INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa · University of Ottawa

ODEASY: A QUERY TOOL FOR ORDINARY DIFFERENTIAL EQUATIONS

By

© **Kim M. Tam**

A MASTER THESIS

Submitted to
The University of Ottawa
in partial fulfillment of the requirements
for the degree of

MASTER OF COMPUTER SCIENCE

Department of Computer Science

1997

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-36744-4

Canada

ABSTRACT

ODEASY: A QUERY TOOL FOR ORDINARY DIFFERENTIAL EQUATIONS

By

Kim M. Tam

Ordinary differential equations (ODEs) are widely used in the modeling of physical systems. However, the numerical methods employed to approximate solutions to systems of ODEs can be constrained by truncation error, round-off error, and problems of stability and stiffness. These issues can be circumvented for equations that have known analytical solutions if the solutions are readily accessible in a database. The challenge then is to find a method of identifying an ODE given that it is unlikely that a query of the same ODE will have the equation elements presented in the same sequence and format by all users. Several possibilities of sorting equation elements are explored and an algorithm is developed and implemented which sorts the elements in order to convert the equation into a canonical form. The sorting algorithm explicitly defines this canonical form. By converting semantically identical equations into the same canonical form irrespective of the sequence of term entry, this algorithm is the basis of the development of ODEASY, the 'Ordinary Differential Equation Answer SYstem', which is a database system that accepts an ODE in any format for query. The input ODE is sorted to serve as its own index into the database of analytic solutions.

ODEASY offers a sophisticated user interface for accessing the analytic solutions of ODEs of the first, second, and third order and of both the linear and nonlinear variety. ODEASY is a standalone PC system that also offers potential as a supporting tool to applications that provide a numerical approach to solving continuous system simulation.

ACKNOWLEDGEMENT

The author wishes to express her most sincere appreciation and gratitude to her supervisor, Dr. Tuncer Oren, for his advice and encouragement in this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Ordinary differential equations (ODEs) have widespread applications in the mathematical modeling of physical problems and are the fundamental mathematical components in continuous system simulation. Except in the simplest of cases that can be solved algebraically, the majority of modeling is performed with numerical methods to approximate ODE solutions. With advances in computing, numerical methods offering greater accuracy have been developed and the possibilities for multiprocessing have spawned active investigations into parallel ODE solvers. However, all numerical methods are approximations that must deal with truncation errors and are challenged by stiffness problems and in the case of multiprocessor algorithms, by some of the limitations of parallelism. These issues can be avoided or at least alleviated in cases where the problem contains an ODE with an analytical solution and that solution is known and readily accessible. Currently, there are exact solutions for over 5000 ODEs of various types including linear/nonlinear and first, second, third, and higher orders compiled in sources such as [Polyanin and Zaitsev, 1995] and [Fogiel, 1986]. However, due to the volume of equations, the targeting of any particular equation in these and other sources is inefficient. This look-up process could be automated if the difficulty of matching equivalent ODEs that have elements in different sequences can be overcome. This ability is critical for the success of any application that offers a database of solutions since users cannot be limited by the exact format and sequence of the ODEs in the database. The motivation of this thesis is to first develop a method to sort the elements of an ODE in order to convert the equation into a canonical form. Hence, the sorted result is the same regardless of the permutation in the element sequence of the original ODE. Secondly, such a method can then be incorporated into the development of a database system capable of accepting an ODE in any sequence and format.

Several possibilities for equation element sorting are explored. An algorithm is developed which will consistently sort an ODE into the same structure irrespective of the ODE's input sequence. This algorithm is implemented as part of a database and query tool: ODEASY – "ODE Answer SYstem". ODEASY is a user-friendly system that

imposes no restrictions on the format and sequencing of elements in an ODE that is entered to query for a solution. Any alphabetic character may represent the variables and constants. This variability in the input is bridged into a database that accepts a uniform set of variables and constants and recognizes just one sequence in the way the elements of an equation may be ordered. A parser and the sorting algorithm ensure that the input ODE is changed into the canonical form expected by the database, and hence, the ODE serves as its own index into the database. With an effective means of accessing a particular solution of interest in a database of known analytic solutions, ODEASY can be a practical alternative to traditional ODE solving, especially for difficult to solve nonlinear higher order equations.

