 2^{-1} + 2^{-2} = 3FE8$ or even $0.6015625 = 2^{-1} + 2^{-4} + 2^{-5} + 2^{-7} = 3FE34$, all the multiples will be dead on target.

The numeric computations are approximate by definition, and TK's internal 15-16 significant digits are plenty to support the usual required accuracy of no more than 5-6 digits. Still, the f.p.a. problems of two kinds are quite close to the surface of everyday TK usage. One kind irritates us by being evidently imprecise, such as the above listfill example. The second kind deals with error accumulation and propagation. It is an aside in the context of this commentary, although the remedies usually depend on the discussed mathematical insight and analysis, and require measures of mathematical nature at both model design and usage stages.

How much mathematics does TK require and provide?

TK Solver is rarely used for purely mathematical exercises. The TK Solver designers and publishers count on the user's or application developer's input of the subject-related knowledge and offer a framework for molding together the knowledge of TK, mathematics, and life.

Finally, a drastic example of the difference between lexical and mathematical analysis and the need to complement one by the other. Let us consider two TK models (or parts of larger models):

A: $x^{.7} + y + z^{1.2} = 2$ $x^{1.4} + y^2 + z^{2.4} = 4$ $x^{2.1} + y^3 + z^{3.6} = 6$	B: $x^{.7} + y + z^{1.2} = 2$ $x^{1.4} + y^2 + z^{2.4} = 4$ $x^{2.1} + y^3 + z^{3.6} = 6$
--	--

The VEC matrices of both models are the same:

Eq-ns:	1	2	3
x	x	x	x
y	x	x	x
z	x	x	x

This implies that the problems may be solved by the Iterative Solver given workable guess values of any two variables out of three unknowns. Actually, this conclusion is already based on a trivial analysis carried out subconsciously by every self-respecting TK user without drawing the reduced VEC matrix which, with y and z assigned guess values, would look as follows:

Eq-ns:	1	2	3
<i>x</i>	x	x	x
<i>err2</i>		x	
<i>err3</i>			x

(In other words, *x* would be evaluated from the 1st equation and the error terms, for the Iterative Solver to work on, from the 2nd and 3rd equations; not a big improvement compared to conventional methods, but still better than working with three guesses for all *x,y,z*).

This all works for model A which consists of 3 independent equations and where carefully chosen guess values $y=-1$ and $z=1$ lead to the solution $x=2.301921$, $y=-.514869$ and $z=1.762589$. The model B with ^ operators replaced by * cannot produce any solution at all. The reason: the equations are not independent. The 2nd equation is twice the 1st, and the 3rd is a sum of the 1st and 2nd. This piece of information cannot come but from mathematical analysis.

The **eleventh law** of TK (speaks against abuse of list/block solving):

List solving or block solving is a way of performing a series of simple solutions automatically for a series of values of input variable(s); it should not be launched before simple solving works satisfactorily.

This law is of the “do’s and don’ts” type. There is no surprising news, no complicated analyses and no hard-to-follow examples. It addresses a real problem, though. TK novices usually learn about list solving soon after starting with the product, most often in connection with plotting simple function graphs. They conclude from this first experience that listsolving is something to be routinely applied to any problems dealing with a multitude of situations.

Here is a narrower and better prescription for remembering and applying listsolving: It is a technique for automating the repetitious instances of simple solving. This recommendation implies the fulfillment of two conditions. Firstly, the model must work well in the simple solving mode. Secondly, there must be a good chance that it is applied to a genuinely repetitious task, i.e. that it will work well and return credible responses in the whole listsolving interval.

A drastic example of an abuse of listsolving: using it for transformation of datapoints for curve fitting. A better way: use a procedure function with a for-next loop triggered by the Examine command or calling the function from the beginning of the Rule Sheet.

Block solving is probably as much underused as listsolving is abused. Being able to list solve (i.e. block solve) without a list associated with an input variable may come in handy. Another useful trick: zooming down to a place of interest by creating a gap (inserting rows) in a table with input/output listsolving data, using listfill for generating missing input data, and block solving over the gap to fill it.

