SRC TR 85-36

Adapting The Spreadsheet To Engineering Problems

by

M.E. Palmer, M.G. Pecht and J.V. Horan

TR-85-86

Vol. 4 No 2
Sept. 1985

Adapting the Spreadsheet to Engineering Problems

Lotus 1-2-3 can do useful work in numerical analysis. Experience with programming in a spreadsheet's language will also show what it lacks, and where—hopefully—software developers will make improvements.

M. E. Palmer III
Assistant Professor
M. G. Pecht
Assistant Professor
J. V. Horan
Research Assistant
Department of Mechanical Engineering
University of Maryland
College Park, Maryland

The popularity of financial spreadsheet programs for microcomputers is well known. The newer spreadsheet packages take advantage of many advances in microcomputer technology, especially growth in the processor's memory-address space. They have extended abilities to represent data graphically and some have programming languages included.

Using operations common to numerical analysis, spreadsheets can also be used to solve engineering problems. To demonstrate this, Lotus 1–2-3 and a Compaq microcomputer have been used as a model spreadsheet and hardware environment.

Editor's Note: This article is adapted from the authors' presentation during the 1985 ASME International Computers in Engineering Conference in Boston this August.

The Spreadsheet As A Programming Environment

A spreadsheet is composed of a two-dimensional matrix of entries, called cells, which can contain information of three types: text, numbers, or formulas. Numbers can be thought of as a formula with a constant value. The formula in a cell can contain references to other cells or even to itself. The value of the formula is inferred from these references. Text either has a zero value or an error value. Cells are specified by giving their coordinates in the spreadsheet matrix, generally using sequential letters for the horizontal coordinate and numbers for the vertical coordinate. The cell *B*3 thus represents the second column and the third row of the matrix. The formulas in spreadsheet cells are simple algebraic equations, with cell coordinates used as algebraic variables.

The cell references are of two types, relative and absolute. An absolute reference is to a specific cell; a relative reference is a location relative to the location of the formula. These modes of reference can be mixed in the horizontal or vertical coordinates. The prefix $ is used to denote an absolute reference, and the absence of a prefix

cell $B1$ contents — {calc}/$XI(A1<A2)$~/X
cell $B2$ contents — '/XG $B1$~

assuming that cell $A1$ contains the iterate error and cell $A2$ contains the stopping criteria.

A spreadsheet can be used to solve nonlinear equations in more than one variable. The fixed-point and Newton algorithms are easily extended to multiple dimensions and the secant method can also be extended. One such extension is the Broyden algorithm. Except for the fixed-point method, the multiple dimension nonlinear methods require at least the ability to perform matrix multiplications (Broyden's algorithm) and usually matrix inversion or an equivalent linear equation solution. Most spreadsheets do not have these capabilities built in. The multidimensional fixed-point algorithm is the most suitable and easily programmed method for use with spreadsheets, but convergence is not always assured, even for arbitrarily precise initial estimates. When convergence occurs it is generally slow since the algorithm is only first-order-convergent. Nonetheless, many multidimensional rooting problems encountered in engineering practice can be solved using a fixed-point formulation.

Solution of linear equations. The ability to solve systems of linear equations is the second fundamental operation in numerical analysis. Two classes of techniques used to solve linear equations are direct and iterative. Examples of direct methods are Gaussian and Gauss-Jordan elimination and various reduction methods such as LU decomposition and Crout's method. Iterative methods include Gauss-Jacobi, Gauss-Siedel, and relaxation methods.

Solving arbitrary linear systems of equations by direct methods is difficult because the spreadsheet cannot efficiently index values into defined ranges and the keyboard macro language is primitive. The authors have written a macro for Gaussian elimination (without any pivoting strategies) for a system of three unknowns. The resulting macro was approximately 100 commands long and execution was very slow. The construction of this algorithm was closer to assembly language programming than higher level expression. For more realistic algorithms with pivoting and arbitrary matrix sizes, the resulting keyboard macro would be clumsy and quite slow.

