COMPUTER-AIDED CHEMICAL ENGINEERING

1. Introduction

In the 1950s, the chemical industry was the first civil industry to make extensive use of computers. At that time, computers were relatively more expensive (a mainframe might cost US$1 million). However, they were inexpensive compared to the cost of a chemical plant that would typically exceed US$100 million. The cost of a mainframe could be repaid in its first application to process design. Distillation columns were being designed by computer in the 1950s and process simulation programs were available by the early 1960s. The first commercial simulation package was probably PACER, which was well established by the mid-1960s. The first on-line computer controlled processes were commissioned in the late 1950s. Thus, Computer-Aided Chemical Engineering has nearly 50 years of history.

Computers are now cheap and there is one on the desk of every engineer. Computer aids are used at every stage from conception through design to operation. Virtually all chemical engineering is now "computer-aided chemical engineering".

At the conceptual stage, software is used to plan and analyze laboratory experiments. Computer programs are used to estimate chemical and physical properties for chemical species for which experimental data are lacking. Software tools, such as computer-aided molecule design (CAMD), are employed to devise chemicals with desired properties. CAMD is increasingly relevant as the chemical industry moves to the position that its primary goal is to sell effects rather than chemicals. Thus, it sells detergents, solvents, fuels and fibers and only incidentally sells specific chemicals that have these properties. Process synthesis is used for the conceptual design of processes that will manufacture the desired chemical species and mixtures.

At the design stage, processes (developed manually or by computer synthesis) are simulated in detail to ensure safe and economic operation. The designs may also be optimized to minimize costs, maximize profits or meet environmental or safety criteria. Computer programs are used for both the process and mechanical design of individual unit operations. The safety and environmental impact of the proposed processes is assessed using appropriate software.

A wide range of computer tools may be applied to operating plant. Computers may be applied on-line or off-line. On-line applications include regulation control and optimizing control of continuous processes. On-line computer-control is also applied for scheduling batch and semibatch processes. From the earliest applications, on-line computers have been used for safety monitoring and rapid automatic shutdown. Most current on-line software includes such emergency response facilities.

Off-line optimization of operating variables can give substantial benefits. Processes rarely operate exactly as designed, even when the design is optimized. Uncertainties allowed for in design are largely resolved in operation. Reduced uncertainty enables more accurate simulation than was possible at the design stage. Greater accuracy enables optimization of operating policy to exploit favorable outcomes to the uncertainties and to mitigate the consequences of unfavorable outcomes. Typically, 20–50 variables are available for optimization. Beyond 2 or 3 optimization variables, it is impossible to determine optimal conditions manually, and the optimum may even be counterintuitive. There are examples where off-line optimization has doubled or tripled the operating margins for processes. The impact on plant profitability can thus be high. Off-line studies are also undertaken to assess potential process improvement, eg, through modification of operating schedules or through retrofit design.

In practice, an engineer may have up to 200 different computer programs that can be applied to aid the efficient design and operation of chemical processes. These programs must give correct answers and the answers must be correctly interpreted. A typical range of computer software employed by a large manufacturing company is given in Table 1. The AIChE on-line directory (1) lists a wide range of available commercial software.

The objective of this chapter is to assist both chemical engineers developing software and chemical engineers using software written by others.

The section Developing Engineering Software covers program development. It is aimed primarily at engineers writing relatively small programs, or contributing specialist modules for larger programs. This section provides guidance on commonly occurring problems, and includes material important to engineering software that is not usually covered in software engineering or numerical analysis texts. At the same time, introductory material and references are provided for specialist chemical engineering software engineers.

The section Using Engineering Software is designed for engineers using third-party software. It describes steps necessary to ensure that software is used effectively and it refers to more comprehensive works on the topic.

The section Current Advances briefly describes some areas of active research in computer-aided process engineering (CAPE).

This chapter does not describe particular chemical engineering software in detail. The breadth of material (as indicated in Table 1) makes such a description beyond the scope of a single chapter. Neither does the chapter give a comprehensive exposition of software validation or numerical analysis. There is extensive literature available for specialists in this area and some of this material is referred to. Nevertheless, the material presented here is intended to be sufficient for chemical engineers involved in computer-aided chemical engineering as an incidental part of their work. In-depth discussion of on-line computer control is

Table 1. **Examples of Computer-Aided Engineering Software Summary Table**

Data Correlation and Prediction of Physical and Chemical Properties
Fitting experimental data for physical properties. Predicting physical properties
Fitting experimental reaction equilibrium and rate data
Prediction of flammability, toxicity, and other data important for safety computation
Prediction of ozone depletion, greenhouse effect, and other data for environmental studies
Prediction of equipment failure and repair rates
Unit Operations
Heat exchanger process and mechanical design (includes multi pass, multi fluid, tube, plate). Fired heaters
Evaporator design
Design of pressure vessels according to various national and international standards
Simulation and design of distillation columns, batch, and continuous distillation
Absorption and stripping column simulation and design
Design of column internals, packed columns, valve tray, sieve tray, and bubble-cap
Reboiler and condenser design
Compressor and expander design and simulation
Liquid–liquid extractor simulation and design
Modeling Rankine cycle systems
Gas and steam turbine modeling
Analysis and performance of agitated vessels
Pressure drop calculations for liquids and for gases in isothermal and adiabatic flow
Two-phase pressure drop calculations in pipes and conduits
Two-phase choked flow
Instability in two-phase flow
Surge analysis
Estimation of tube vibration
Tubular and fluidised bed reactor simulation and design
Fluidized bed drier modeling
Driers, indirectly heated, directly heated. Spray driers
Restrictor orifice design
Combustion calculations
Non-Newtonian flow and heat transfer calculations
Furnace design and radiant heat transfer
3-D fluid flow, heat, and mass transfer through Computational Fluid Dynamics
Environmental Calculations
Dispersion of gases, aerosols and smokes from stacks, ruptures, and fires
Dry and rain-enhanced deposition of aerosols and smokes from plumes
Leaching from landfill sites and dispersion of leakages through groundwater
Modeling river networks for accumulation of pollutants
Concentration of pollutants in land and marine life (vegetable, animal and microbial)
Integrated effect of releases to air, water, and land
Safety Studies
Hazard analysis, fire and explosion, toxic chemical release
Bursting disk and pressure relief valve computations
Adiabatic and isothermal relief in piping networks
Design and simulation of flare release systems
Process Availability and Reliability
Plant availability computed from equipment failure and maintenance statistics
Availability with stand-by equipment
Process Simulation
Simulation and design of steady-state processes
Data reconciliation (estimation of statistically most likely performance from measurements)
Simulation and design of unsteady processes: Control, start-up, shut-down, upset conditions
Design of batch and semi batch processes; dedicated, multi purpose and multi-product plant

Table 1 (*Continued*)

Discrete dynamic simulation of batch and semi-batch processes
Simulation of linked distillation columns
Simulation of heat-exchanger networks
Simulation of evaporator trains
Site simulation: energy use, utilities requirements, major intermediates (e.g., HCl)
Economic evaluation of plants and sites
Optimization of design and operating conditions
On-line Computation
On-line optimization to compensate for performance, market, and raw material changes
On-line data reconciliation
On-line regulation control to ensure stable performance in the face of disturbances
On-line fault diagnosis
Condition monitoring (prediction of equipment deterioration from noise etc. measurements)
Optimization of start-up, shut-down and load change trajectories
On-line scheduling, e.g., of batch-process operation
Aids to the Design Process
Intelligent piping and instrumentation diagram systems
Design rationale systems (knowledge-based design)
Interchange software based on Process industries STEP standards (ISO 10303-221)
Process Synthesis
Minimal energy or utility requirements for a process
Minimal energy or utility requirements for a site
Optimal heat-exchanger network design
Optimal separator network design
Optimal process design by process synthesis

deliberately avoided (although an important part of the subject) because it is covered in a separate chapter in this encyclopedia on Process Control . Many of the principles employed to produce quality software and quality-assured results from software are common to other areas of Quality Assurance, for which the reader is referred to the chapter in this encyclopedia on Quality Control.

2. Developing Engineering Software

This section presents general principles in developing engineering software, which should:

1. Meet defined engineering goals.
2. Be based on sound principles of chemistry and physics.
3. Be testable and maintainable.
4. Be tested to ensure that it correctly codes the models on which it is based, for example that it is dimensionally consistent.
5. Take account of the finite precision of computer arithmetic to give numerically accurate results.

This section covers program design, preparation and testing, and covers aspects of numerical analysis relevant in developing engineering software. The

general principles are illustrated with examples from well-known chemical engineering topics.

2.1. Program Design. We present guidelines broadly based on the European Space Agency (ESA) software engineering standards (2). These guidelines are easily adaptable to specific engineering applications. Professional software companies may apply more detailed and formal tools. For example, the Unified Modeling Language (3) presents a formal object-oriented approach to software design, backed by commercial software tools. A number of established texts [eg, NIST (4) and W. Perry (5)] give guidance for professional software engineers. Tanzio (6) puts these validation methods in a chemical engineering context. The general approach given here is consistent with such more detailed tools. Figure 1 presents an outline of the software design process.

The initial stage is to specify exactly what the user wants, the User Requirements. Where the software automates a procedure well known to the intended end-users, these requirements should be drawn up in consultation with the end-users. Where the software can give functionality beyond the experience of current users, the input of innovators in the field is required.

The rationale (reasoning) behind each requirement should be recorded to ensure that the program is written efficiently. For example, there may be a requirement that a simulation should be capable of dealing with two-phase gas/liquid flow. If the intended use is to simulate the boiling and evaporation of water to feed steam turbines, the whole range from 100% liquid to 100% vapor must be covered. Vapor/liquid equilibrium must also be covered and an approach using steam thermodynamics might be appropriate. On the other hand, the intended use may be the simulation of flow in gas pipelines in which small levels of liquid contamination might occur. This requirement is easier to meet with simpler, faster, and more easily tested software. The provision of the rationale for each requirement enables the programmer to provide the most appropriate tools and gives scope to meet the requirement in alternative ways.