Exploratory and creative list/block solving

The last paragraph provides a glimpse at a “creative” use of list solving and block solving. Again, if you are interested in repetitious solutions (as is usually the case), you should first make sure that your model works fine in the simple solution model. Try to prevent failures along the way, such as division by zero, logs of negative numbers, etc. Finally, you should not expect from listsolving what it cannot provide (e.g. changing the problem set-up in the middle of a listsolving sequence).

On the other hand, there is nothing dreadful about failing listsolving instances -- if the failure is a part of useful information about the model or the investigated phenomenon. An example: The problem used at the end of the commentary to the 10th law was naturally contrived. We used list solving to find out that the solution of the problem A exists only if the right-hand side constants defined as c , $2*c$ and $3*c$ use the value of c between 1.2 and 2.8 or, more precisely, between 1.209571 and 2.693976.

By the way, the last two values are the results of mathematical analysis of a TK problem. The Iterative Solver used for finding the limits of c returned the numerical results correct up to 13-14 significant digits. It is different from a true mathematical analysis of a closed-form solution, if such a solution exists. It does exist in the given case where the limiting values of c have been determined to be precisely

$$C_{1,2} = \frac{9 - \sqrt{9 + 6 \cdot \sqrt[3]{3} + 4 \cdot \sqrt[3]{9}} \pm \sqrt{18 - 6 \cdot \sqrt[3]{3} + 4 \cdot \sqrt[3]{9} + \frac{6}{\sqrt{9 + 6 \cdot \sqrt[3]{3} + 4 \cdot \sqrt[3]{9}}}}{2}$$

Pretty useful piece of information, huh? Unless you relate more readily to the 6-, 7- or more-digit number shown in the previous paragraph. (This has little to do with list/block solving but you still can step back and, using listsolving, check if the interval of c 's is correct.)

The **twelfth law** of TK (usually learned too late, i.e. after getting burned):

Respect and take advantage of the difference between values of variables (that are initialized at the beginning and cannot be overwritten during the solution process) and values of list elements (no automatic initialization, free to overwrite any time).

This is mainly a reminder. There is the concept of *status* of TK variables, which may be given, guessed (a special kind of given), evaluated, known (i.e. given or evaluated), or unevaluated.

At the beginning of the direct solution process all variables, with the exception of those given and guessed, are initialized by turning their status to “unevaluated”. During the solution process only the unevaluated variables are allowed to acquire new values and become evaluated. All known variables (i.e. given and evaluated) are protected against changes or rewrites. This is an essential feature of the TK concept. It keeps the known variable values available for substitutions in any equations and for checking that the equations hold and rules are satisfied. This is quite unlike the procedural languages (including TK procedure functions) which don't need to worry about

holding equations. These languages let the assignment statements rewrite any variables anywhere anytime.

Beside the above arrangement supporting the declarative nature, TK may need some information protected from initialization at the beginning of the solution process and unprotected during solution. Lists and list elements serve this purpose among other roles they play in TK. The existence of a list is indicated by the appearance of its name in the List Sheet. Any list element may hold a value or may be blank. There is no concept of status of a list or list element.

A simple example

You wish to count the number of times you launch the Direct Solver after loading a model. You create a list `count` and keep the counter in the 1st element of that list. You save the model with value zero in its 1st element. That way you secure initialization of the counter whenever the model is loaded. The last thing to do is to write into the 1st row of the Rule Sheet a rule `place('count,1)='count[1]+1`. The `place` function is a TK-specific built-in function which takes whatever value evaluated comes out of the rule and places it into the desired element of the desired list. The `place` function is not invertible but it also can appear anywhere in the rule. `'count[1]=place('count,1)-1` would also work fine, it's just harder to read.

Alternatively, you could use as the 1st rule `place('count,1)=elt('count,1,0)+1`, and forget about building the list beforehand and seeding it with zero. `elt` or `element` is another TK built-in function that returns `'count[1]` in our case or the third argument, 0, if `'count[1]` is blank. Another part of the trick: placing or assigning a value to a non-existing list causes creation of that list in the List Sheet.