These observations also apply to frequently used operations such as matrix multiplication, evaluation of determinants, matrix inversion and transposition. Inner products and some matrix norms are easily evaluated using @SUM, @MAX, and other built-in functions. Inclusion of range (matrix) inversion, range multiplication, and range transposition functions among the predefined operations would expand the power of commercially available spreadsheets.

Spreadsheets have all the advantages of other interpretive environments, with the additional advantage of immediate and global feedback. They also have the disadvantages.

Iterative solutions of linear equations are much easier to program with spreadsheets because these are really just applications of the multidimensional fixed-point algorithm. The major difficulty remains the determination of convergence, and the need for vector multiplications (inner products) for some of these algorithms tends to make the cell or macro coding tedious unless the matrix is sparse. Iterative algorithms are mainly used to solve linear equations resulting from the discretization of elliptic field problems by finite difference or finite element techniques. Here, the structure of the resulting equations makes the cell coding quite simple.

The ability to solve tridiagonal systems of linear equations [3] is especially important because these systems appear frequently when solving second-order boundary value problems using finite difference or finite element methods. If the natural order of calculation is selected, this algorithm can be programmed directly in the spreadsheet cells. Natural order calculation means that cell values are calculated each time they are mentioned in a cell formula, although the results are the same as recursive-descent calculations. Table I is a sample spreadsheet showing the formulas for solution of tridiagonal equations. The formulation is simple and extremely quick, with one hundred equations being solved in just under four seconds. The equations to be solved are (for $i = 1, \ldots N$)

$$A_i x_{i-1} + B_i x_i + C_i x = D_i$$

The unknown x 's are calculated using i in the following algorithm:

$$\alpha_1 = D_1/B_1 \quad \beta_1 = -C_1/B_1$$
$$\alpha_i = (D_i - \alpha_{i-1} A_i)/(A_i\beta_{i-1} + B_i)$$
$$i = 1, \ldots N$$
$$\beta_i = -C_i/(A_i\beta_{i-1} + B_i)$$
$$x_N = \alpha_N$$
$$x_i = \alpha_i + \beta_i x_{i+1} \quad i=N-1, \ldots 1$$

As can be seen in Table I, the vectors A, B, C, D, α, β, and x are assigned columns in the spreadsheet, with the appropriate formulas entered for the α, β, and x columns.

It should be noted that in Table I all equations between the first and last are relocated duplicates of each other. Therefore once one of the rows is entered the others can be obtained by use of the range copy facilities.

Interpolation and approximation. Interpolation formulas are important to construct if analytic computations are to be performed on tabular data. In general, interpolation formulas are generated by forcing some interpolating function to ex-

cell $B1$ contents — {calc}/$XI(A1<A2)$~/X
cell $B2$ contents — '/XG $B1$~

assuming that cell $A1$ contains the iterate error and cell $A2$ contains the stopping criteria.

A spreadsheet can be used to solve nonlinear equations in more than one variable. The fixed-point and Newton algorithms are easily extended to multiple dimensions and the secant method can also be extended. One such extension is the Broyden algorithm. Except for the fixed-point method, the multiple dimension nonlinear methods require at least the ability to perform matrix multiplications (Broyden's algorithm) and usually matrix inversion or an equivalent linear equation solution. Most spreadsheets do not have these capabilities built in. The multidimensional fixed-point algorithm is the most suitable and easily programmed method for use with spreadsheets, but convergence is not always assured, even for arbitrarily precise initial estimates. When convergence occurs it is generally slow since the algorithm is only first-order-convergent. Nonetheless, many multidimensional rooting problems encountered in engineering practice can be solved using a fixed-point formulation.

Solution of linear equations. The ability to solve systems of linear equations is the second fundamental operation in numerical analysis. Two classes of techniques used to solve linear equations are direct and iterative. Examples of direct methods are Gaussian and Gauss-Jordan elimination and various reduction methods such as LU decomposition and Crout's method. Iterative methods include Gauss-Jacobi, Gauss-Siedel, and relaxation methods.

