STEAM CYCLE ANALYSIS SPECIFICALLY FOR THE MICROCOMPUTER

Dudley J. Benton
Tennessee Valley Authority
Engineering Laboratory
Norris, Tennessee

presented at the
ASME Winter Annual Meeting
December 26-30, 1990
Houston, Texas

ABSTRACT

A steady-state steam cycle model is presented as a practical example of an innovative approach to large nonlinear systems analysis on a microcomputer. Rather than scaling down an existing mainframe code to fit on a PC and being forced to chose between limited versatility or impractical runtimes, the approach described here is to develop the methods which best utilize the strengths of a microcomputer and avoid its weaknesses as much as possible. This same concept could be applied to systems other than steam cycles. Some of the differences between mainframes and microcomputers which significantly impact engineering software are also discussed.

INTRODUCTION

Computers are becoming more and more indispensable as tools of the engineer. Energy systems can be among the more complex applications of computer analysis. Energy systems have been analyzed by computers for the past few decades to some degree--mostly on mainframes. With the introduction and increasing availability of microcomputers having sufficient power to perform technical analyses, a number of mainframe codes have been adapted to run on these smaller machines.

The basic concepts on which these mainframe codes were developed may be ill-suited to microcomputers because of basic differences in their hardware architecture compared to mainframes. The engineer who develops such a code (software) may well understand the analysis which needs to be performed, but not the way in which the particular machine (hardware) will carry out the task.

Engineers often see a computer as a "black box" which dutifully performs calculations as symbolically instructed to do so in FORTRAN. Very few engineers can program in assembler or have any significant understanding of what instructions are generated by a high level language compiler.

Optimum Performance from Available Equipment

The challenge for the software developer is to obtain the optimum performance (speed, user-friendliness, etc.) from the available hardware. Differences in mainframe and microcomputer hardware and various high level language compilers are not insignificant. Ignoring these differences when developing software will not necessarily result in bad programs, but will result in inefficient ones. Depending on optimizing compilers to compensate for the differences is simply ineffective.

Optimized Preprocessing + Interpretive Postprocessing

Nonlinear systems such as steam cycles require iterative solution. The innovation illustrated here is to separate the task into two steps: optimized preprocessing and interpretive postprocessing. Whereas most codes have fixed procedures and accept only data as input, the code required for the second step in this analysis task must accept data and procedure. This task separation removes as much decision making as possible from within the iteration required to solve the nonlinear system.

Because this optimization is done only once per configuration, greater computational effort can be invested in the optimization step which will be recovered many times over in running the second step of the analysis. The ability of the second step program to operate on both data and procedure (i.e., real-time interactive code and data manipulation) greatly expands the available level of user interaction for customizing and debugging.

The Illustration: A Steady-State Steam Cycle Model

In the first step--optimized preprocessing--the paths of influence between each component in the system are analyzed and the optimum manner in which to solve each component is determined. Also in this first step the optimum order in which to combine the components into the ensemble which represents the system and the optimum way in which to store this information in a database is determined. This optimum procedure will be unique to each system and need only be determined once for a given system unless its configuration is changed. The second step--interpretive postprocessing--is to actually solve the system for a prescribed set of conditions or constraints.

Because the resources allocated to analyzing the system are also optimized in the first step, even systems with as many as a thousand components (e.g., a nuclear system with three complete independent strings[1] of heaters, six 2-stage MSRs[2], and 3 series condensers each with a different pressure[3]) can be solved on a PC (20MHZ 80386) in less than 15 minutes. A typical single-string fossil unit requires less than 2 minutes on such a machine. Furthermore, the ability of the second step to interpret and execute FORTRAN-like code enables the engineer to embed existing code for specific components or develop

custom code and even thermodynamic properties for specialized applications.

This concept of subdividing a larger task into optimized preprocessing and interpretive postprocessing steps could certainly be applied to systems other than steam cycles. The present application to steam cycles is simply used as a demonstration that the concept is practical. The innovation illustrated in this steam cycle model is not that of solving simplified equations or idealized configurations in order to make a formidable task more tractable, but of taking a different approach to solving the same equations--an approach specifically designed to utilize the strengths of microcomputers and avoid their weaknesses.

[1] A 2- or 3-string steam system is one having two or three parallel streams of nominally identical feedwater heaters (regenerative heat exchangers) and feedpumps which ideally carry one-half or one-third of the feedwater and extraction steam. Their operation in prototype systems is not identical; however, it is common practice to lump them together into a single stream. In the cited example these are not lumped together, but rather analyzed separately so that the actual operating conditions can be accurately modeled. Much as is the case in solving a parallel pipe network, solving a 3-string steam system requires more than three times as much computational effort as a single stream system.