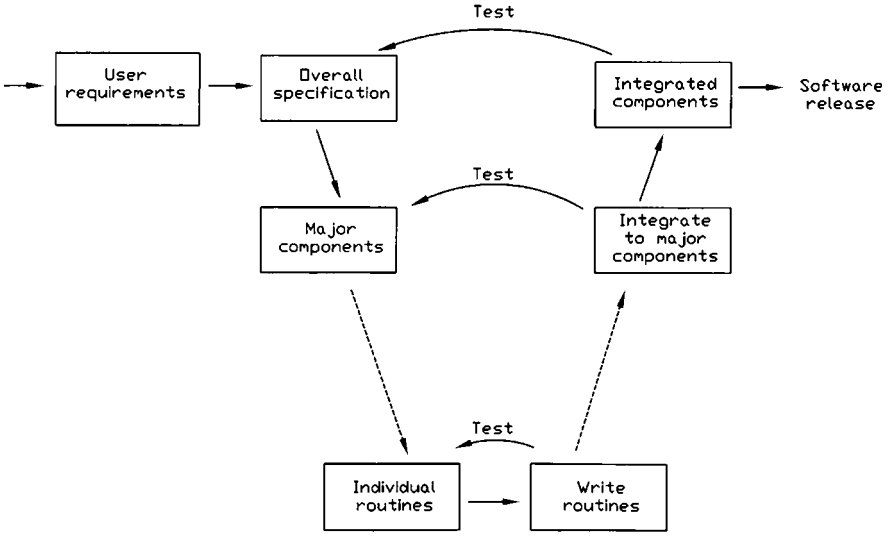

Fig. 1. Software design process.

The User Requirements cover:

The functionality, that is the chemical engineering problems to be solved.

The user interface, ie, how the user will interact with the program (the appearance of any windows, the use of buttons and keys etc).

The environment in which the program will be used. Will it be stand-alone, accessed over the web or used in some other way, eg, as an additional function to be attached to a spreadsheet?

The programs to which it must be interfaced. Does it import data from another program and/or export results to a further program?

The services that it must use. For example, must it employ a defined physical properties package?

The User Requirements prioritize the required functionality. *Essential* requirements must be met in the first release; without this functionality, the software does not meet its basic goals. *Desirable* requirements would add value to the program, but could be deferred or omitted. Desirable features may be further prioretized, ranging from those that a majority of users would wish to see in an early release, to those that would add only marginal value. If they are easily met, desirable features may appear in the first release. A User Requirements document should include features that were considered and rejected for good reason.

The User Requirements should translate directly into tests that will verify that (at least for the specific values tested) the software meets the requirements. These tests form part of a validation plan for the software.

The overall software package is divided into its major components. These software components may be subroutines, classes (objects), components dynamically linked by middleware, or separate programs that will be invoked as needed. The interfaces between these components must be defined so that they can work together. At this stage, separate acceptance tests for each major software component are defined.

The major software components are further subdivided and acceptance tests for each subcomponent defined. The extent of subdivision depends on the size and complexity of the program.

When the program is written, each component is verified against the pre-defined tests before it is integrated into the final program. Similarly, subcomponents are tested both before and after integration. The test procedure is an integral part of the program design.

Before programming starts, a "build" sequence is defined. The build methodology provides early delivery of a simple program offering a subset of the requirements. The program is enhanced in each subsequent build as more functionality is added. For example, a simulation may initially work only for single-phase mixtures with simple ideal thermodynamics and a small subset of chemical components. Each build is tested and evaluated as it is developed. There will be a number of builds before the first version meeting the essential user requirements is released.

The build approach has benefits both in meeting delivery dates and in improved program quality. Delivery dates are improved because any delay in the first build signals problems that can be identified and corrected early. A sche-

dule without early delivery may enable such software problems to remain unrecognized for a long period. Quality is improved because experience with early builds enables the User Requirements to be refined. Iterative refinement gives a better final product. Actual use of a program (even a version with limited functionality) is more effective in highlighting opportunities and difficulties than any paper study. These early releases also give an opportunity to involve the intended final users where, previously, the specification was largely drawn up by specialist innovators.

The build discipline extends beyond the initial release into the support and development phase. User experience provides suggestions for enhancements and uncovers bugs that escaped pre-release testing. In response to this experience, the User Requirements document is updated, the changes prioritized, and new releases planned each with its own test and build schedule. The prioritization of changes depends on the intended end-use of the program. For example, a bug that causes a program crash requires urgent correction in an on-line control application. However, an engineering design program bug that gives answers leading to unsafe designs requires more urgent correction than a bug that causes occasional program crashes.

Each build and release should be archived so that they remain accessible over extended periods. (For example, in a plant upgrade, users might wish to compare results with results they obtained 5 years earlier. Additionally, enhancements may introduce new bugs, and it may be desirable to go back to the last bug-free version.) Versions of major components should similarly be separately archived. Dynamically linked components may follow separate, unsynchronized, build and release patterns.

An effective software design and validation procedure thus lasts throughout the life of the software.

2.2. Programming Languages and Modeling Systems. Chemical engineering software may be written in a conventional high-level language, an artificial intelligence (AI) language, a general-purpose or specific equation-based modeling system, or using spreadsheet tools. Many engineers construct simulations using modeling tools rather than write special-purpose programs. Simulations developed using these tools must be designed and tested as for conventional programs. There is the same scope for logical and numerical errors.

A separate list of references is given for programming languages and modeling systems. The general-purpose languages described below are not referenced because they are so well known that a wide choice of texts is available from most libraries and other book suppliers.

General-Purpose Languages. Popular high-level languages include C/C++, Java, Fortran90, Delphi, Smalltalk, Eiffel, Ada and BASIC. Most of these languages are now object-oriented. Such object-oriented languages group methods and data in a way that is natural to the engineer. Indeed, the original object-oriented language, Simula, see Birtwhistle and co-workers (7), was developed specifically for simulation. It is no coincidence that Birtwhistle came from a background of simulation in the chemical industry.

The most used languages for large-scale engineering software are Fortran and C++, with most new software written in C++. The evolution of C++ from C results in alternatives for many common features. For example, there are four subtly different array types, a “vector”, a “valarray”, a fixed-bound directly

declared array, and a pointer to dynamically allocated memory. The programmer can also create array-like container classes. Confusion between alternatives gives scope for subtle bugs. Later facilities frequently provide safer alternatives to earlier relatively risky methods. In recognition of the risks, Stroustrup (8) repeatedly emphasizes safe programming methods. A simple subset of C++ is recommended emphasizing safety rather than speed. For example, use only "vector" for arrays.

Languages Designed for Artificial Intelligence. AI languages include LISP and Prolog, and can be used for rule-based programming. They may be convenient for implementing standards that are presented as rules. For more general programs, the rules may be heuristic (based on experience) or based on fundamental science. Where they are based on experience, applications should preferably be restricted to the systems for which the experience was gained. Interrogation facilities should be provided to display the rules employed and the conclusions may need to be tested by conventional modeling tools.

Equation-Based Modeling Systems. Equation-based systems range from equation manipulation systems such as Mathematica and MathCad, through general tools for solving and optimizing problems defined as equations (eg, GAMS) to tools specifically for the chemical engineer, such SpeedUp, and gProms.

Equation manipulation systems are rarely used for large engineering calculations. The difficulty is that there is no control over the numerical characteristics of the resulting equations. These systems are more generally used in developing algorithms. The manipulated equations are examined before the results are incorporated into an assignment-type program or an equation-oriented modeling system.

The remaining equation-based systems do not rearrange individual equations. For example, they cannot convert $a = \ln(b)$ to $b = \exp(a)$. Large sparse sets of algebraic equations are solved by forming a local linearization and solving the local linearization. The equations are re-linearized at the resulting solution point and the iteration continued until a solution is obtained. Differential equations are solved numerically using robust integration methods.

Spreadsheet Tools. Spreadsheets, such as Excel and Lotus 1-2-3, offer many attractions to the engineer. A calculation can be put together quickly and a variety of graphical output is immediately available without special programming. Most spreadsheets also have standard database interfaces for input and output of extensive data.

Spreadsheets have the disadvantage that the formulas tend to be hidden and conditional expressions, required to maintain numerical accuracy, are difficult to program. The selection of cells as matrix sizes are changed also depends on the way that the programmer handles cell references and errors can result. Most importantly, a user may inadvertently overwrite a formula cell with a value. In all these cases, there is the danger that plausible but wrong answers can be obtained. Consequently, it is difficult to apply quality assurance measures to spreadsheet calculations.

It is recommended that spreadsheet computations are restricted to simple accounting-type operations that are easily checked by hand. Relatively complex computations should be undertaken with a conventional programming language.

If required, a conventional program can be linked to a spreadsheet to get the benefits of spreadsheet input–output and of the postprocessing tools that become immediately available.

2.3. Programming. Programs should be written to minimize programming errors. This section concentrates specifically on techniques for minimizing incorrect results and run-time failures. The section on numerical analysis deals with numerical errors in otherwise correct programs. The topics covered in this section are

Dimensional consistency of programs.
Limiting side-effects.
Limiting run-time.
Limiting use of computer memory.
Arithmetic failures.
Division into component parts.
Checking data.
Handling uncertainty.
Minimizing error messages.

Dimensional Consistency. Every assignment and comparison in a chemical engineering program should be dimensionally consistent. Thus, any program that assigns a velocity to a mass is certainly wrong. Errors such as writing

$$E = 0.5 * m * v$$

when the intended assignment is

$$E = 0.5 * m * v * v$$

cannot be detected by any of the standard high level compilers. Such mistakes do not cause the program to crash and rarely give obvious run-time errors. In the above example, the error is not obvious unless v differs considerably from unity. Such errors can be detected by checking that every term on the right-hand side of an assignment or comparison has the same dimensions and that the dimensions are the same as on the left-hand side. Tools are being developed for automating such tests before compilation, but generally the tests must be done manually.