Solving arbitrary linear systems of equations by direct methods is difficult because the spreadsheet cannot efficiently index values into defined ranges and the keyboard macro language is primitive. The authors have written a macro for Gaussian elimination (without any pivoting strategies) for a system of three unknowns. The resulting macro was approximately 100 commands long and execution was very slow. The construction of this algorithm was closer to assembly language programming than higher level expression. For more realistic algorithms with pivoting and arbitrary matrix sizes, the resulting keyboard macro would be clumsy and quite slow.

These observations also apply to frequently used operations such as matrix multiplication, evaluation of determinants, matrix inversion and transposition. Inner products and some matrix norms are easily evaluated using @SUM, @MAX, and other built-in functions. Inclusion of range (matrix) inversion, range multiplication, and range transposition functions among the predefined operations would expand the power of commercially available spreadsheets.

Spreadsheets have all the advantages of other interpretive environments, with the additional advantage of immediate and global feedback. They also have the disadvantages.

Iterative solutions of linear equations are much easier to program with spreadsheets because these are really just applications of the multidimensional fixed-point algorithm. The major difficulty remains the determination of convergence, and the need for vector multiplications (inner products) for some of these algorithms tends to make the cell or macro coding tedious unless the matrix is sparse. Iterative algorithms are mainly used to solve linear equations resulting from the discretization of elliptic field problems by finite difference or finite element techniques. Here, the structure of the resulting equations makes the cell coding quite simple.

The ability to solve tridiagonal systems of linear equations [3] is especially important because these systems appear frequently when solving second-order boundary value problems using finite difference or finite element methods. If the natural order of calculation is selected, this algorithm can be programmed directly in the spreadsheet cells. Natural order calculation means that cell values are calculated each time they are mentioned in a cell formula, although the results are the same as recursive-descent calculations. Table I is a sample spreadsheet showing the formulas for solution of tridiagonal equations. The formulation is simple and extremely quick, with one hundred equations being solved in just under four seconds. The equations to be solved are (for $i = 1, \ldots N$)

$$A_i x_{i-1} + B_i x_i + C_i x = D_i$$

The unknown x 's are calculated using i in the following algorithm:

$$\alpha_1 = D_1 / B_1 \quad \beta_1 = -C_1 / B_1$$
$$\alpha_i = (D_i - \alpha_{i-1} A_i)/(A_i\beta_{i-1} + B_i)$$
$$i = 1, \ldots N$$
$$\beta_i = -C_i/(A_i\beta_{i-1} + B_i)$$
$$x_N = \alpha_N$$
$$x_i = \alpha_i + \beta_i x_{i+1} \quad i=N-1, \ldots 1$$

As can be seen in Table I, the vectors A, B, C, D, α, β, and x are assigned columns in the spreadsheet, with the appropriate formulas entered for the α, β, and x columns.

It should be noted that in Table I all equations between the first and last are relocated duplicates of each other. Therefore once one of the rows is entered the others can be obtained by use of the range copy facilities.

Interpolation and approximation. Interpolation formulas are important to construct if analytic computations are to be performed on tabular data. In general, interpolation formulas are generated by forcing some interpolating function to exactly represent a table of data. The unknown coefficients in the interpolating function are then obtained by solving the set of linear (or nonlinear) equations derived by substitution of the tabular data. As noted previously, the resulting equations are difficult to solve with a spreadsheet. Simple interpolations such as polynomials and trigonometrics are therefore difficult to evaluate. Alternative methods of obtaining interpolating polynomials, such as Newton's divided difference interpolant, are simple to implement on a spreadsheet, the major difficulty being the calculation of the interpolating polynomial values from the computed coefficients.

One form of interpolation which can be implemented simply using a spreadsheet is the cubic spline [3]. The linear equations generated when determining a cubic spline are tridiagonal and thus easily solved using the formulation shown in Table I. The resulting interpolated values can then be obtained by programming a cell with a cubic evaluation routine and by using the spreadsheet's data sorting functions. This routine provides a smooth function to represent a relatively sparse set of data. The spreadsheet can then graph the curve. Without this function, it would only draw straight lines between data points.