[2] An MSR, or Moisture Separator Reheater, is a combination moisture remover and tube-in-shell heat exchanger which is used in nuclear powered steam systems to dry the steam before entering the low pressure turbine(s). Their function is to protect the low pressure turbine(s) from droplet impact erosion and vibration at the expense of thermal efficiency. Their inclusion in a steam system greatly increases the numerical "stiffness" and thus computational effort required to solve the system as they are a strong positive feedback device. A 2-stage MSR has both a high temperature and low temperature reheating tube bundle which are supplied with high and low pressure extraction steam respectively. Two-stage MSRs result in greater numerical "stiffness" and more positive feedback than do single-stage MSRs.

[3] Some steam systems are designed with multiple condensers arranged in series such that each successive one receives the cooling water from the previous one. This design results in multiple backpressures. Because many steam cycle codes are unable to handle this arrangement, it is common practice to lump the individual condensers and low pressure turbines into a single composite one. Such a practice, of course, precludes an as-built analysis of the prototypical system. In the cited example the three condensers and low pressure turbines are analyzed individually. Modeling series condensers increases the positive feedback in the system, the resulting numerical "stiffness", and the computational effort.

MATCHING SOFTWARE TO HARDWARE

Processors are simply not generic number crunchers. They do not all perform the same basic tasks with proportional speed and convenience. While there are significant differences between contemporary mainframes of differing design, the differences between mainframes and microcomputers are probably more striking. These differences are not just speed and size. While computer science majors are no doubt well drilled in these differences, engineers may not be.

Benchmarks Can Be Misleading

Benchmarks using supposedly unbiased machine speed comparison programs are in the author's experience essentially useless if not actually misleading. A few examples may serve to illustrate how important understanding the matching of software to hardware can be. In the past few years the author has performed various speed comparisons on a dozen or so mainframes, microcomputers, minicomputers, and workstations. Manufacturers will intentionally not be mentioned in what might be considered a negative context.

One benchmark provided by a manufacturer showed the floating-point speed of their machine to be 50% greater than a competitor's similar machine. This benchmark did not contain any transcendental functions (e.g., arctangent). When transcendental functions were added, the comparison became 2 to 1 in favor of the competitor. The reason for this discrepancy was found to be the absence of hardware transcendental functions in one machine and the presence in the other. If one were interested in transcendentals this would be an important discovery--hopefully made before the purchase. This illustrates how it can be important to understand the hardware before developing the software. In this case the first machine would be much better suited to computing Legendre than Fourier transforms. An optimizing compiler cannot make such a compensation.

Another comparison between two similar machines showed a 3 to 1 advantage with a program requiring many thermodynamic property evaluations (basically polynomial expansions), yet no advantage with a program requiring iterative solution of large matrices. In this case the floating-point processor in the first machine was 3 times as fast as the second. However, the first machine was a hybrid 16-bit processor which needed additional overhead to address more than 64K bytes of code or data; while the second was

a true 32-bit processor having no such difference in addressing overhead for the two programs. In this case the first machine might be much better suited to solving potential fields using boundary elements than finite-differences. Again, an optimizing compiler cannot make such a compensation.

A third comparison was between two FORTRAN compilers generating code to be run on the same microcomputer. The initial tests showed a 4 to 1 advantage in favor of the first compiler. It was subsequently discovered that the first compiler used short integers (16-bit) by default and the second defaulted to long integers (32-bit). When the option was set for the second compiler to use short integers and the test rerun, there was only a negligible difference in the two with a slight advantage in favor of the second. A compiler cannot determine beforehand that integers above 32767 will never be needed and thus instructions to use long integers can be ignored.

More illustrations could be given, but these three are easily repeatable and serve to illustrate the point: there is a definite advantage to understanding the hardware and the compiler before developing software.

Utilizing Strengths/Avoiding Weaknesses

Microprocessors--especially hybrid 8/16-bit ones like Intel's--are quite well suited to character (8-bit) operations. Mainframes and true 32-bit minicomputers and workstations are comparatively less suited to these tasks. In order to perform character operations on most true 32-bit processors it is necessary to either pad (using only 1 of 4 bytes in each word) or mask-and-shift (eliminating unwanted bytes and right justifying). Padding wastes 75% space (for 24 unused bits per word) and increases memory shuffling and page faulting. Mask-and-shift operations at least quadruple the work required for otherwise simple comparisons. An interesting and enlightening discussion of the differences between various computer hardware architectures and optimal code generation in light of these differences is given by Muchnick (1988).