Alternatively, dimensional consistency can be checked dynamically (at run time). Any computer language permitting overloading (including all object-oriented languages) permits the definition of new data types. A "dimensioned" data type, which consists of the floating point value plus the dimensions, is needed. For example, with dimensions in the sequence mass (M), length (L), time (T), temperature (Θ), a force of 27.9 may be recorded as $\{27.9, 1, 1, -2, 0\}$. The significance is 27.9 units, with dimension MLT^{-2}.

In this system, the standard mathematical operations are modified so that multiplication multiplies the values and adds the dimensions. Addition, comparison, and assignment check that the dimensions are consistent and generate a run-time error for any inconsistency. Other operations are similarly modified.

The technique is slow, but gives a thorough check. The final release version may have the checking facility removed.

As a further assurance, engineering programs should be written in dimensionally consistent units. Thus, do not mix viscosity in poise (the cgs unit) with pascal-seconds ($kg \cdot m^{-1} \cdot s^{-1}$, the SI unit). A dimensionally consistent program will work equally well in the cgs, SI(MKS), or fps systems. It avoids dimensional constants embedded in the program. Unit conversion should be applied at input and output, not within the program itself.

Side Effects. Side effects occur as follows. A statement is written such as

$$y = \text{somefunction}(x)$$

This operation alters "y". There is a suspicion that "x" may also be altered. No other variables should be altered. If, on calling "somefunction", an unrelated variable z is altered, the change is known as a side effect. Side effects give error-prone programs and make debugging and upgrading difficult. Where side effects are excluded, a problem in the above statement can be localized to "somefunction". That is then the only function that needs checking. If side effects are allowed, every function that might contain "z" has to be checked as does every line of code in which z appears. If a function containing z also contains side effects, the number of lines of code that need checking are further multiplied. It is thus strongly recommended that side effects be avoided.

Some languages strictly forbid side effects, to the extent of not allowing changes in x. However, all popular languages allow side effects. Fortran COMMON allows every program module to access variables listed after a COMMON label. It is not necessary to put these variables in argument lists to access them. Assigning values to COMMON variables thus creates a side effect. COMMON was recommended in the early days of Fortran because it gave better run-time efficiency than passing parameters as subroutine or function arguments. Fortran90 discourages the use of COMMON and provides safer alternatives. C++ provides both pointers and global variables. Pointers enable several variables to refer to the same memory location. Thus allocating a value to one of the variables alters all the others. Reference variables can provide similar confusion. If not declared "constant", global variables can be accessed from anywhere. Thus, any routine that is called can alter one or more global variables (C++ global variables are similar to Fortran COMMON in this respect). These altered global variables can have unsuspected effects elsewhere in the program. Pointers were recommended in the early days of C to avoid copying whole structures. Runtime efficiency was thereby improved. In engineering programs, the time saving is likely to be minimal; there are many more operations performed on the elements of a structure than merely copying them. The major current use for C++ pointers is accessing components dynamically linked by middleware.

Polymorphism provides a concise way of performing related tasks. (Polymorphism provides a common interface capable of invoking a variety of related behaviors that do not need to be defined in advance.) However, its use employs side effects, and it is deliberately excluded from some languages (Fortran90). It should be used only when the alternative would be a more complex program.

All side effects should be used sparingly and commented whenever they are used.

Limiting Run Time. There should be a known upper limit on run time for all engineering software. On-line programs should be strictly timed in advance so that there is always slack time and interrupts cannot accumulate without limit. Possible long run times for off-line programs should be noted in the documentation and a warning message displayed before the time-consuming part of the calculation commences.

Run-time should be estimated from the data values read. Where theory is deficient, the engineering programmer should produce an experimental correlation of run time versus problem size. It is then possible to set a fixed upper limit on number of iterations that will give an acceptable run time on most computers. Where, because of the limit, the program runs out of iterations, the programmer can provide a meaningful message related to the engineering problem being solved. In most cases, an estimate of the result can also be given. Unless the program has its own stop button, a program stopped by the user can provide no meaningful message.

Limiting Use of Computer Memory. As for run time, it is should be possible to make a conservative estimate of how much memory will be required as a function of data values. An upper limit for space required should be set in advance so that memory overflow is avoided. Where additional space is taken at each iteration (eg, in branch and bound optimization), an iteration count can limit memory used. A run-time error message from the computer operating system when computer memory is exhausted is of no value to the user. However, the programmer can provide a meaningful message related to the engineering problem. Where possible, it is more efficient to allocate the maximum memory needed at the beginning of the computation rather than resize arrays as the computation proceeds.

Arithmetic Failures. Arithmetic errors include divide by zero, square root, or logarithm of a negative number, and exponential overflow. Each potential failure should be tested before the expression is computed. If part-way through an iteration, the error should be suppressed (see section on Minimizing Error Messages) otherwise a meaningful error message should be displayed (for example, crossover in the computation of log–mean temperature). Most arithmetic failures can be avoided by appropriate numerical analysis (see section on Numerical Analysis) or program organization (see section on Arranging Expressions for Computation). An engineering computer program should never fail with a message generated by the computer operating system; such failures do not help the end-user.

Division into Component Parts. Programs divided into logical self-contained component parts are quicker to write and easier to test. However, component size must be chosen carefully. The benefits of componentization are lost with excessively large components. On the other hand, a program divided into excessively small components is dominated by interface programming. The interface programming introduces more lines of code and more potential bugs. It also increases run time and obscures the program logic. Considerations of program size and run-time should not dominate in deciding component size. It takes many man-years to write a megabyte of object code. The bulk of memory is taken by numerical data, text, and particularly graphics data. Any marginal space saved by sharing object code between programs would be completely

swamped by a saving that could be made by displaying a simpler graphic. Similarly, time savings by in-line coding (repeating source code in each routine to avoid the time taken to call common code through an interface) are negligible. The next hardware release will make much greater time savings. Time reduction can best be achieved by attention to the algorithm. For example, employ a polynomial algorithm rather than an NP algorithm, and amongst polynomial algorithms, choose one second order, rather than third order, in problem size. Component size should thus be made on the basis of clarity, simplicity, and maintainability rather than run-time and size.

Components can be provided as classes or functions that are linked as integral parts of the compiled program, or can be dynamically linked at run time. Dynamically linked components are favored where they are obtained from third-party sources. Dynamic linking is also favored for components with a distinct use that may be shared by several independent programs. Dynamic linking allows components to be developed and released independently of the main program. However, the resulting lack of release synchronization makes quality assurance more difficult, particularly when there are large numbers of dynamically linked components. There is a risk that users will employ a different set of components than has been tested with the delivered program. Consequently, dynamically linked components should be employed only when there is a strong case for using them. Object-oriented programming languages are designed to be modular, and small components can be written as classes and bound into several different programs. The majority of components are best provided as classes and functions within such object-oriented programming languages.

Checking Data. Erroneous output from a well-written program is most likely to be the result of erroneous data entered by the user. Programs should be made resistant to erroneous data both by checking the magnitudes of the values input and by checking the consistency of the data. An error message should be issued if the sign is wrong. For example, the sign convention for the "B" coefficient in the Antoine equation differs in different databanks. The coefficient should be checked to ensure that the sign is consistent with the convention in the program.

The most common data input error is confusion over units and dimensions. These errors can be trapped by checking that the numbers are within reasonable bounds. For example, a program designed to deal with liquid hydrocarbons should warn against density >3000 kg/m^3 or <300 kg/m^3. The density of water is 1000 kg/m^3, 1.0 g/cm^3 and 62.5 lb/ft^3. Consequently, this check will detect incorrect units for any common liquid. Similarly, incorrect conversion between metric units can be detected because most conversions introduce a power-of-10 error. Issue a warning rather than an error in case a user wants to model less common species. It is the end user's responsibility to check all warning messages (see section on Using Engineering Software). Additional tests are available for specific properties. For example, gas specific heats can be checked using the ratio C_p/C_v, with $1.0 < C_p/C_v < 1.7$. For gases, the Prandtl number is well predicted from the ratio of specific heats, which gives a further check on the consistency of heat capacity, viscosity, and thermal conductivity data. There are also a number of rigorous tests for thermodynamic consistency of data.

For data generated by the program, messages giving likely error bounds should be produced. Such messages should be given both for estimated data (eg, physical properties computed by the group contribution method) and for default values. Such warning messages serve the additional purpose of reminding the user that a default has been used. For example, a program designed for hydrocarbon liquids might employ a mean default viscosity, but generate a message indicating the possible range from lightest to heaviest.

Handling Uncertainty. Chemical engineers employ many semiempirical correlations, the potential errors in which should be notified to users. These correlations give uncertain predictions, even with accurate data. For example, pressure drop calculations are based on the work of Stanton and Pannell (9) updated by Moody (10). Turbulent heat transfer correlations are derived from the work of Dittus and Boulter (11), updated by McAdams (12). There is considerable experimental scatter about the empirical charts and equations put forward by these authors. It is recommended that programs should not add safety margins to account for the correlation uncertainties. The cautious "safe" bound depends on the user's application and it is the user's responsibility to apply safety margins. It is the programmers′ responsibility to make the uncertainties clear to the end user. Provision should also be made for the user to explore the effect of uncertainties on the results of the computation. In many cases, the effect can be simulated without direct access to the model or its built-in parameters. For example, the user can investigate the effect of an uncertain heat transfer coefficient by altering the corresponding area. In other cases, it may be necessary to give the user direct access to uncertainties in the model. For example, the user can be provided with a parameter that is set to 0.0 for the most likely result. It is set 1.0 for the high result at the 90% probability level and −1.0 for the low result at the 90% probability level.