One can easily approximate integrals by programming cells of the spreadsheet. The trapezoid rule is particularly useful because the weights of the quadrature formula are all constant. More accurate formulas, such as Simpson's rule, are still possible, but entering the variable weights makes the resulting programming somewhat tedious. Gaussian quadrature formulas are particularly simple to implement, assuming that a table of weights and abscissas is available to the spreadsheet. Derivative approximation is a matter of direct calculation if finite difference formulas are used. Otherwise, cubic splines can be implemented and differentiated.

Least-squares fitting of data. A frequently applied technique in numerical analysis is the fitting of experimental data using least squares [4]. Linear least-squares fitting is equivalent to solving an overspecified set of linear equations and requires matrix transposition, multiplication, and inversion. The spreadsheet's deficiencies in these tasks have already been noted. The fitting of a straight line through data, which is probably the most frequently used least-square technique, or the fitting of any two-parameter linear or linearizable model, can be performed by directly programming the necessary linear algebra. The spreadsheet's statistical functions for average and standard deviation can also simplify the process. The formulas for the coefficients in the model $y = a + b\,x$, given a set of data (x_i, y_i), can be obtained from the equations

A spreadsheet can be created and executed much more quickly than writing and running a program in, for instance, Basic.

$$a = ((Sx^2 + E(x)^2)E(y) - E(xy)E(x))/Sx^2$$
$$b = (E(xy) - E(x)E(y))/Sx^2$$

where $E(.)$ is the expectation function (@2AVG in spreadsheet notation) and Sx is the standard deviation (@STD) of the x range. If the spreadsheet also includes a correlation function, $E(xy)$ can be obtained from its use. Otherwise a range containing the arrays x, y must be programmed.

For the more general case of linear least-squares fitting, the inclusion of a generalized matrix inverse along the lines of the APL domino function [5] would be of fundamental importance. This function operates as matrix division, giving a solution to simultaneous equations either in the classical sense or in a least-square sense if the system is overdetermined. Such a generalized function could also be used as a range inversion function.

For nonlinear, least-squares problems, Gauss-Newton algorithms could be employed if a spreadsheet could invert matrices. The matrix elements could be programmed into cells in a range, and during every iteration the matrix would be automatically updated. Levenberg-Marquardt enhancements and more complex algorithms, such as Gauss-Newton or Broyden's methods, could be added fairly easily.

Initial value problems for ordinary differential equations. Marching problems, such as initial value problems for ordinary differential equations, can be programmed directly in a spreadsheet. Explicit formulas, such as the Euler method and the Runge-Kutta methods [3], can be defined, and the formulas copied. Implicit methods, such as the improved Euler method [3], can also be programmed directly, and the entire spreadsheet recalculated iteratively. However, this is very inefficient, because while values near the initial value are being iterated, so are those near the desired endpoint, using poor estimates for previous steps. This inefficiency can be overcome by using a keyboard macro, such as the one presented for nonlinear equations, to complete the iteration on a given time level and then to move the cursor down to the next level and repeat. Simultaneous differential equations cause no difficulties. The authors have found that if explicit numerical methods are used, problems of this type can be solved much faster using a spreadsheet than using a conventional programming language, such as interpreted Basic.

Boundary value problems for ordinary differential equations. There are three major techniques used to solve boundary value problems for

Table I
Spreadsheet to Solve Tridiagonal Equations

	A	B	C	D	E	F	G	H
1								
2								
3	i	A(1)	B(1)	C(1)	D(1)	Alpha(1)	Beta(1)	X(1)
4	1	0	2	−1	1	+E4/C4	−D4/C4	+F4+G4*H5
5	1+A4	−1	2	−1	0	(E5−F4*B5)/(B5*G4+C5)	−D5/(B5*G4+C5)	+F5+G5*H6
6	1+A5	−1	2	−1	0	(E6−F5*B6)/(B6*G5+C6)	−D6/(B6*G5+C6)	+F6+G6*H7
7	1+A6	−1	2	−1	0	(E7−F6*B7)/(B7*G6+C7)	−D7/(B7*G6+C7)	+F7+G7*H8
8	1+A7	−1	2	−1	0	(E8−F7*B8)/(B8*G7+C8)	−D8/(B8*G7+C8)	+F8+G8*H9
9	1+A8	−1	2	−1	0	(E9−F8*B9)/(B9*G8+C9)	−D9/(B9*G8+C9)	+F9+G9*H10
10	1+A9	−1	2	−1	0	(E10−F9*B10)/(B10*G9+C10)	−D10/(B10*G9+C10)	+F10+G10*H11
11	1+A10	−1	2	−1	0	(E11−F10*B11)/(B11*G10+C11)	−D11/(B11*G10+C11)	+F11+G11*H12
12	1+A11	−1	2	−1	0	(E12−F11*B12)/(B12*G11+C12)	−D12/(B12*G11+C12)	+F12+G12*H13
13	1+A12	−1	2	0	0	(E13−F12*B13)/(B13*G12+C13)	−D13/(B13*G12+C13)	+F13