Finally, on this example, could we say `'count[1]='count[1]+1`? We could, and the Direct Solver would take it for its face value and return an "Inconsistent" error message. The construct `'count[1]='count[1]+1` would do its job if it appears in the procedure function named, say, `Increment`, and if the first rule in the Rule Sheet is `call Increment()`.

The above example shows several aspects of interfacing the scalar-variable-oriented declarative component of TK with free-wheeling stateless lists and their elements, one of these aspects being the non-invertibility of the `place` and `elt` functions and the `listname[arg]` construct. Related examples, techniques and tricks are as useful as they are countless and well beyond the scope of this material. Many may be found in the TK Library and documentation.

In defense of the variable/list double-dichotomy

Here we put a twist to this commentary different from all others and take a defensive stance. Why complicate things by mixing two criteria, one being the data dimensionality (scalars, lists or vectors, matrices or arrays), and the other the presence or absence of status in the context of declarative or procedural computation? If we stick to the scalar vs. list and status-defined vs. stateless distinction, there would be $2 \times 2 = 4$ combinations, which the present TK reduces to two. We miss the other two occasionally during TK programming and usage. What would be the cost of having them? The main item would be spending much more time on design and modification of TK models and declaring to which category every variable and list belongs. That would require more complicated sheet structure, interface, command structure, built-in function offering, and so on.

Comparing the alternatives, the current distinction between status-bound variables and status-free lists works remarkably well. Once there is a variable in the Variable Sheet (and Rule Function Subsheets as Variable Sheet extensions), TK automatically keeps track of its status. Once there is a list anywhere, TK treats it as a status-free object. This covers the two most significant elements of the 2x2 matrix. Bridging the absence of the other two elements is not hard for users who understand these questions in principle.

The variables and lists meet in a somewhat peculiar way in listsolving and blocksolving which was the subject of the previous eleventh law. The lists associated with variables for the purpose of list/block solving may be thought of as acquiring the variables' status, but it may be better to consider them plain vanilla lists with element values that are retrieved from or written to the lists in some orderly fashion. There is nothing sacred or mystic about the connection between the equally named variable ABC and list ABC except that you can associate the latter with the former by a single keystroke, L, in the Status field of the ABC row in the Variable Sheet. You can often get additional mileage by using Variable Subsheets for changing the names of lists associated with certain variables.

Let us try a little puzzle on you. You have a model with three variables, a, b, c , and a single rule $a=b*c$. You bring up the List Subsheet for c and you see the values 3,4,6,8,12. You press F10 and you see 8,6,4,3,2. You press F10 again and 3,4,6,8,12 comes back. Another F10 and you see 8,6,4,3,2 again. And so on. What is going on and what is assigned to what? (No other rule or function is present).

The **thirteenth law** of TK (which is mostly a warning, and partly an encouragement):

Beware of multiple solutions

That is, check whether the solution which TK returned was the one you were looking for, and get more or all solutions to choose from if desirable.

The multiplicity of solutions satisfying given rules comes from the mathematical nature of the problems at hand. Polynomial equations of odd order (including linear equations) have at least one real solution. Then the number of solutions goes up until it reaches the order of the polynomial. So, a 5th order polynomial may have 1, 3 or 5 real roots depending on how many times the graph of the function crosses the x-axis; an 8th order polynomial may have 0, 2, 4, 6 or 8 real roots. There may be an infinite number of roots of equations containing trigonometric functions (the simplest example is an equation $\sin(1/x)=0$; the density of roots increases with the decreasing absolute value of x). If the graph of the error function (defined as the difference between the left- and right-hand expression in the equation) doesn't cross the x-axis at all, then there no solution to be found. The situation is more complicated in cases of two or more simultaneous equations with two or more unknowns. We will return to this subject at the end of this commentary.

It is true that the convergence of the Iterative Solver to this or that root depends on the choice of the guess value, however, it is not a simple question of proximity or distance. It is more a question of Newton's interval or the interval of guess values converging to a particular solution. It's a matter of appreciation for the workings of the Iterative Solver and the way the tangencies and tangential planes are "drawn" and used for improving on the guess values during the iterative solution.