Minimizing Error Messages. Most chemical engineering computations are iterative. The end-user is only concerned with the correctness of the final result. Error and warning messages should be suppressed until the final iteration. Such messages should then be output where they can be archived as part of a decision audit trail created by the end user. Where potential failure conditions arise during the iteration (eg, underflow or overflow), a suitable value should be generated to allow the iteration to continue in anticipation that the error will disappear as the solution is approached. The values generated should avoid introducing function discontinuities, which can have an adverse effect on convergence.

2.4. Numerical Analysis. A program that gives wrong answers or fails is unacceptable. This section introduces general principles and useful tools for avoiding or minimizing these problems. The principles are illustrated through specific examples. We concentrate on very simple cases that will be faced by the majority of engineer programmers. eg, we consider evaluation of expressions that might come to 0/0. This situation potentially arises wherever there is a division. There are many numerically difficult cases that are not treated in this chapter, eg, solution of stiff differential equations (those with a very wide range of time constants) or large sets of ill-conditioned algebraic equations. Programmers are referred to specialist texts on numerical analysis in such cases, eg, Epperson (13). Chemical engineers are most likely to include third party routines rather

than write such specialist methods themselves. There are a number of sources for such routines, eg, the HSL Library (14), the NAG Library (15), and Numerical Recipes (16). The basic advice provided here for the simple cases still applies for the more difficult problems; eg, it is still necessary to avoid numerical errors in evaluating expressions and to check on convergence.

Four areas that can give rise to numerical problems are covered.

1. Expressions that evaluate to 0/0.
2. Convergence of simple iterations. Both single variable and multivariable iteration is considered.
3. Solution of equations and matrix inversion.
4. Numerical solution of differential equations.

Expressions That Evaluate to 0/0. Consider a program for designing a countercurrent heat exchanger. The program employs a log–mean temperature difference. The simple method of programming the mean is to put:

$$\texttt{deltaTmean} = (\texttt{deltaT1} - \texttt{deltaT2})/\ln(\texttt{deltaT1}/\texttt{deltaT2})$$

Where

`deltaT1, deltaT2 and deltaTmean`

are the temperature differences at the two ends and the log-mean of these two differences.

The expression is impossible to compute if one of the following conditions apply:

1. The temperature differences have opposite signs at the two ends. Such a crossover is physically impossible. The program will fail with a "logarithm of negative number" error.
2. The temperature difference is zero at one or other end. The program will fail with a "divide by zero" error

   (`deltaT2 zero`) `or ''logarithm of zero'' error` (`deltaT1 zero`).
3. The temperature differences are the same at the two ends. The ratio

   `deltaT1/deltaT2`

   is unity, the logarithm of which is zero. The program then fails with a "divide by zero" error. This condition is thermodynamically most favorable and is trivial to compute by hand.

   In addition to the above obvious failures, significant errors arise when

   `deltaT1 and deltaT2`

   are similar. Consider the following case:
4. `deltaT1 differs from deltaT2`

   by a small fractional difference δ and the machine precision is ε. For floating point arithmetic, ε is a fractional error almost independent of the

absolute size of

`deltaT1 or deltaT2`.

The proportional error in both the numerator and the denominator is then ε/δ. These errors rarely cancel. If δ is sufficiently small, the error can exceed 100%. This situation is not as rare as it might seem. For example, an optimization can gradually bring the two temperature differences together until the error condition arises. This error is worse than the errors 1–3 because it gives a wrong answer with no warning.

Cases 3 and 4 are both numerical problems resulting from the finite precision of computer arithmetic. Cases 1 and 2 are more fundamental, and are treated below under "arranging expressions for computation". Solving case 4 automatically solves case 3 and it can be treated as follows. Writing the temperature differences as u and v, and the mean as y, gives:

$$y = (u - \mathrm{v})/\ln(u/\mathrm{v}) \tag{1}$$

The general approach is to expand the terms that approach zero as a series. The denominator then becomes

$$\ln(u/\mathrm{v}) = 2[(u - \mathrm{v})/(u + \mathrm{v}) + \{(u - \mathrm{v})/(u + \mathrm{v})\}^3/3 + \cdots] \tag{2}$$

Substituting equation 2 into equation 1 gives:

$$y = 0.5(u + \mathrm{v})/[1 + \{(u - \mathrm{v})/(u + \mathrm{v})\}^2/3 + \cdots] \tag{3}$$

Equations 1 and 3 are alternative ways of computing the mean. For small values of the difference, the term in curly brackets is given by

$$(u - \mathrm{v})/(u + \mathrm{v}) = \delta/2$$

When δ is small, it is apparent that equation (3) gives the correct result. For larger values, the truncation error steadily increases. Equation 1 has the opposite behavior; for small values of δ, round-off gives erroneous results, but it is accurate when δ is large. Figure 2 shows the dependence of round-off and truncation error on δ for single precision on an IBM compatible PC. To maintain maximum precision throughout, equation 3 should be used when it is more accurate than equation 1. The proportional errors are, respectively.

Equation 1, round-off error resulting from finite precision in computer arithmetic:

$$r = \mathrm{abs}(\varepsilon/\delta)$$

Equation 2, truncation error in taking only a finite number of terms in an infinite series:

$$t = \delta^2/12 \tag{4}$$

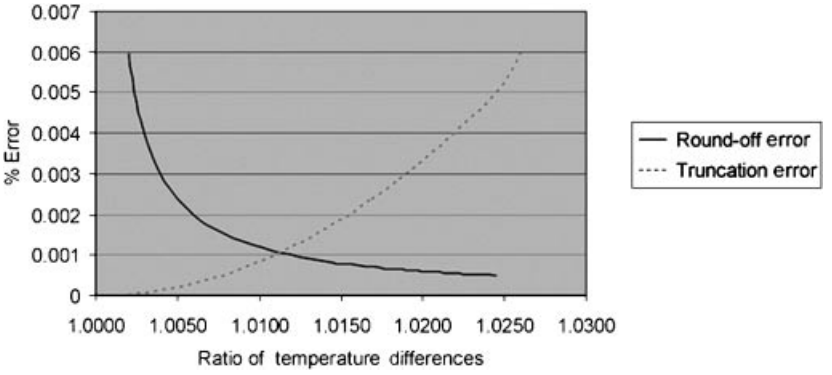

Fig. 2. Errors in computing log–mean temperature differences.

Equation 4 is derived from equation 3 by taking only 1 term in the series so that the first term ignored is $\{(u-\mathrm{v})/(u+\mathrm{v})\}^2/3$. Equation 3 is preferred to equation 1 when $t < r$. Namely when

$$\mathrm{abs}(\delta^3) < 12\epsilon \tag{5}$$

Taking only the first term of equation 3 gives the following simple expression:

$$y = 0.5(u+\mathrm{v})$$

Thus, for small temperature-difference ranges, it is more accurate to use the arithmetic mean as an estimate of the logarithmic mean than to evaluate the logarithmic mean directly.

The value of ε is obtainable either from documentation on the compiler used, or directly from facilities available as part of the computer language employed. For example, in C++, the value is given by

```
eps = numeric_limits::epsilon();
```

and

```
eps = numeric_limits::epsilon();
```

depending on whether single or double precision computation is employed. On a PC, the value of

```
eps
```

for single precision is 1.19207e-7. The worst error using equation 5 is then

$$r = t = \epsilon^{2/3}/12^{1/3} = 1.058e-5$$

Thus, even always choosing the best method of computing the mean still gives a maximum error 100 times worse than machine precision. Figure 2 shows the truncation and round-off errors plotted against ratio of temperature differences for a PC in single precision. This figure shows that it is more accurate to use the arithmetic mean up to differences in excess of 1%.

In double precision, the value of eps for a PC is 2.22045e-16, and the worst error is 1.6e-11. In this case, the computer precision is reduced by a factor of 100,000.

If the resulting precision is not adequate, the error can be further reduced by taking the first two terms of equation 3. Equation 3 can now be used over a wider range up to

$$\text{abs}(\delta^5) < 40\epsilon$$

On a PC, the maximum error is then given by

$$r = t = \epsilon^{4/5}/40^{1/5} = 1.382\text{e} - 6 \text{ (single precision) or } 1.1435\text{e} - 13 \text{ (double precision)}$$

The loss in precision is then reduced to a factor of 10 in single precision and a factor of 100 in double precision.

These considerations of numerical precision apply equally to equation-based systems. Such systems should employ a log–mean function coded as above, or an approximate log–mean that cannot accumulate numerical error, eg, the Underwood mean

$$y = [(u^{1/3} + \text{v}^{1/3})/2]^3 \tag{6}$$

Care must be taken in implementing equation 6 to ensure that the cube root function employed is capable of computing the roots of negative numbers.

This example illustrates a general point in engineering software. Wherever there could be a divide by zero error, check to see whether there is a definite value at the zero point. If there is, large errors are likely when the numerator and denominator are near zero. It is then necessary to make a series expansion; in many cases of both the denominator and the numerator. The resulting expansion can be used to find the range in which it is better to use the first few terms of the expansion rather than use the expression that evaluates to 0/0. It is also possible to determine how many terms of the expansion to employ to achieve a desired precision.

Convergence of Simple Iterative Schemes. Iterative methods are used to solve many engineering problems. For a single variable problem, successive estimates can be written

$$x + e_1, x + e_2, x + e_3, x + e_4, \cdots$$

Where x is the correct result and e is the error. The term $(x + e_1)$ is the first guess of the solution. For n th order convergence, successive errors are given by

$$e_{i+l} = ke_i^n$$

where k is a constant coefficient. If, for any i, $ke_i^n < e_i$, all subsequent errors decrease and the scheme converges.