16-bit microprocessors such as Intel's are capable of performing long integer (32-bit) operations, but at a considerable increase in computational expense. Long integer addition and subtraction require approximately three times that of similar short integer operations. Long integer multiplication takes at least 4 times as long as short; and long integer division may take 64 times as long as a similar short integer operation.

The relative speed of floating-point and integer calculations varies considerably with computer architecture. Some machines (e.g., HP-1000F and HP-A900) perform floating-point calculations at a rate comparable with integer operations. Other microprocessors specifically designed to handle floating-point operations (e.g., Intel 8x87) are much slower than their integer counterpart. This mismatch in the Intel floating-point (or co-) processors has become even more pronounced as the integer (main) processors have improved. The Intel processors have improved in architecture and speed at a much faster pace than the floating-point coprocessors (Fried, 1985). A 20MHz 80386 is approximately 20 times as fast as a 5MHz 8086; whereas a 20MHz 80387 is only 4 times as fast as a 5MHz 8087. The disparity in the early processors (8086/8087) was already severe and has subsequently worsened by a factor of 5. This disparity should not be ignored in the development stage if these machines are the destination of the software being developed.

Some floating-point calculations should be avoided if at all possible. For instance, most engineers who have long used hand-held scientific calculators are surprised to find out that Intel coprocessors do not have an instruction to perform exponentiation (e^X, 10^X, or X^Y). In fact, such operations require no less than 40 instructions in assembler and are encoded as function calls by high-level language compilers with that additional overhead (Microsoft, 1987). Exponentiation operations are common in steam property calculations.

Other floating-point calculations can be converted to integer operations with implied divisions. On machines with a large disparity in the relative speed of integer and floating-point operations this can have a substantial effect on speed. Many helpful details on the operation of Intel floating-point processors and assembly language code are given by Duncan (1990).

Double precision (64-bit) floating-point operations are unnecessary in most engineering applications except matrix inversion and are totally unnecessary in steam cycle analysis. Single precision (32-bit) floating-point operations require only half the memory and may be as much as four times as fast depending on the hardware. Some compilers actually default to double precision, but can be instructed not to; while some others do not even support single precision. This is an important consideration when selecting a compiler and even a language in which to develop a program.

Hybrid 8/16-bit microprocessors have a distinct disadvantage compared to true 32-bit machines when accessing large arrays (above 64K-bytes). When developing a program to run on one of these machines it is crucial to avoid large arrays as much as possible and to access them in blocks if this cannot be avoided (most processors have special instructions for moving blocks of data). The selection of a compiler can also make a significant difference in this situation. Depending on the compiler, accessing elements of an array by equivalence can also be much faster than by index. Detailed discussions of the differences between various compilers of the same high-level language and their relative strengths and weaknesses are given by Wolf (1985) and Shaw (1988) for FORTRAN and C respectively.

Selecting Optimum Algorithms

Even complicated more efficient algorithms are preferable to brute force. Microprocessors can generally handle more code and more sophisticated algorithms much better than simplistic code and many iterations. Multi-step methods which utilize information from previous iterations are generally preferable to single-step methods which only utilize information from the current iteration and rely heavily on relaxation for stability.

It regrettably needs to be mentioned, that textbook algorithms are frequently not the most efficient ones available. There is a considerable lag time between the writing of even the best textbook and its selection from the shelf of a working engineer. Most textbooks are intended to convey principles on which the student can build--not the latest developments. This is especially true in textbooks from one field (e.g., thermodynamics or heat transfer) which draw on methods from prior existing textbooks from some other field (e.g., numerical mathematics); and engineers are not alone in their tendency to seek out information from within the field with which they are most comfortable.

Kernalization

A final and extremely profitable area of software improvement is kernalization, or identifying computationally intensive specific tasks within a program and developing specialized code to perform that task in an optimal manner (Hewlett-Packard, 1980). The use of assembler to encode these kernels is especially advantageous. Many such kernels are available which were written in assembler for Intel processors but can be called from a high-level language such as FORTRAN (Benton, 1987). For example, one such kernel will evaluate a regular polynomial 5 times as fast as optimized FORTRAN or C. Another will take the absolute value of an array of floating-point numbers 17 times as fast as optimized FORTRAN or C.