Generally, the multirootedness or multiplicity of solutions complicate TK model design and usage. We could use as an example the FRUSTUM model, which, in some situations, one would expect to yield two solutions, one for acute and one for obtuse frustum of cone. We mention modestly at several spots that the examples in the Appendix deal specifically with an acute frustum. The Appendix would have to be expanded in order to cover the acute/obtuse issue.

Let's look instead at the fully invertible function `cone1`, made into a separate model CONE1.TKW. The model covers most of the combinations of input and output variables, yet there are anomalies. For instance, for given slant $s=6.2$ and height $h=5$ we find the opening angle $th=72.5$ deg, radius $r=3.67$ and volume $V=70.37$. The next assignment may be to keep given slant, fix the volume V at its present value, and find th, h, r for an obtuse cone. The model as is cannot do that even by iteration because it is fully invertible, it automatically uses principal values when backsolving for the arguments of trigonometric functions, and it always slips into the acute cone domain. We would have to cancel the redundant equations in order to do the assignment, guess approximately half of the current height or twice the radius, solve iteratively, and arrive at $th=143.6$ deg, $h=1.94$ and $r=5.89$.

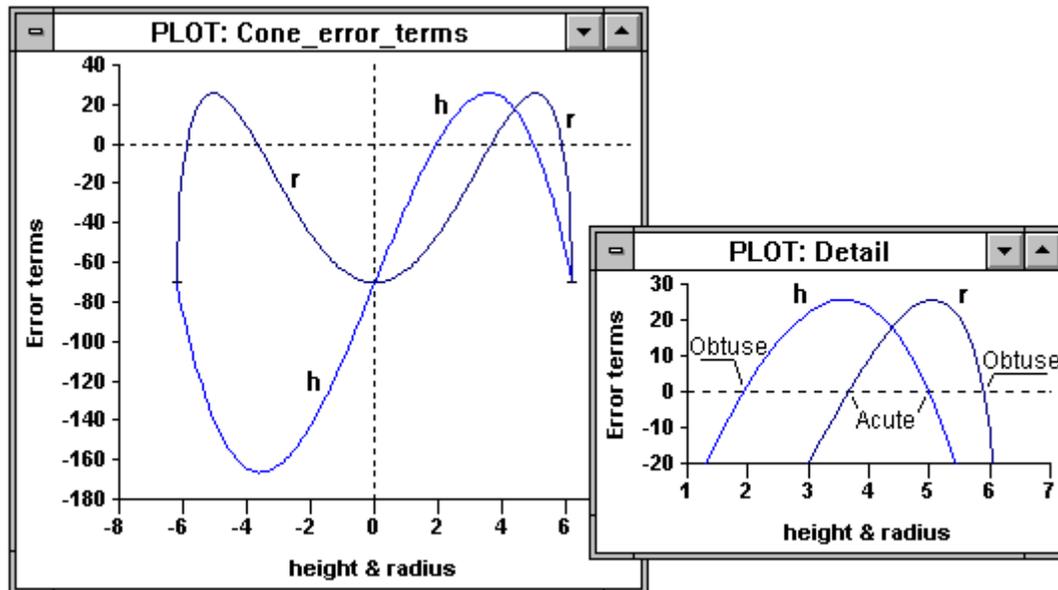
This looks like a defeat of all the effort which went into the development of fully or highly invertible functions and models. Well, a conflict and discontinuity in TK work is to be expected if the redundant equations were developed and the invertibility beefed up, based on the assumption of a single-root solution, and the resulting function or model is used for computing "the other root(s)". Remedy: Take into account the existence, possibility or desirability of multiple solutions from the very beginning of the function/model development. Then, either protect the function/model from the attempts to generate the other solutions (by checkpoints, error messages, etc.) or treat the other solutions separately by means of different variable names, set of rules or functions, etc. We owe you a practical example for this in the Appendix, perhaps later. A semi-practical example is in the TK Library; it is the classic QUADR.TKW in the ROOTS subdirectory. It interprets a quadratic equation as a set of two equations linking five variables, three coefficients and two roots, or, alternatively, real and imaginary part of conjugate complex roots; given the values of any three of the five variables, the model evaluates the remaining two.