First order convergence ($n = 1$). is commonly found in chemical engineering computations. (It only takes one first-order step in an otherwise second-order scheme to reduce the order of convergence.) For first order convergence,

$$e_j = k^j e_o$$

Many chemical engineering programs accept convergence if

$$f = \text{abs}\ (x_i - x_{i-1}) = \text{abs}\ (k^i e_0 - k^{i-1} e_0) < c$$

where c is some convergence criterion.

The corresponding error may, however, be much larger than c. Thus

$$e_i = kf/(1 - k) \tag{7}$$

The rate of convergence (measured by k) is likely to change from data set to data set. On slowly converging data sets, eg, with $k = 0.9$, the error is likely to be large compared to the convergence criterion c. The recommended strategy is to compute k (from f_i/f_{i-1}) and average it over several iterations. The error given by equation 7 can then be estimated and a consistent precision accepted on convergence.

Where the magnitude of k consistently decreases from iteration to iteration, convergence is better than first order. For higher-order convergence, equation 7 gives a conservative estimate of residual error (thus, the actual error is less than the value computed by the equation).

For a multi-variable problem with m variables, f in equation 7 is replaced by

$$f_i = \sqrt{\sum_j (x_{j,i} - x_{j,i-1})^2 / m}$$

e_i is then a measure of the mean error of the variables. For each iteration, k is computed from

$$k = f_{i+1}/f_i$$

Where higher order convergence is achieved, this procedure gives a conservative estimate of residual errors.

If e_i cannot be reduced to zero, it indicates either that the set of equations has no solution or that there is an accumulation of numerical error. In either case, the numerical methods discussed in the following sections may be applied.

Solution of Equations and Matrix Inversion. There is a frequent requirement in chemical engineering to solve sets of equations. For example, applications arise in modeling complex multiple reaction processes, in statistical analysis fitting experimental data, and in balancing flowsheets including recycles. Frequently, the equations are nonlinear; such sets of nonlinear equations are solved by successive linearization. Thus, solution of sets of linear equations is central to much chemical engineering computation.

Where a single nonlinear equation is solved, the programmer will normally write the solution routine. The solution point should first be bounded by two points with residuals of opposite sign. These bounds should be reduced at each iteration as new points are computed. If a next iterate indicates a solution outside the bounds, the basic iteration should be replaced by a simple division algorithm (eg, halving) that generates a potential solution within the bounds. This fall-back algorithm will be invoked when the local linearization has a low, or zero, slope and thus predicts a solution far from the current point. Bounding in this way thus automatically avoids divide by zero. It is equally applicable to Newton's method (the linearization is the tangent line at the most recently computed point) or the secant method where the linearization is a line joining two points on the nonlinear residual curve.

Nonspecialist engineers will solve large sets of nonlinear equations using third party tools that have been optimized to minimize accumulation of numerical errors. However, when used as part of a larger iterative scheme, such third party tools can fail either because a set of equations having no solution has been generated, or because of accumulation of numerical errors. Steps can be taken to minimize such failures and we outline here some of the steps that can be taken.

It is first necessary to have some understanding of the problem characteristics that lead to solution failures. There are two cases in which solution methods frequently fail. The first is when the equations are redundant; the second is when they are inconsistent. The cases can be illustrated by reference to the following sets of equations.

Consider the equations

$$2x_1 + 3x_2 = 2$$
$$2x_1 + 3x_2 = 2$$

and

$$2x_1 + 3x_2 = 3$$
$$2x_1 + 3x_2 = 1$$

The first set shows redundancy (the same equation is repeated twice) and there are an infinite number of solutions. The second shows inconsistency and there are no solutions. In an iterative scheme to solve a nonlinear set of equations, it would be hoped that this situation would not occur at the final solution point. (Such inconsistent problems can easily arise, eg, in modeling systems including distillation. During iteration, it can occur that 100% of a steadily produced or fed component exits from the top and is recycled. There can be no material balanced solution until other values have been modified so that $<100\%$ is recycled.) We require a strategy that will allow the iteration to continue. A suitable strategy is to modify the equations as follows. Write the set of equations

$$2x_1 + 3x_2 = 3 + \epsilon_1$$
$$2x_1 + 3x_2 = 1 + \epsilon_2$$

Now solve the problem that $\Sigma\epsilon_i^2$. is minimized. This step converts an inconsistent set of equations to a set that can be solved or (as in this case) a set that contains redundancy. Redundancy can be removed by adding the constraint that Σx_i^2 is minimized. This constraint must have a much lower weight than the constraint to minimize the errors in the right-hand sides, or an incorrect solution will be obtained when a correct solution is possible. Depending on the physical situation being modeled, this strategy can be modified to produce answers with any desired properties without compromising a correct solution where that is possible. Modifying the problem can thus eliminate difficulties for third-party solvers.

The most frequently used approach to solving sets of linear equations is LU factorization [see *Perry's Chemical Engineers' Handbook* (17)]. Whether or not the inverse of the matrix of x coefficients is also produced, the method is third order in problem size. Thus, run time is proportional to the cube of number of equations. The discussion above relates primarily to this solution strategy. However, a number of chemical engineering programs employ an alternative method when solving sets of nonlinear equations which, in principle, is faster. It is noted that the matrix to be inverted only changes slightly at each successive linearization. Instead of completely reinverting the matrix, the inverse can be directly updated (eg, by quasi-Newton or by methods that exactly replace one row or column). These updates are second order, so that the relevant linear equations can be solved by a second-order rather than a third-order method. With >10 equations, the method can be significantly faster. A disadvantage of the updating method is that the quality of the inverse can progressively deteriorate as errors accumulate. This section presents a simple procedure for checking and refining inverse matrices.

The most direct test that matrix B is the inverse of matrix A is to form the difference

$$D = AB - 1$$

$$D_{ij} = \sum_k A_{ik}B_{kj} \qquad i \neq j$$

$$D_{ii} = \sum_k A_{ik}B_{ki} - 1$$

The size of d can be measured in various ways. The simplest is to find d, the root-mean-square size of the elements. Thus, for an $m \times m$ matrix

$$d = \sqrt{\sum_i \sum_j D_{ij}^2/m}$$

If d is comparable with machine precision, B is a good estimate of the inverse of A. If d is too large, an improved estimate of B is given by

$$B' = B(1 - D)$$

If $d < 1/m$, it is guaranteed that $d' < d$, where d′ is found from the matrix AB'. This convergence criterion is conservative and B' may be better than B, even when the condition is not met. The refinement can be applied iteratively to achieve sufficient accuracy from any suitable inverse estimate. The method is simple and safe to program because it requires no divisions. It is, however, third order and slower than recomputing the inverse by LU factorization.

Similar methods can be employed for handling matrix inversions that arise in nonlinear optimization and statistical analysis.

To avoid numerical errors during a larger iteration, it may be desirable to convert non-invertable matrices to invertable matrices. Similar adjustments can be made to those described above for solving redundant or inconsistent equations.

Solution of Differential Equations. Numerical analysis textbooks describe a number of robust methods for solving differential and partial differential equations and many of these methods are available in computer program libraries, such as those referenced above. It is beyond the scope of this chapter to classify and describe these methods. In general, it is recommended to consult such specialist sources in developing solutions to engineering problems. However, chemists and engineers frequently employ simple discretization when solving problems such as tracing composition changes through a plug-flow packed-bed reactor, or the progress of a reaction in a well-stirred batch. This simple discretization is equivalent to a forward-difference (forward Euler) solution procedure. The poor numerical performance of the method is well known. We present here, a modification that can improve the performance of simple discretization methods without employing more powerful library methods. The Central difference (or Tapezoidal rule) method is described.

In order to solve a single differential equation to give f as a function of t, the forward difference method works as follows: The parameter f and its derivative df/dt are computed at time t. An estimate of f at the next time increment is then

$$f_{t+\delta t} = f_t + (df/dt)_t \delta t \tag{8}$$

The value of f at time $(t + \delta t)$ enables the derivative at this point to be computed and the integration continued. The procedure is simple and requires no iteration. However, it has several disadvantages. Even without accumulating numerical error, it can predict unstable oscillatory behavior for systems that are stable in practice.

Improved performance can be obtained by replacing the derivative computed at t by the mean of the derivatives computed at t and $(t + \delta t)$. The new formula is

$$f_{t+\delta t} = f_t + 0.5[(df/dt)_t + (df/dt)_{t+\delta t}]\delta t \tag{9}$$

The computation proceeds as follows. The value at time $(t + \delta t)$ is computed as for the forward difference case. The derivative at the new point is computed and substituted into equation 9 to give an improved estimate. The iteration is continued to convergence. The value of f that solves the equation may be obtained by established methods such as the Secant or Newton method, or, if the derivative expression is sufficiently simple, analytically.

Where there are a large number of elements in f, the central difference method thus requires the solution of a set of (in general) nonlinear algebraic equations. Process simulators include powerful built-in methods for solving such equations. These built-in solvers have been successfully employed to model three-dimensional (3D) reaction and heat transfer using central differences within a conventional dynamic process simulator.

The benefits of central difference are that the same precision can be achieved with fewer (longer) steps and the method does not show spurious unstable behavior.

Where error checking is not a part of the integration routine employed, the precision of integration should be checked. For an nth order integration, the error is given by

$$\epsilon = k\Delta x^n$$

where ε is the error and Δx the step-length for integration. For a first-order integration, the error per step varies as the square of step length. The total error is the error-per-step multiplied by the number of steps. The number of steps is inversely proportional to step length. Hence, the net effect is that total error is directly proportional to step length.

A simple check that can be put into any engineering program is to repeat the integration with twice the step length. Two error estimates result:

$$y_1 = y + k\Delta x$$

and

$$y_2 = y + 2^n k\Delta x^n$$

Combining the two equations, gives

$$y_1 = y + (y_2 - y_1)/(2^n - 1) \tag{10}$$

The error in the most accurate integration is thus less than or equal to the difference between the two estimates. Adding this double step-length integration increases computational time by 50% for simple integration and by 25% for two-dimensional (2D) integration. The cost in computational time is thus relatively low in order to provide a measure of quality control for the integration. Where there is confidence in the order of integration, equation 10 can be used to obtain a better estimate of y. (For $n = 2$, this is known as Richardson's h-squared method). Where there is doubt about the order, it is cautious to assume a higher order for predicting a solution but a lower order for estimating the error. Theoretically, forward difference integration is first order and central difference integration is second order.