While not intended as a complete math package, ODEASY, which accepts both manual and point-and-click equation input, can be compared for entry and solution of ODEs with other ODE solvers. Furthermore, ODEASY has the potential of supporting other systems that provide a numerical approach to continuous system simulation. In this manner, ODEASY may serve to assist other applications to progress into a complete problem solving environment (PSE) for comprehensively dealing with continuous system simulation.

Chapter 2 provides an overview of modeling with ODEs. Chapters 3 and 4 assess current methods and tools available for solving ODEs and establish the rationale for ODEASY. In chapter 3, an overview of methods of solution and their drawbacks are presented. In chapter 4, several knowledge bases and software packages available for solving differential equations are surveyed. Chapter 5 discusses methods for matching an ODE. Chapter 6 presents the details of ODEASY: the sorting strategy, architecture, and interface. The conclusion is provided in Chapter 7. The source code can be found in the Appendix V.

2. MODELING WITH ODES

Many physical systems and relationships may be modeled, i.e., formulated mathematically, via ordinary differential equations which are applicable whenever a physical law involves the rate of change of a function, such as velocity, acceleration, force, etc., and hence, are of particular relevance in physics and engineering. For example, the derivative

$$y' = \frac{dy}{dt} \qquad \text{Eqn 2.1}$$

may reflect the population growth rate of certain organisms over time (where y =population number, t =time). If the growth rate is equal to the current population, $y(t)$, then the differential equation for this model is $y' = y$. This has the general solution $y = ce^t$ (where c is a constant).

When the arbitrary constant, c, is assigned a specific value, the function becomes a particular solution satisfying a given condition resulting from the physical system. With linear equations, there are as many arbitrary constants as the order of the equation and hence, as many initial conditions are required to generate a particular solution. This is the framework of initial value problems where for an equation of nth order, n initial conditions, i.e.

$$y(t_0) = K_0, y'(t_0) = K_1, \ldots y^{(n-1)}(t_0) = K_{(n-1)} \qquad \text{Eqn 2.2}$$

(where K_i are given values), at point, t_0, are provided to customize general solutions into more applicable particular solutions. As an example, assume that the rate of decomposition of a radioactive substance is proportional to the amount that is present. If we start with 5 grams of the substance at time $t = 0$, we can develop a particular solution to find the amount left at a later time. Denoting the amount of substance available at time t by $y(t)$, the rate of decay is dy/dt and the physical process can be represented by an ODE of the first order:

$$y' = \frac{dy}{dt} = ky \qquad \text{Eqn 2.3}$$

where k is a definite physical constant for the decay of the substance and is negative. The general solution valid for all t is $y(t) = ce^{kt}$. The unique behaviour of this physical process is determined by the value for the arbitrary constant, c, that is defined by the initial condition, $y(0) = 5$ grams. Thus, the particular solution is $y(t) = 5e^{kt}$ which characterizes the amount of substance still present at any time $t \geq 0$. Other well known ODE applications include Kirkhoff's modeling of electric currents [Larson and Hostetler, 1986] and Newton's laws of classical mechanics [Brauer and Nohel, 1986].

2.1 Continuous System Simulation

Simulation refers to the process of experimentation with mathematical models of real systems. According to [Shannon, 1975], simulation is "the process of designing a computerized model of a system (or process) and conducting experiments with this model for the purpose either of understanding the behaviour of the system or of evaluating various strategies for the operation of the system". The motivation for performing simulation arise from the difficulty associated with directly experimenting on the real system due to factors such as cost, time, danger, and unavailability of the system.