Struggle for an obtuse cone

Back to the cone model for a moment. Regardless of the invertibility hiccup (or, more precisely, the failure to secure a smooth sailing through the multiroot problem), the user is required to apply a good deal of subject-specific knowledge (such as the fact that going from an acute to obtuse cone at constant volume means reducing the height and increasing the base radius) in order to get the job done. Common sense and/or VEC matrix analysis show that after canceling redundant equations either h or r can be used equally well as guess variables for iterative solution of the obtuse cone geometry when s and V are given. Equally well? Not quite. The Iterative Solver was very sensitive to the choice of guess value of r . It converged only from the guess values less than 6.2 and greater than 5.5, otherwise it returned the "Exponentiation: Argument error". The Iterative Solver was much more generous with h as a guess variable. It converged from a guess value anywhere between 0 (or even -2) and 3. What is at play here?

The graphs of error terms as functions of the guess values of h and r offer an explanation. They are combined in one plot, but they resulted from separate list solving for $err=f(h)$ and $err=f(r)$. In both cases the error term was attached to the 5th equation where the Iterative Solver put it when it was running. The definition interval for the argument of both functions is from -6.2 through

+6.2. The reason: at given $s=6.2$, an absolute value of h or r larger than that would lead to an attempt to take square root of negative argument when solving the 1st equation.



The graph of $err=f(r)$ shows that there are not two but four solutions, two with positive and two with negative values of r, th, AC (the latter make some mathematical, but, obviously, no physical sense). It is clear from the graphs that it is much harder for the Iterative Solver to catch the “obtuse point” on the curve r than h .

The cone model made multiroot-invertible, after all

The excessive demand on the TK, mathematical and subject-related (i.e. cone-related) insight and skills to go through the above analysis prompted me to try a more user-friendly alternative. The result: an example of bringing the fully invertible model now called CONE2 up to the task of handling the two-root solution by adding a variable *cone* with values ‘A (for acute) or ‘O (for obtuse). Here is an additional 6th rule and a revamped set of redundant equations:

```

if th<=90 then cone = 'A else cone = 'O
AC * c2osd(th/2) = pi() * h^2 ; Redundant equations:
s2cd(90-th/2) * AC^3 = 9*pi() * V^2
if cone='A then s2cd(th/2)*s^3*pi()=3*V
if cone='O then c2sd(90-th/2)*s^3*pi()=3*V

```

If the value of *cone* is not given or differs from ‘A or ‘O, and it is needed to break the ambiguity (e.g. when s, V are given), the model behaves as if underdefined, i.e. doesn’t yield a solution until ‘A or ‘O is specified. If th can be evaluated without prior knowledge of *cone*, the ‘A or ‘O will appear in the output field. If the value of *cone* is given and if it disagree with the evaluated one, the 6th equation is marked as inconsistent. One of the redundant equations reads differently for obtuse cones; that required an introduction of another list function, $f(x) = \cos^2 x \cdot \sin(x)$. An aside: If multiroot situations are handled by separate variables and separate subsets of equations, VEC matrices may help to streamline the model/function development and solution. Including the variable *cone* into a VEC matrix for our example may not help much; also, if there are conditional rules, there must be separate VEC matrices for different cases.

The changes look small, but the effort required for the multiroot generalization of the fully invertible model CONE2 was not negligible. It wasn't a developer-friendly situation. Take these final results as our personal tribute to the coneheads of the world.

Truly interesting finale to the multiroot discourse

...or a reward to those who read or browsed the presented material up to this point. Plotting a graph of the error term as a function of a guess value in the case of a single guess variable, and using it for selection of an initial guess is a fairly simple concept. We used it in connection with finding an obtuse cone with given values of slant and volume. But how do we investigate multiroot problems requiring two or more guesses? Here is an example: a simple set of two 3rd order polynomial equations in two unknowns, x and y :

```
===== RULE SHEET =====
S Rule-----
* x^3 + x^2*y - x*y^2 - y^3 + x^2 + x*y - y^2 - x + y + 1 = 0
* x^3 - x^2*y + x*y^2 - y^3 + x^2 - x*y + y^2 - x + y - 1 = 0
```