2.5. Arranging Expressions for Computation. The general principles to be applied in arranging equations for computation are as follows:

1. The equations should be as nearly linear as possible over as wide a range as possible.

2. The equations should be computable for all values of the right-hand side variables.
3. Explicitly computable expressions should be preferred to expressions that need to be solved iteratively.

Linearity is required because expressions frequently form part of larger iterative schemes. Such iterations are nearly always solved by local linearization (eg, Newton–Raphson iteration). The size of the region within which rapid convergence is achieved is determined by the size of the region within which the relevant equations are (nearly) linear. These considerations apply equally to conventional (assignment-type) programs and to equation-based modeling systems.

If the right-hand side (or equation) is not computable, no iterative scheme can make sensible progress to a converged solution.

Explicit expressions are preferred both because they are faster and because the risk of an iteration failing to converge is eliminated.

In order to put the equations into the best form for computation, it may be necessary to derive them from first principles rather than employ a conventional textbook formula. The general principles will be illustrated by reference to two specific examples, heat exchanger simulation and isothermal flash simulation.

For heat exchanger simulation, the equation most often seen in chemical engineering literature is

$$Q = UA\Delta T$$

where ΔT is the logarithmic-mean temperature difference. This equation has been used extensively in simulation, optimization and even process synthesis studies. The heat exchanger is simulated by estimating one of the exchanger outlet temperatures, computing the other by heat balance, and hence determining the log–mean temperature difference. The log–mean temperature difference is used to compute the heat transferred. The outlet temperatures are then recomputed and the iteration continued. This iteration is seen explicitly in modular simulators but, although present, may not be immediately obvious in equation-based simulators.

An improved formulation can be obtained by noting that the log–mean temperature is derived from an analytical solution of an ideal countercurrent (or cocurrent) exchanger with constant physical properties. This analytical solution can be employed directly to avoid the temperature iteration, thus

$$T_{\text{out}} = aT_{\text{in}} + (1 - a)t_{in} \tag{11}$$

where

$$a = p(1 - S)/(1 - pS)$$

$$p = \exp\{UA(S - 1)/(M_h C_h)$$

$$S = M_h C_h / M_c C_c$$

where T is hot stream temperature, t cold stream temperature, M is mass flow rate, C is specific heat, and subscripts and c and h correspond to cold and hot streams.

There is a similar equation for outlet cold temperature. When $S \approx 1$, equation 11 shows the same numerical problems as for equation 1 when u ≈ v. A similar expansion is required to compute a for this case.

We have achieved the following goals:

1. T and t are linear functions of the inlet temperatures
2. The temperatures are computable for all right-hand side variables. Specifically, the temperature cross-over problems associated with computing log–mean temperatures are eliminated.
3. The expressions for outlet temperature are explicit and not iterative.

Equation 11 is particularly useful in computing the temperature distribution in a network with fixed fluid flows. The temperature distribution is determined by linear equations and the complete temperature distribution can be obtained quickly and noniteratively.

For nonconstant physical properties, the equation gives the same errors as are incurred by employing the log–mean temperature. (It is based on exactly the same treatment, so this behavior is to be expected.)

The log–mean temperature is more frequently seen in the literature because it is suitable for hand "design" calculations. The computer-aided approach is to design through simulation. Thus, a design is hypothesized that is then simulated. Sizes and operating conditions are adjusted to improve performance.

For the heat exchange problem, it is possible to generate equations that are exactly linear in the required (temperature) variables. Where such an exact linearization is not possible, it is still desirable to make the equations as nearly linear as possible. The following example illustrates application of the principle to such a problem, namely, the simulation of a simple isothermal flash.

In the simplest isothermal flash simulation, an ideal mixture of known composition and feed rate is fed to a vessel held at constant temperature. The feed splits into liquid and vapor phases, and the objective is to determine the proportion and composition of each phase. In the simplest case, the vapor mole fraction, y_i, is computed from the liquid mole fraction, x_i, by the k-value relationship assuming that k-values depend only on temperature, thus

$$y_i = k_i x_i$$

An iterative solution is employed starting with an initial estimate of the liquid fraction, u. Material balance gives

$$z_i = ux_i + (1-u)y_i = ux_i + (1-u)k_i x_i \tag{12}$$

where z is the feed mole fraction.

Rearranging equation 12 gives

$$x_i = z_i/[u(1-k_i) + k_i] \tag{13}$$

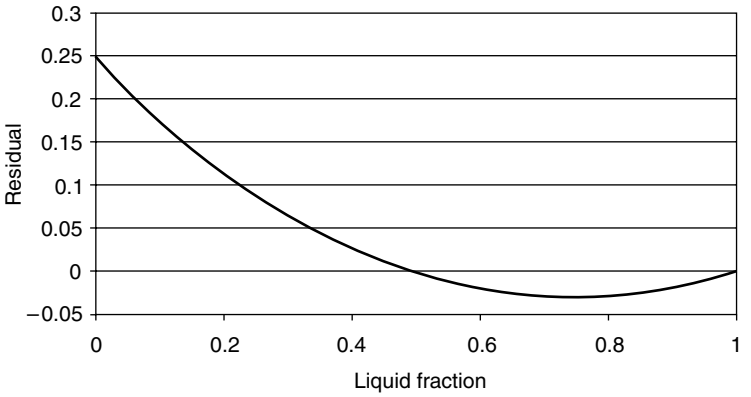

Fig. 3. Isothermal flash residuals: Liquid mole fractions sum to 1.

The liquid mole fractions sum to 1. Hence, equation 13 gives

$$1 = \Sigma z_i/[u(1-k_i)+k_i] \tag{14}$$

Equation 14 is a single variable equation for u. The solution can be substituted into equation 13 to obtain the liquid mole fractions, and hence also the vapor mole fractions. The flash can thus be fully computed.

Equation 14 does not meet our criterion of a near-linear relationship. Figure 3 shows a typical plot with a solution at $u = 0.5$. It is seen that there is a spurious solution at $u = 1.0$. Any iterative scheme shows poor convergence properties if the initial estimate is >0.5, and it can converge to the wrong solution. These difficulties are equally experienced in conventional programs and in equation-based modeling systems.

A more nearly straight line is obtained by taking the difference between the vapor and liquid mole fractions, which results in equation 15:

$$0 = \Sigma(k_i-1)z_i/[u(1-k_i)+k_i] \tag{15}$$

The corresponding curve is shown in Figure 4. Differentiation of equation 15 shows that it is monotonically positive, and nearly linear, for all compositions and k-values.

Equation 15 gives the additional benefit that the end points correspond to the relevant dew and bubble-point conditions. Thus, if the function is zero at $u = 0.5$, the mixture is all-liquid, just at its bubble point. A positive value corresponds to a single-phase liquid below its bubble-point. Similarly, if the function is zero at $u = 1.0$, the mixture is all-vapor at its dew point, and negative values correspond to a gas-phase above its dew point.

The linearity of equations can often be improved by judicious choice of variables. For example, in computing bubble and dew points, the relationship between the logarithm of the total pressure and $1/T$ is more nearly linear than the relationship between P and T. In other cases, it is often better to chose partial pressures than total pressure and mole fraction as variables.

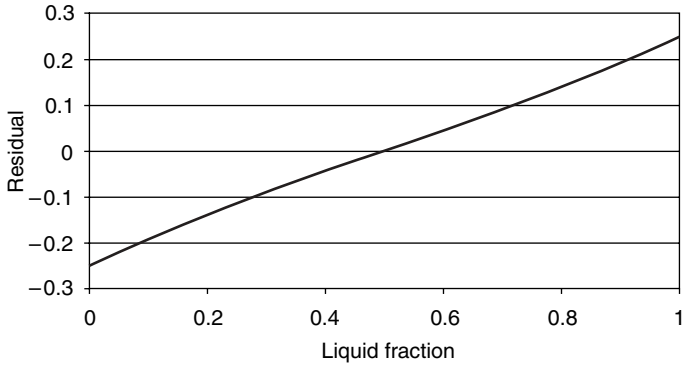

Fig. 4. Isothermal flash residuals: Sums of liquid and vapor mole fractions equal.

3. Using Engineering Software

The moral and professional responsibility for engineering decisions rests with the engineer making the decisions rather than the authors of any software employed. Usually, the legal responsibility also rests with the engineer making the decisions. Many of the topics introduced in this section are described more fully by Best et al. (18).

Engineers using computer programs written by others must thoroughly understand the application to which the program is applied. They should provide a decision audit trail so that all recommendations can be checked. The audit trail should include

1. A clear statement of the problem being tackled.
2. A statement of the assumptions made and their justification.
3. A review of the software applicable to the problem.
4. Identification of the chemical species that might arise.
5. Review of the data sources and the range of temperature, pressure and composition over which they are valid.
6. Review of the models employed by the software, their validity and applicability.
7. Estimation of the errors that might be introduced through the data or the models.
8. Sensitivity tests to assess the consequences of possible data or model errors.
9. The alternative solutions that have been generated.
10. A critical assessment of the performance and risks of the alternative solutions.
11. The recommended solution.

The audit trail should also include all error messages generated by the software used and a critical assessment of the implications of the messages.

The statement of the problem is the vital starting point for the study. Economic, safety, and environmental goals should be clearly stated. For chemical or petroleum production, requirements for product purity and production rate should be stated and the reasons for these requirements recorded. For example, the end uses of the product may be given and the consequences of impurities noted.