Expected Return on Development Effort

Admittedly, all of these suggestions increase the effort required in the development stage of a code. It is implicitly assumed in this paper that a greater investment in development--which is done once--will yield even greater dividends in application--which is done many times.

SPECIFICS OF THE STEAM CYCLE MODEL

The general features of the steam cycle model are presented as examples of how these concepts might be implemented and as suggestions of what might be done in this and other areas. The main emphasis of this paper is not to present the numerical results of this compared to other similar programs, although such information is available (Shelton, 1988). The objective here is to stimulate development and to offer innovative, proven techniques to other software developers. To this end, over a hundred parts and related programs have already been made available through the ASME/CIME computer, including various FORTRAN and assembler source codes, executables, examples, and documentation (Benton, 1987-90).

User Interface Shell

In addition to the two-step analysis procedure already discussed, the steam cycle model is managed by a customizable user interface shell program. This shell was written in assembler and is not limited to steam cycle analysis. The shell performs such tasks as managing different applications (i.e., keeping track of the data and procedure files associated with each steam system). The shell also makes sure that the user performs certain steps in the proper order (e.g., the user may change the configuration of a system many times or in several steps, but it must be re-optimized before it can be solved). The shell also routes the output from the steam cycle model to the CRT, printer, or a file. The shell allocates system available memory in order to run other tasks such as the two-step steam cycle model and auxiliary programs (e.g., manufacturer-specific turbine characteristics, heat balance plotting program, thermodynamic property tables, cooling tower analyzer, and a quick simplified steam cycle model).

Optimizing Preprocessor

The first part of the two-step steam cycle model--or the optimizing preprocessor--performs several tasks. It receives information in symbolic form (i.e., convenient for the user) and performs a thorough check for consistency. During the consistency check the program will automatically add as many mixing and/or splitting junctions as required to connect the components as implied by the user. It then automatically numbers and names the minor components (e.g., pipes, nodes, junctions, etc.). This service saves considerable effort over other programs which require the user to number and keep track of every single junction and pipe or flow stream and carry these through any configuration changes. Once the preprocessor has numbered and assessed the configuration of the system, it determines the optimum (space-saving) manner in which to store this information in a database which is passed to the postprocessor.

The preprocessor then analyzes the connections or paths of influence between the components as to their degree of dependence and implicitness. The optimum (timesaving) order in which each parameter (i.e., flow, energy, temperature, pressure, etc.) is to be computed by the postprocessor is determined based on its degree of dependence and implicitness. For instance, an independent parameter should be computed before a dependent as should an explicit one before an implicit. Some components are found by the program to be solvable from the inlet(s) out; while others may be solved from the

exit(s) in. This bidirectional explicitness is part of the optimization process.

As this optimize-by-ordering process proceeds, all parameters which were indeterminate will become either explicitly or implicitly determinant. When no more explicit relationships between parameters can be found (i.e., all remaining relationships are implicit), an estimation is made as to which implicit parameter can be assumed (and later iteratively corrected) which is most likely to have the least influence on the rest of the system (i.e., the least critical guess).

At some point in this process, unresolvable indeterminacies may be identified. If this occurs, the user is called upon to select the manner in which to resolve these. Thus the same system may be solved differently depending on how the user responds to interrogation. One example of this is implied split flows. Parallel paths typically lead to ambiguities. These may be resolved at the inlet, exit, intermediate point, or some combination of these depending on the specific connections. When the entire solution procedure has been determined by this optimization process, the procedure for solving the system (including the procedure to correct any assumptions, test for convergence, and process nonfatal errors) is written to a file in cryptic form (convenient for the machine rather than the user). The preprocessor also inserts any custom data (which can override various defaults), custom procedure (which can consist of FORTRAN-like code and user-defined submodels), or custom summary (user-defined output instructions) at what it determines to be the appropriate location within the data and procedure which the preprocessor itself has generated. Any encryptions which can be made in the custom data, procedure, or summary are also made at this time in order to speedup execution.

The result of the preprocessor is a file which will be passed to the postprocessor containing all of the data and instructions it will need to solve the system in an optimal manner. All of these functions are performed by the preprocessor using only character (8-bit) and integer (16-bit) operations.