It is possible, in this special case, to derive a simpler redundant equation by subtracting the 2nd equation from the 1st:

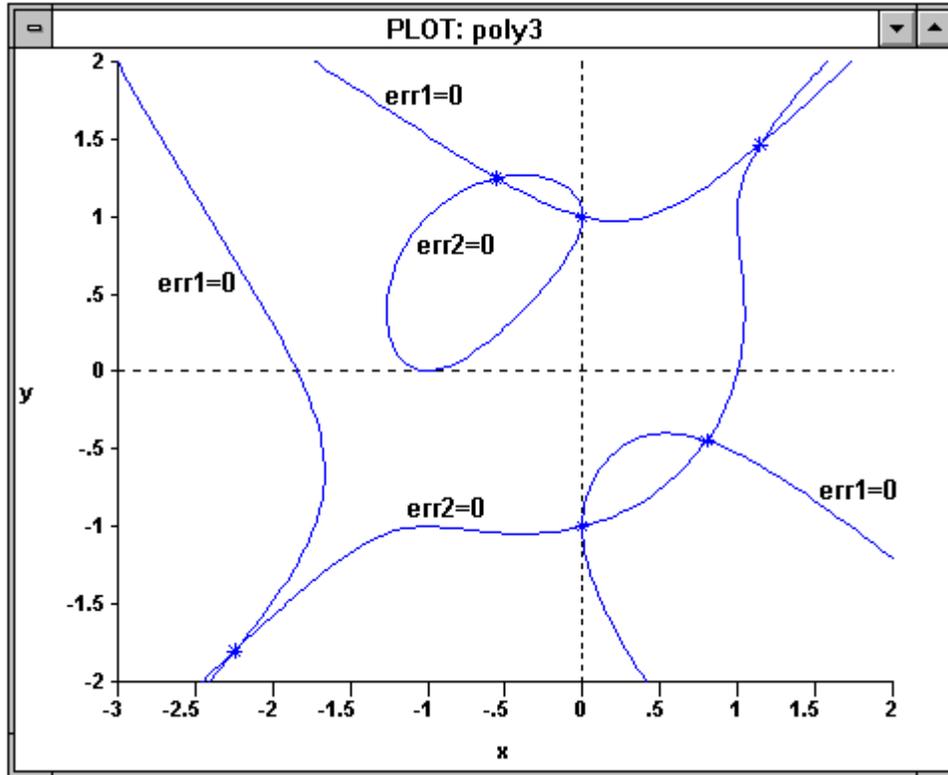
$$x^2*y + x*y*(1-y) - y^2 + 1 = 0$$

but it will not help to solve it. Common sense and the VEC matrix

Eq-ns:	1	2	3
x	#	#	#
y	#	#	#

shows that both x and y have to be guessed and the error terms attached to the original two equations turned close enough to zeros (we will ignore the useless 3rd equation). The fact that they are polynomial equations points to the existence of multiple solutions. Where should we look for them?

In a single-guess case we choose an interval of values for the guess variable and evaluate the error term, looking for intersections of the x -axis. Similarly, in a two-guess case we have to decide in what region of the plane of guess variables x,y we should evaluate both error terms, find the lines along which $err1=f1(x,y)=0$ and $err2=f2(x,y)=0$, and look for intersections of those lines, because the coordinates of the intersection points are the values of x,y for which the equations hold. This long sentence says it all. You might split it to make it easier to follow or you can look at the following plot and see the location of six real roots in the interval $-3 \leq x \leq 2$ and $-2 \leq y \leq 2$; there are also two complex roots and one degenerate (the total order of the equations is 9).



It was easy to use the approximate location of the intersection points with the Iterative Solver to obtain the numeric values of all six real roots:

x:	0	0	-2.246980	0.801938	1.147899	-0.554958
y:	-1	1	-1.801938	-0.445042	1.465571	1.246980

We hope to develop a utility for generating plots such as these for the benefit of the equation-solving public.

One can imagine generalizing this kind of survey for problems with three guess variables, although the implementation and showing the results on screen and hard copy wouldn't be easy. The generalization for four and more guess variables cannot even be imagined except as mathematical abstraction.