The software available for the study should be listed. One study might employ up to 30 separate programs. These may range from simplified modeling and synthesis software for generating a range of possible process variants, through detailed process simulation, to software for mechanical design, safety assessment and environmental impact assessment. Each potential program should be assessed against criteria of suitability for purpose; these criteria are discussed in more detail below. The extent to which each program has been validated should also be noted. General-purpose programs obtained from third-party suppliers are generally well validated. However, they will not have been validated for the specific problem to which they are to be applied. The end-user thus has to assess the relevance of the prior validation and may need to formulate further tests.

All software needs to be validated by the user as well as the writer (6,18). The documentation should fully describe the models employed and give clear guidance on use of the program. If this information is deficient, it is indicative that the program may also be deficient. Where the company employing the engineer has in-house standards, company-validated software should be employed. The engineer should clearly understand the phenomena being modeled by the software. It is not possible to take responsibility for decisions made in ignorance. Companies should retain a consultant, or in-house expert, on any software that they use. In this way, all users can consult experts who fully understand the software employed. The program should be verified for simpler systems for which results are known in advance. The validation is then gradually elaborated until the required results are obtained.

An important part of any study is to identify the chemical species that might occur; no computer program can model components omitted from the data. Minor components can build up if there is no way of discharging them. There are cases (particularly with batch process) where minor components have caused explosions and toxic releases. Such minor components can also effect the properties of mixtures, eg, causing or breaking azeotropes.

For each chemical component, the reliability of the data available should be reviewed and data sources identified. Where the properties are represented by parameters in correlating equations, the valid range of the correlations should be established to ensure that they apply to the specific problem to be solved. Error bands for the values predicted by the correlations should be established. Where values outside the fitted range are required, extrapolations should be based on fundamental thermodynamic principles and application of theory with a sound scientific basis, such as the kinetic theory of gases. Many correlating equations are polynomials with little sound scientific basis. An extrapolation based on scientific principles will be more reliable than extrapolating a polynomial formula from a database. Where values are extrapolated, error bounds should also be set by reference to scientific principles. Particular care should

be taken when conditions near to the critical point arise. Similar considerations apply to data that is estimated when experimental data is lacking. It should not be assumed that data from well-known reference books are reliable. These books expressly deny responsibility for errors. Data from such sources should be checked for thermodynamic consistency and for consistency with data for similar chemical species. Similar considerations apply to data and correlations supplied with commercial software. The reliability of such data should be determined either by entering a confidentiality agreement with the software supplier or by comparing predictions with experiment and data from other sources.

It is recommended that physical property predictions are displayed graphically to provide clearer comparison than is available from a table of figures. Mixture properties need to be assessed in the same way. In studying binary mixtures, validation should include very low concentrations of each component. Simplified theory is available for low concentrations and discontinuity errors (where the mixture equations differ from the pure component equations) are more likely to be apparent.

Correlations and data should be applied only under the conditions for which they have been established. For example, if the correlations have been derived for non polar mixtures, they are unlikely to be valid for polar mixtures. The use of computers does not obviate the necessity for experimental investigation. Where a mixture has not been studied previously, experimental confirmation of predicted properties may be needed.

The models employed should be similarly validated. Software suppliers should give full details of the models that they employ. The applicability of the models needs to be checked. For example, if the pressure of a flowing gas changes by >10 or 20%, compressible flow models are needed. As for physical property data, models should be checked for continuity at points where the model might change. For example, at low liquid concentrations, the pressure drop for a two-phase gas–liquid mixture should approach the pressure drop for a pure gas for low liquid concentrations.

Start-up as well as steady performance may have to be assessed. For example, a significant number of environmental discharge contraventions occur during start-up when catalysts have not reached operating temperature and separation systems are not working to full efficiency. Care must be taken that non-steady simulation models are adequate for modeling start-up. Dynamic models developed for regulation control studies are rarely adequate for start-up and shut-down studies.

Every engineering system is built in the face of uncertainty. It is the engineer's responsibility to ensure that the uncertainties are understood and scoped. Uncertainties arise in the data and in the models within the programs employed. Uncertainties extend to the possible hazards, reliability, and environmental impact of the process studied. Once the uncertainties have been identified, sensitivity tests can be undertaken to assess their impact. The consequences can be classified into safety hazards, environmental impact, operability, and economic impact. The uncertainties should include allowance for operational flexibility. Sensitivity tests can be integrated to indicate the combined effect of multiple uncertainties (see the section Integrating Multiple Uncertainties).

Alternative solutions should be considered. For example, in generating a process design, alternative processes should be generated, with alternative flowsheet structures and alternative unit operations. In many countries, it is a requirement that genuine alternatives must be evaluated and the alternative with the best environmental performance selected. The proper evaluation of these alternatives is thus a vital part of the decision audit trail leading to the final design. The impact of uncertainties on the alternatives should also be evaluated; some alternatives may be more sensitive to uncertain data than others.

Brief notes on the special considerations in using spreadsheets and neural networks are appended to this section, as is a procedure for integrating the effects of multiple uncertainties.

3.1. Use of Spreadsheets. As discussed in the section Programming Languages, quality assurance of results obtained from spreadsheets requires special attention. In particular, there is the potential for the user to inadvertently change formulas in cells. Strict guidelines on spreadsheet use should be issued. Caution is required for safety-critical applications. The decision audit trail should include a full copy of the spreadsheet program, not just its results.

3.2. Neural Networks. Neural Networks contain a large number of fitted parameters. Statistically, the larger the number of parameters, the less significance any such fit has. In many cases, the statistical significance of Neural Net predictions is minimal. Consequently, the Net may be an effective method of interpolating the conditions in which it was trained but it may not be a good basis for predicting behavior outside the training region. Applications in control may be valid because there is a constant stream of data enabling retraining to be undertaken and the data is likely to span the conditions being predicted. In other applications, however, engineers should use neutral networks with caution.

3.3. Integrating Multiple Uncertainties. Extensive sensitivity tests are recommended above. Sensitivity tests are normally undertaken by simulation with the "best estimate" values of the uncertain parameters and with perturbed estimates. In this way, the sensitivity of performance to the assumptions can be assessed. The uncertainties should include both data uncertainties and model uncertainties (see the section Handling Uncertainty). Where specific interactions are important, users may explore perturbing several uncertain parameters in the same simulation. However, the combinatorial problem of exploring all possible combinations of uncertainties makes it impracticable to explore more than a few multiple uncertainty cases. There is much research on handling multiple uncertainties, particularly optimal design under uncertainty. However, there is no widely agreed tool. In this section, we present a tool that has been employed successfully in a number of studies and is simple to apply. In the future, better tools may be available.

The expressions given below are applicable to models that, in the region of the expected solution, can be approximated to cubic expressions (including interaction terms). The uncertain parameters are distributed independently and symmetrically. The expected value of a performance measure, y_e, can be then obtained from

$$y_e = y_0 + \Sigma[y\{\Delta x_i\} + y\{-\Delta x_i\})/2 - y_0](\sigma_i/\Delta x_i)^2 \qquad (16)$$

The variance, σ_y^2, of the performance measure is given by

$$\sigma_y^2 = y_0^2 - y_e^2 + \Sigma[(y^2\{\Delta x_i\} + y^2\{-\Delta x_i\})/2 - y_0^2](\sigma_i/\Delta x_i)^2 \quad (17)$$

Where, y may be any uncertain performance measure (component concentration, stream temperature etc). The parametter y_0 is the value of y computed with all uncertain parameters at their expected values. $y\{\Delta x_i\}$ is the value of y computed with all uncertain parameters at their expected values except for variable number i, which has its value at $(x_{0i} + \Delta x_i)$. The summations are over all the uncertain parameters. The parametter σ_i is the standard deviation of parameter i.

Where there are n uncertain parameters, $(2n + 1)$ simulations are required to compute all the uncertain outcomes. Thus, the total number of computations is independent of the number of performance measures to be assessed.

In engineering design, most parameters have independent uncertainties. However, interdependencies can arise. For example, an experimentally measured rate constant (K) may typically be computed using

$$K = Ae^{-E/RT} \quad (18)$$

In this expression, both the pre-exponential A and the activation energy E will be uncertain. Note that high reaction rates can be obtained either by increasing A or by decreasing E. Consequently, there is uncertainty as to whether there is a high pre-exponential or a low activation energy, and this uncertainty will be higher than the uncertainty in the predicted rate constant. An error ellipse can be drawn around the best estimate of the two parameters. It will show a large probability that the pre-exponential is higher than the expected value and the activation energy is also higher, but there is a low probability that the pre-exponential is high and the activation energy is low. There are two ways of treating such parameters, the values of which cannot be estimated independently. The first is to replace A and E in the study by a variable that goes along the major axis of the error ellipse and a variable that goes along the minor axis. These two variables are statistically independent. The second approach is to put

$$K = K_0(1 + s)$$

where K_0 is computed from equation 18 and s measures the scatter of the experimental observations about the predicted line. In this case, the two correlated uncertain parameters are replaced by one uncertain parameter, s, with a mean value of zero.

The uncertain distribution functions for most parameters found in engineering studies are, to sufficient accuracy, symmetrical. However, there is obvious asymmetry in parameters such as a mass transfer coefficients. These may have large uncertainties, but they clearly cannot be negative. It is found that the logarithms of such parameters are statistically sufficiently symmetrically distributed. Such logarithmic transformations can be applied equally to parameters x or y in equations 16 and 17. In most engineering problems, estimates of standard deviation are little more than guesses. In comparison, the

form of the error distribution is a relatively second-order consideration and does not need to be treated in more detail.

This approach to integrating uncertainties can be built into software. However, more often it will be used as part of the quality assurance tests employed by the end user. For example, if the engineer records sensitivity results in a spreadsheet, equations 16 and 17 can be used to generate additional columns giving the integrated effect of uncertainty.