Interactive System Constraints

After the preprocessor and before the post processor is run, it is necessary for the user to prescribe the system constraints. These constraints include such parameters as main steam flow or net output, boiler exit temperature or heat input, condenser cleanliness or backpressure, etc. The user can also "tag" any parameter in the system and thereby cause it to be treated as a system constraint. Tagged parameters can be set to a constant, scaled by some proportion to another system constraint, specified by a user-defined function (i.e., FORTRAN-like code), or even implicitly determined by a submodel (i.e., many lines of user-defined code). Prescribing these system constraints is accomplished by the shell or manager program invoking an interactive menu utility which interrogates the user for this information or alternatively uses default values supplied by the preprocessor.

Interpretive Postprocessor

The second part of the two-step steam cycle model--or the interpretive postprocessor--also performs several tasks. It first reads the system configuration information in cryptic form (i.e., convenient for the machine rather than the user), including how this is to be stored in a temporary database. This database is necessary due to the limited memory available with PCs and the large amount of data necessary to describe complex steam systems having hundreds or even thousands of components. The postprocessor accesses this information in the database in blocks which are swapped according to "age" and frequency of use. Thus, the database manager (which is a part of the postprocessor) "learns" as the system is being solved. Only a minimal consistency check is performed on the system by the postprocessor as it reads the system configuration and creates the database.

After this point, the postprocessor basically executes the procedure provided to it by the preprocessor. Either the combination of these instructions and the system constraints will result in the solution of the system and listing of the results, or a nonfatal error (the procedure for dealing with each having also been supplied by the preprocessor) may be encountered, or the intervention of the user (by entering an unsolicited keystroke) may be detected. If the procedure is carried out to completion, the program stops. If a nonfatal error is encountered, the user may be invited to intervene, correct the problem, and order the program to resume or abort.

If the user intervenes by choice or invitation, the program can process any of its several hundred commands interactively as if it were carrying out the instructions of the preprocessor. These commands include solution of components (i.e., turbine, pump, heat exchanger, etc.) and interactive execution of FORTRAN-like code (equations, print, read, rewind, goto, if(...)then, simultaneous equations, thermodynamic properties, etc.). Also provided in this interactive mode are on-line help and a built-in editor which can recall a predetermined, adjustable number of previous commands.

The command interpreter (which operates in either batch mode--reading commands from a file--or interactive mode--reading commands from the keyboard), can efficiently process a wide range of commands. These are interpreted in order of convenience for the machine so as to improve speed. Cryptic (i.e., machine-oriented) form is checked before symbolic (i.e., user-oriented) form, as are abbreviations before entire expressions. Most frequently used commands are checked before less; thus the command interpreter "learns" the style of the commander as it processes commands. Further improvements in efficiency are realized as most frequently used expressions and macros[4] are kept in a convenient location within the database.

Comparison and Contrast to Other Similar Codes

Several other large-scale, off-line, steady-state steam cycle codes are commercially available and widely used. These include PEPSE (Energy Inc.), SYNTHA (Syntha Corp.), THERM (EDS Assoc.), and THERMAC (Expert-EASE Systems, Inc.). These codes in some form are available for use on mainframe, mini-, and microcomputers. SYNTHA has been available on Control Data mainframes for a number of years. PEPSE, THERM, and THERMAC have logged significant use on microcomputers (see for instance, Dixon et al., 1984, Larson et al., 1988, Kettenacker, 1988, and Jain et al. 1989.) This is by no means an exhaustive list of steam cycle codes. Neither is there any intention to deprecate these or any other codes.

The present code, called SCRAP (Steam Cycle Rankine Analysis Package), was developed from start to finish on and for microcomputers. It has never been run on a mainframe; nor would it be suitable for use on a mainframe. The strengths, weaknesses, and idiosyncrasies of microcomputers were carefully considered at every step of its development. Substantial portions of the code (including the database manager, file and CRT I/O, editor, command interpreter, and the steam property generating functions) were written entirely in assembler. None of the previously mentioned codes utilize a dynamic database in which to store the information describing large systems (except that which might be part of a virtual operating system external to the code). None of the previously mentioned codes utilize optimized pre- and interpretive postprocessing. None of the previously mentioned codes can interactively process code or support application-specific customized procedure without recompiling the source code.

[4] Macros are single symbols that refer to an entire cluster of individual commands which may in turn include other macros. The preprocessor defines some macros for its own use. The user can define other macros or redefine the ones defined by the preprocessor; thus changing the way in which the postprocessor handles certain equipment and exceptions.

The present steam cycle code is sufficiently different in its developmental concept from the previously mentioned codes as to represent a distinct technology. SCRAP is fundamentally different. Perhaps the most striking difference is the command interpreter which can batch or interactively process symbolic algebraic expressions, file manipulation, and I/O. The command interpreter can even process expressions having nothing to do with steam cycles or engineering. This same command interpreter kernel could be used as the core for other engineering applications such as chemical process plant modeling.