ordinary differential equations. For highly nonlinear boundary value problems, the conventional method is called the shooting technique [3]. Basically, this is equivalent to transforming the boundary value problem into an initial value problem with unknown initial values. The proper initial values are then determined to cause the solution to yield the given boundary values. This is a two-step process, the first being the solution of an initial value problem and the second being an update of the unknown initial values. The first step is easily implemented on a spreadsheet. The second step is equivalent to solving a set of nonlinear equations for the unknown initial values. The difficulties of using a spreadsheet in the multivariate case have already been noted; the single variable case is easily solved using the secant algorithm.

The other two methods of solving boundary value problems involve discretization by either finite difference or finite element methods. These are usually applied to quasilinear problems and produce a system of quasilinear equations with a bandwidth equal to the order of the differential equation plus one. For second order ordinary differential equations, the bandwidth is three, so the resulting equations are tridiagonal and solvable by techniques already, with nonlinearities handled by iteration.

Solution of partial differential equations. Two distinct classes of partial differential equations will be discussed here: field or elliptic problems, and the class of parabolic and hyperbolic equations with a time-like independent variable.

Elliptic field problems in rectangular or intersecting rectangular regions are ideally suited for solution using a spreadsheet. In such a region, a rectangular, not necessarily uniform, grid can be set up, and the governing equation discretized using either finite difference or finite element techniques. In such a discretization, the resulting equations have a highly geometrical connectivity. The finite difference and certain finite element solutions of Laplace's equation on a uniform grid yield the simple geometric statement that each node value is the average of its four closest neighbors. In a spreadsheet, the program becomes exactly that statement, and only need be entered at one node, then copied into all other interior nodes, each node being a cell. The appropriate constants or equations are then developed and entered into the cells that represent the boundary nodes. The simplicity of this representation has been noted elsewhere [1]. The spreadsheet is set on iteration and the calculations started. On nonuniform grids, an array of horizontal and vertical coordinate values can be kept in a row and column outside the region and these values used in the interior equations. These interior equations are relative copies of one fundamental equation. The region geometry and boundary values are easily changed. Relaxation can be added to the discretized equations to speed convergence. For three-dimensional problems in blocked regions, similar arguments can be made. The major difference in implementation is that the region must be divided into planes, with each plane representing a different depth. The equations can then be programmed in the same way as for the two-dimensional case.

For regions that are more irregular than piecewise rectangular regions the same difficulties are encountered whether one is using a spreadsheet or conventional programming. A rectangular mesh is placed on as much of the region as possible and programming of this subregion proceeds as described. The complementary part of the region is then programmed essentially point by point, with special equations to represent the local geometric nonuniformities. This argument is more appropriate to finite difference than to finite element techniques. It would be difficult to assemble finite element "stiffness" equations in the conventional manner by means of spreadsheet programming.

Parabolic or hyperbolic differential equations can be solved by using a combination of the techniques used for elliptic equations and those used for initial value problems for ordinary differential equations. This is quite possible if the equations contain at most two space-like independent variables, the time-like variable being treated like the depth in the elliptic three-dimensional case. All of the necessary techniques have been described.