4. Current Advances

There are number of areas of computer-aided chemical engineering that are gaining in importance as the pressure for a more environmentally friendly chemical industry grows. This survey does not claim to be comprehensive. It gives a few of the important developing areas, namely, computer-aided molecule design, process synthesis, and flexible design.

4.1. Computer-Aided Molecule Design. CAMD has been developing rapidly during the last 10–15 years with an increasing number of successful applications. Techniques such as the group contribution method enable the properties of molecules to be predicted before they have been synthesised. CAMD exploits these abilities to design molecules that have desired properties. Recent developments and applications are given by Harper and Gani (19). Initial applications have been in the development of selective solvents; particularly non halogenated solvents with reduced toxicity and reduced ozone depletion potential. Potential applications include the development of economic, effective, safer, and less polluting chemical products.

4.2. Computer-Aided Process Synthesis. Process syntheses has been studied for over 30 years but, until recently, applications have been limited to energy reduction studies. Process synthesis differs from process optimization in the variables selected for optimization. Process optimization adjusts only continuously variable parameters such as lengths, temperatures, and pressures. Process synthesis optimizes also discrete variables. Recent advances in computer hardware and software promise a much wider range of application.

In a sense, all process design is process synthesis. Thus, the designer selects

The sequence of operations.

The choice of unit operations (eg, extractive distillation or liquid–liquid extraction).

The selection of hot- and cold-stream heat exchange matches.

The choice of separating agents.

In industry, design is still usually undertaken by hypothesizing a flowsheet that is incrementally improved as a result of simulation and other studies. However, Johns (20) summarizes a range of current computer-aided synthesis techniques that show promise of getting better results by automating more of the conceptual design process. Many give a feasible design as a direct output of optimization. Others, particularly Pinch Technology (21), deliver performance

targets as goals for subsequent detailed design. Rigorous integer optimization is difficult, and is not practicable for many industrially relevant problems. Artificial intelligence methods aim to solve problems not treatable by rigorous optimization. Such heuristic (AI) methods develop good, although not necessarily optimal, designs. Automatic optimization, whether mathematical or heuristic, can only consider a limited number of performance criteria (eg, just cost). In practice, a design will be judged also by other criteria such as operability, safety, and environmental impact, many of which cannot be adequately included as constraints or objectives in the optimization. It is, therefore, desirable that the optimization produces a number of alternative flowsheet structures that can be evaluated against these additional criteria.

Robust simulation models are required for both optimization and synthesis. The optimizer is likely to generate sizes and/or operating conditions that are outside the normal simulation range and it is important to avoid run-time errors. Optimizations should not set constraints that may be difficult to meet. For example, instead of setting a minimal product purity constraint, it may be better to set a realistic cost penalty for purity shortfall. The optimization will be easier (because the function will be smooth up to the desired purity). Furthermore, if the purity constraint cannot be met, the user has a meaningful result, instead of a failure message.

The flowsheet structures obtained by process synthesis are frequently insensitive to uncertain commercial and technical data. Failure to achieve an optimal flowsheet structure is more likely to result from shortcomings in the optimizer than uncertainties in the data. Where the choice between two structures is sensitive to an uncertain parameter, it is desirable to consider both alternatives against other criteria not included in the optimization. This insensitivity enables simplified models to be employed for determining the range of likely process structures. Each structure is then subsequently simulated and optimized using relatively rigorous models. Simplified models can be made more resistant to run-time error and can be made more than a thousand times faster than rigorous models. Tools are available for tuning simplified models against rigorous models.

The benefit of computer-aided process synthesis is that very large numbers (eg, millions) of process alternatives can be implicitly evaluated. The evaluation identifies processes that are economic and have reduced environmental emissions. It is impracticable to evaluate such a large range of alternatives by hand. Regulatory authorities increasingly demand that processes are evaluated to ensure that there are not competitive alternatives with lower environmental impact. Process synthesis enables these demands to be met in a rigorous manner.

4.3. Flexible Design. Parameters such as physical size are expensive to change after a process plant is built. However, flow rates, temperatures, and pressures can be changed. Thus, if the desired purity cannot be achieved at the nominal throughput, it may be possible to achieve it at a reduced throughput. Flexible design recognizes that operation can be optimized after production starts and that retrofit modification is possible where initial production targets cannot be met. It recognizes that it may be uneconomic to design to ensure that, even under the worst combination of uncertain outcomes, target production rate is met. Market size is often one of the most uncertain parameters on which a

chemical engineering design is based. As the market builds up, there may be the opportunity to debottleneck a plant that initially underperforms. Flexible design explores the trade-off between applying excessive design margins and running a risk that a target production rate cannot be met.

Flexible design requires optimization under uncertainty. In such optimization, it is recommended that hard constraints on production rates are avoided. Instead, it should be recognized that there is flexibility in operation policy and realistic penalties should be applied for shortfalls that might occur under unfavorable outcomes to the uncertain parameters. This approach also avoids sharp discontinuities in the cost function that makes optimization more difficult.

There is active research in automated methods for optimal design under uncertainty. The technology is, however, not yet available for use outside the relevant research schools.

5. Conclusions

The use of computer aids does not reduce the responsibility of engineers. Indeed, with fewer simplifying assumptions, there is a greater requirement that the engineer has a fundamental understanding of the technology. Furthermore, more detailed models demand more extensive data that also needs to be obtained and critically assessed. The speed and accuracy of the computations makes it practicable to design better processes. Processes can be more thoroughly evaluated and their risks and opportunities more thoroughly assessed. Recent developments promise better, cleaner, safer processes.

BIBLIOGRAPHY

"Computer-Aided Engineering" in *ECT* 4th ed., Vol. 7, pp. 128–163, by M. T. Tayyabkhan, Tayyabkhan Consultants, Inc. and H. Britt, Aspen Technology, Inc., "Computer-Aid Engineering" in *ECT* (online), posting date: December 4, 2000, by M. T. Tayyabkhan, Tayyabkhan Consultants, Inc. and H. Britt, Aspen Technology, Inc.

CITED PUBLICATIONS

1. American Institute of Chemical Engineers, On-line Software Directory, http://www.cepmagazine.org/features/software/.
2. C. Mazza, J. Fairclough, B. Melton, D. DePablo, A. Scheffer, R. Stevens, M. Jones, and G. Alvin, *Software Engineering Guides*, Pearson Education, Harlow, UK, 1995.
3. G. Booch, I. Jacobson, and J. Rumbaugh, *Unified Modeling Language User Guide*, Addison Wesley Longman Publishing Co., Reading, Mass, 1998.
4. *NIST, Reference Information for the Software Verification and Validation Process*, NIST Special Publication 500–234, U. S. Dept of Commerce, Washington, D.C. Mar 1996.
5. W. E. Perry, *Effective Methods for Software Testing*, Wiley, New York, 1995.
6. M. Tanzio, Validate your Engineering Software, *Chem Eng Prog* **97** (7), 64, 2001.
7. G. Birtwhistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard, *Simula – Begin*, Chatwell-Bratt, Bromley, UK, 1979.

8. B. Stroustrup, *The C++ Programming Language*, 3rd edn, Addison-Wesley, Reading, Mass., 1997.
9. T. Stanton and J. Pannell, *Philos. Trans. R. Soc.* **214**, 199, (1914).
10. L. F. Moody, *Trans Am Soc Mech Eng.* **66**, 671, (1944).
11. F. W. Dittus and L. M. K. Boelter, *University of Berkeley, Pubs Eng.* **2**, 443, (1930). Reprinted in *Int. Comm. Heat Mass Transfer* **12**, 3, (1985).
12. W. H. McAdams, *Heat Transmission*, 2nd ed., McGraw Hill Book Co., Inc., New York, 1942.
13. J. F. Epperson, *An Introduction to Numerical Methods and Analysis*, Wiley, New York, 2001.
14. HSL (formerly the Harwell Subroutine Library), *AEA Technology Engineering Software*, Harwell, UK.
15. NAG Library, The Numerical Algorithms Group Ltd, Oxford, UK.
16. W. H. Press, S. A. Teukolsky, and B. P. Flannery, *Numerical Recipes in C++*, Cambridge University Press, Cambridge, UK, 2002. (Also available in Fortran 77 and Fortran 90).
17. R. H. Perry and D. W. Green (eds), *Perry's Chemical Engineers' Handbook*, 7th Edition, McGraw-Hill, New York, 1997.
18. R. Best, G. Goltz, J. Hulbert, A. Lodge, F. A. Perris, and M. Woodman, *The Use of Computers by Chemical Engineers.* IChemE (CAPE Subject Group), Rugby, UK, 1999.
19. P. M. Harper and R. Gani, *Comp Chem Eng.* **24**, 677, (2000).
20. W. R. Johns, *Chem. Eng. Prog* **97** (4), 59, April 2001.
21. *User Guide on Process Integration for the Efficient Use of Energy*, IChemE, Rugby, UK (1992).

References to Programming Languages and Modeling Systems

LISP: P. Graham, *The ANSI Common Lisp Book*, Prentice Hall, Inc., Englewood Cliffs. N.J., 1995.
Prolog: I. Bratko, *Prolog Programming for Artificial Intelligence*, Longman, Reading, Mass, 2000.
Mathematica: S. Wolfram, *The Mathematica Book*, Cambridge University Press, Cambridge, UK, 1999.
MathCAD: R. W. Larsen, *Introduction to MathCAD 2000*, Prentice-Hall, Inc., Englewood Cliffs, N. J., 2001.
GAMS: *General Algebraic Modeling System*, GAMS Development Corporation, Washington, D.C.
SpeedUp: Aspen Technology Inc, Cambridge, Mass.
gProms: Process Systems Enterprises Ltd, London, UK.
Excel: Microsoft Corporation, Redmond, Washington.
Lotus 1-2-3: Lotus Corporation, Cambridge Mass.

W. R. JOHNS
Chemcept Limited