Even if there were no need to handle very large systems by using the dynamic database, and even if the speed of solving the system were of no concern, the versatility of being able to prescribe custom procedure for each unique application would make the concept used in the present model an attractive alternative to other codes. Although there is some sense in which similar steam systems are "generic" in design, there seem to be unique arrangements or components in most real systems. The past approach to handling these uniquenesses has been to either approximate using a near equivalent or provide numerous specific components (modifying the source code to handle each new one). The present approach is to have the preprocessor provide the procedure to handle the common components and then allow the engineer to provide any unique code (all with no modification to the source code) to handle uncommon components or arrangements.

SUMMARY

Computers are simply not generic number crunchers. At least the basic class of hardware (i.e., mainframe, mini, micro, 32-bit/16-bit, etc.) must be considered before developing software if it is to be efficient or even practical. Significant advantage can be realized through more efficient use of a computer's resources. The efficient use of a computer's resources depends on the approach taken, algorithms selected, and the overall strategy. Additional effort in the development stage can not only result in greater efficiency, but also increased flexibility and even extended capabilities. The concept of subdividing a larger task into optimized preprocessing and interpretive postprocessing has been presented as an innovative way to utilize the strengths of microcomputers and avoid their weaknesses. A steam cycle analysis program has been used to illustrate the merit of this approach.

REFERENCES

Benton, D. J., 1987, "FLIB: Fortran Callable Library", ASME/CIME Bulletinboard, (608)233-3378.

Benton, D. J., 1988, "PROPS: Thermodynamic and Transport Properties of Steam and Moist Air with Selected Power Plant Equipment," loc. cit.

Benton, D. J., 1989, "QUEST: Quick Estimate Steam Turbine Performance Code", loc. cit.

Benton, D. J., 1990, "MUPIT: Multi-Unit Power Plant Cooling System Model", loc. cit.

Dixon, R. R., N. B. Kraje, and R. C. Roberts, 1984, "Current Fossil Fuel Power Plant Performance Monitoring Volume 1: Practices," Report No. EL-3339V1, Electric Power Research Institute, Palo Alto, CA.

Duncan, R., 1990, "Power Programming: Arithmetic Routines for Your Computer Programs, Part 6,"

PC Magazine, Feb. 13, pp. 297-307. (Also see parts 1-5 in earlier editions.)

Fried, S. S., 1985, "The 8087/80287 Performance Curve," *BYTE*, Fall, pp. 67-88.

Hewlett-Packard, 1980, "Vector Instruction Set (VIS) User's Manual" Part No. 12824-90001, Hewlett-Packard Company, Cupertino, CA.

Jain, P., A. Padgaonkar, T. Kessler, D. Gloski, and G. Kozlik, 1989 "Plant Thermal Analysis and Data Trending Using THERMAC," *Proceedings, 1989 EPRI Heat-Rate Improvement Conference*, Report No. 1711, Electric Power Research Institute, Palo Alto, CA.

Kettenacker, W. C., 1988, "Use of An Energy Balance Computer Program in the Plant Life Cycle," *Proceedings, 1988 EPRI Heat-Rate Improvement Conference*, Report No. GS-6635, Electric Power Research Institute, Palo Alto, CA.

Larson, J. and B. Kraje, 1988, "PC-Based Power Plant Performance Analysis: Site Delivery of Powerful Analysis Techniques Now Possible Using PC Technology," *Proceedings, 1988 EPRI Heat-Rate Improvement Conference*, Report No. GS-6635, Electric Power Research Institute, Palo Alto, CA.

Microsoft, 1987, *Macro Assembler 5.1 Reference*, Microsoft Corp., Redmond, WA.

Muchnik, S. S., 1988, "Optimizing Compilers for SPARC," *Proceedings, COMPCON'88*, San Francisco, CA. (Also reprinted in _Sun Technology_, Summer 1988, pp. 63-77.)

Shaw, R. H., 1988, "Compiling the Facts on C," *PC Magazine*, Sept. 13, pp. 115-183.

Shelton, R. J., 1988, "An Evaluation of TVA's Steam Cycle Rankine Analysis Package (SCRAP)," Southern Company Services, Birmingham, AL.

Wolf, C., 1985, "Serious FORTRAN for the PC," *PC Magazine*, Dec. 24, pp. 161-171.