Miscellaneous methods and applications. Because it is important for modeling, most spreadsheets handle statistical analysis with ease. However, the distribution functions provided with spreadsheets are not complete for engineering applications. The addition of statistical distributions such as Weibull, Poisson, F, Chi-square and Student-t would be helpful. In addition, probability functions associated with these distributions, as well as the normal distribution, should be included.

Most applications that use Monte Carlo or probabilistic methods [1] can be programmed on a spreadsheet but are slow to execute even in a compiled environment on mainframe computers because their "convergence" rate is only on the order of the square root of the number of random experiments. In a spreadsheet, where all computations are interpreted, these methods tend to be intolerably slow and reasonable solutions are impractical.

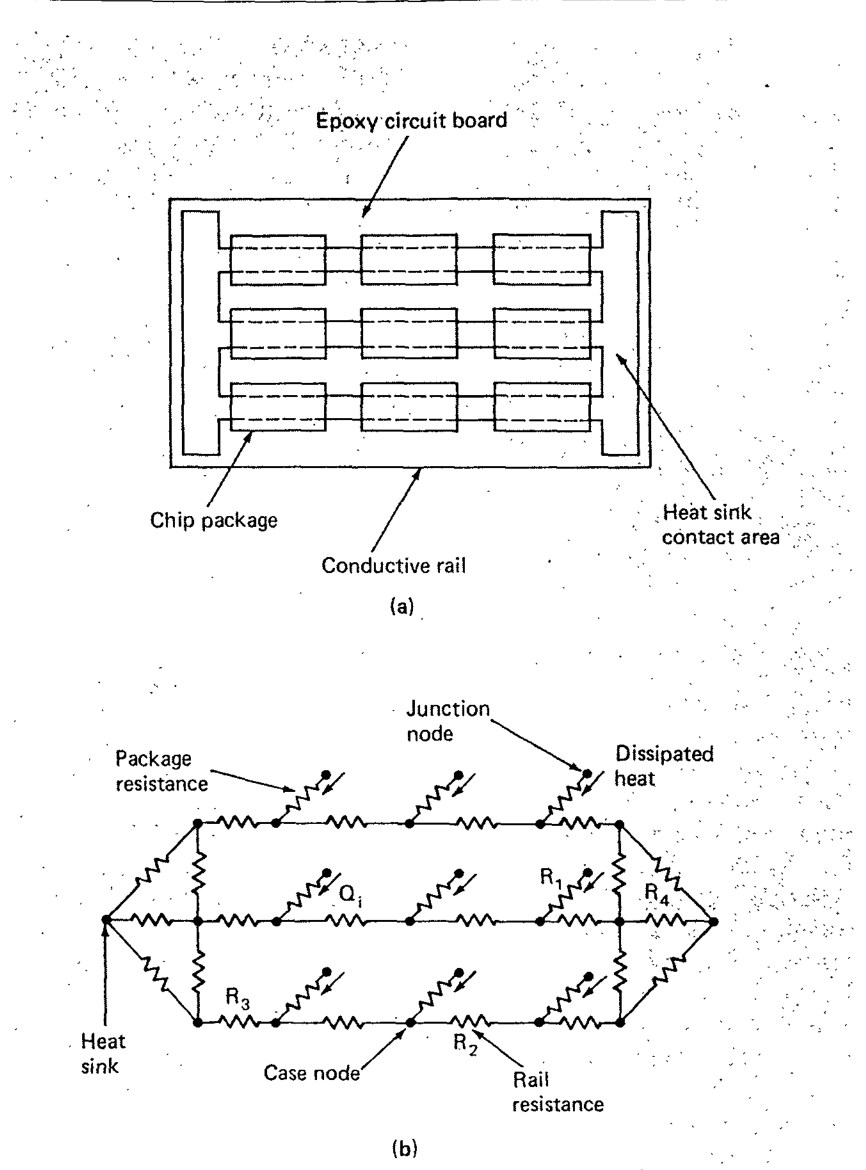

Fig. 1 Heat transfer in printed circuit boards. (a) Top view of conduction-cooled PCB; (b) thermal model.

An interesting teaching application of spreadsheets [6] involves the simulation of digital logic functions found on common integrated circuit chips.

Spreadsheet in Action: Heat Transfer in Printed Circuit Boards

In certain avionic applications, the environment demands that cooling of printed circuit boards (PCBs) be accomplished through thermal conduction. To provide the necessary cooling capacity a system of thermally conducting rails is placed in contact with the chip carrying packages (DIPs or flatpacks) to allow thermal diffusion to heat sinks placed at the edges of the board [5,8]. In order to evaluate the reliability of the resulting PCB, one must perform a thermal analysis. This is typically done using a resistance network model for the heat transfer in the packages and along the rails [7], with the networks having on the order of two hundred to one thousand nodes. The thermal networks are then evaluated using large computer codes, such as Ansys or Sindas.

While developing a design support system for a chip package placement algorithm [7], the au-

Table II
Spreadsheet for Thermal Analysis Of Conduction Cooled PCBs

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1														
2														
3														
4														
5														
6	Chip type --- enter chip numbers								Chip type		Dissipation		Case conductance	
7											watts		watts/deg. C	
8														
9	0	5	2	3	0				0		0		0.05	
10	0	1	2	3	0				1		0.0625		0.05	
11	0	5	4	2	0				2		0.125		0.05	
12									3		0.1825		0.05	
13									4		0.25		0.05	
14	Junction temperatures (deg. C)								5		0.5		0.05	
15														
16									Rail conductances					
17	0	34.5	26.8	26.7	0				watts/deg. C					
18	0	22.9	24.9	25.8	0									
19	0	34.9	30.1	25.8	0				2		0.12			
20									3		0.75			
21									4		0.5			
22	Case temperatures (deg. C)													
23														
24								Sink temperatures (deg. C)						
25	20.5	24.5	24.3	23.1	20.4									
26	20.4	21.6	22.4	22.1	20.3			Left		20				
27	20.5	24.9	25.1	23.3	20.4			Right		20				
28														
29									Junction temperatures					
30								Maximum		34.9				
31	KBD MACRO \C							Previous		34.9				
32	{calc} {calc} {calc}							Error		0				
33	{calc}/XI(J32<J33)~/XQ							Test		0.01				
34	/XG A33~													

thors had to formulate and test several production rules based upon thermal considerations. Networks with approximately 20 to 50 nodes were tested. These small scale models are easily implemented with a spreadsheet program on a microcomputer. The thermal resistance network is simply a set of linear equations with cells representing nodal temperatures. Conductances and chip energy dissipations are found using lookup tables in the spreadsheet, allowing these parameters to be changed without difficulty. Figure 1 shows a typical test PCB along with a resistance network. The junction temperatures are the temperatures experienced by the chips at the junction with the chip carrier. Case temperatures are those at the interface between the chip package and the underlying rail. The spreadsheet in Table II corresponds to the network in Figure 1. The user enters the chip types into the appropriate range in the spreadsheet matrix and executes the keyboard macro *\C* which controls the convergence of Gauss-Siedel iterations used to solve the equations. As output, the maximum junction temperature is given, as well as the entire temperature field.

Most of the production rules tested involved the switching of chip packages on a board based upon the chip energy dissipation and the package location relative to the heat sinks and the local package temperatures. The effects of applying a rule could be immediately observed by performing cell moves in the chip location matrix. The capability of a set of production rules to provide a reasonable chip package distribution could be tested interactively, with minimal programming overhead. The immediate, visual feedback from the spreadsheet was found to be extremely helpful in the production rule testing.

Recommendations For Future Spreadsheet Designs

For frequently encountered small problems, such as fitting a straight line through data or rooting a nonlinear equation, the spreadsheet provides an excellent alternative programming tool. A spreadsheet can be created and executed much more quickly than writing and running a program in, for instance, Basic. As the problem becomes larger, or requires using multiple techniques as building blocks for a solution, spreadsheets and conventional programming require about the same amount of development time. The spreadsheet is more flexible in the ability to independently test modules. For two-dimensional field problems the spreadsheet is a better environment due to the simplicity of entering the discretized formulas and the explicit display of the underlying geometric concepts.

Simple extensions to current spreadsheets which would make them more competitive with environments such as Basic are:

• **Improved indexing.** Direct access to the elements of a range or matrix would allow keyboard macro programming of many matrix operations. Such a feature was included in Symphony as @INDEX.

• **Expanded matrix operations.** Direct operations on ranges such as transposition and matrix inversion would remove many of the obstacles encountered in the programming of various numerical analysis techniques. Functions for the evaluation of determinants and matrix products would also be useful.

• **Extension of functions.** Although spreadsheets can compute most of the common functions encountered in engineering applications (i.e. sine, square root) they cannot perform more specialized functions as readily as Basic or Fortran can. The user needs access to the internal storage and details of the spreadsheet. A programmer could then write a subroutine or function in Fortran, Pascal, or assembly language that could be called from within the spreadsheet.

• **Improved iterative control.** Most nonlinear numerical analysis involves iteration. For efficiency, computations should only be performed until the required precision is obtained. With current spreadsheets, iterations are performed a predetermined number of times. In addition, a programmer has very little control over the order of computations other than the use of a keyboard macro. These macros tend to be difficult to program, compared to the entering of cell formulas, and their execution is slow. A linear linked list structure could give the programmer control over the order and convergence determination of a set of spreadsheet computations. The list could contain cells or ranges to be iteratively computed, along with a convergence criteria for each list entry. Thus a list such as

*A*1/.001/absolute
*C*2 . . . *D*4/.003/relative

could represent the instructions to calculate the value in cell *A*1 until the absolute change in value is less than .001, then iteratively calculate (using the spreadsheet default recalculation procedure) the values in the range *C*2 . . . *D*4 until the maximum relative change in value is less than .003. This structure could be implemented as new keyboard macro instructions or through an external subroutine.

• **Improved speed of computation.** Probably the major advantage of a conventional programming language such as Basic is the ability to test a program in the interpretive mode and then compile the verified code. An equivalent capability could be provided for spreadsheets in the form of a meta-compiler. Thus a saved spreadsheet could be meta-compiled by an external program. The compilation would not be an exact duplication of function since this would imply that the programmer could change a formula in a cell entry, thus requiring the compiled version to interpret the new formulas and defeating the purpose of compiling. Instead, the compiled version of a spreadsheet would have predefined cells for input, which the programmer could enter values into, and predefined output cells whose values would be displayed. These cells can be defined by using only unprotected cells for input and by using windowing to define the output values. It is useful to note that the speed of the interpreted spreadsheet could be improved by using incremental compilation of the cell formulas. The authors know of at least one spreadsheet which uses this technique. Hardware enhancements such as math coprocessors should also be implemented.

Many interesting engineering problems can be solved with spreadsheet—particularly small problems in numerical analysis. For these types the spreadsheet sometimes offers a quicker solution than a conventional language can. Spreadsheets' range of engineering applications can be greatly broadened if they are improved in the ways discussed here. ■

Acknowledgments

The author's research was partially supported by NSF grant CDR8500108 through the University of Maryland Systems Research Center.

References

1 Hayes, B., "Computer Recreations," *Scientific American*, Vol. 249, No. 1, July 1983, pp. 22–36.
2 Ferziger, J. H., *Numerical Methods for Engineering Application*, New York: John Wiley & Sons, 1981.
3 Burden, R. L., Faires, J. D. and Reynolds, A. C., *Numerical Analysis*, 2nd ed. Boston: Prindle, Weber & Schmidt, 1981.
4 Miller, A. R., *Basic Programs for Scientists and Engineers*, Berkeley: Sybex, 1981.
5 Jenkins, M.A., "Domino: An APL Primitive Function for Matrix Inversion—Its Implementation and Applications," *APL Quote Quad*, Vol. 3, No. 4, February 1972.
6 Cortesi, D. E., *Dr. Dobb's Journal*, No. 83, Sept. 1983, pp. 12–16.
7 Horan, J. V., "Graphical Integration of Thermal Analysis with Computer-Aided PCB Design," M.S. Thesis, University of Maryland, May 1985.
8 Steinberg, D. S., *Cooling Techniques for Electronic Equipment*, New York: Wiley-Interscience, 1980.