Continuous system simulation (CSS) deals with dynamic systems that continuously undergo state changes over the time domain. Since this behaviour pattern is governed by rates of change, it can be modeled by ordinary differential equations (ODEs). The modeling is essentially explanatory in nature reflecting causal relationships and can be characterized by an underlying system model of the form:

$$\frac{dZ}{dt} = \dot{Z}(t) = F(t, U(t), Z(t)) \qquad \text{Eqn 2.4}$$

where the $Z(t)$ is an n-dimensional state vector and $U(t)$ represents an m-dimensional input function. Since simulation study the rate of change with respect to time, $U(t)$ is always a known function of time and can be incorporated within the explicit time

dependence that is shown for the derivative function, F. Thus, the CSS task corresponds to solving the model

$$Z(t) = F(t, Z(t)) \qquad \text{Eqn 2.5}$$

over some predefined time interval $[t_I, t_N]$ beginning from a given initial state $z(t_I) = zI$. Except for a very limited class of such equations, the primary solution method applied is numerical integration. ODEASY may assist in the CSS task if one or more of the member ODEs in the model can be replaced by a readily available analytic solution.

Examples of CSS studies include investigations of nuclear reactor control, aerospace vehicle dynamics, and chemical process kinetics. Other applications are described in [Smith, 1987] and [Cellier, 1991].

3. SOLVING ORDINARY DIFFERENTIAL EQUATIONS

This section presents a brief overview of both algebraic and numerical methods of ODE solving along with some of the difficulties associated with their application. These problems, which could be avoided with ODEASY, serve to justify the necessity for an ODE search tool to access a database of analytic solutions.

Simple physical situations including radioactive decay, heat conduction, and hydro dynamics among innumerable others may be modeled using relatively simple ODEs that can be solved algebraically given the appropriate technique. In most simulation studies, however, numerical methods are applied to approximate a solution due to the number or complexity of equations involved.

3.1 Algebraic Methods of Solution

A plethora of techniques exist for the algebraic solution of ODEs. The proper method to apply depends on the structure of the equation which in general form is

$$a_0(t)\frac{d^n y}{dt^n} + a_1(t)\frac{d^{n-1} y}{dt^{n-1}} + \ldots + a_n(t)\,y = r(t) \qquad \text{Eqn 3.1}$$

The order n, linearity (refer to §3.1.3), and whether $r(t)$ appears independently of y indicating a nonhomogeneous equation, or $r(t) = 0$ for a homogeneous equation determines one to several algebraic techniques that are applicable.

3.1.1 First Order Linear ODE

If the equation consists of elementary integrals, various algebraic solution methods such as the separation of variables and the use of integrating factors to obtain an exact equation (which becomes solvable) may be employed. This involves the recognition that the ODE is of a certain pattern, i.e., $g(y)y' = f(t)$ for a separable equation and $M(t,y)dt + N(t,y)dy = 0$ where the LHS is the total differential of some function $u(t,y)$ for an exact equation. The technique for an exact equation can be quite involved and increases in complexity if an integrating factor is also required. Indeed, although simple

equations may be changed into a separable or exact form by inspection, the manipulation of an equation into a certain form in order to apply the correct technique often requires painstaking steps along with some ingenuity as attested by the numerous examples in [Fogiel, 1986].

3.1.2 Higher Order Linear ODE

Linear equations of higher orders of the homogeneous variety might be treated with:

- the characteristic or auxiliary equation or
- the Euler-Cauchy equation if the homogeneous ODE is accompanied with variable coefficients of the same degree as the order of the differentials.

Both require finding the roots of the characteristic equation with the solution being function of the type of roots found (i.e., distinct real, complex conjugate, real double root). The greater the order, the more problematic it becomes in finding all roots since an equation of order n with distinct roots will have n such roots. Generally, a numerical method, such as Newton's method or fixed-point iteration, is required to deal with the characteristic equations of larger order ODEs.

Higher order ODEs or ODEs with variable coefficients that cannot be handled by the Euler-Cauchy equation, might be dealt with by way of power series. However, care must be taken to verify the interval of convergence. The simplification of solutions often requires the identification of recurrence relationships or of a known series that can be summed to elementary functions (e.g., a Taylor or MacLaurin series).

Another possibility for a large order ODE is to reduce it into a first order equation by an appropriate choice of additional variables. For example, the damped harmonic oscillator equation $m(d^2y/dt^2)+n(dy/dt)+ky=0$ can be rewritten in terms of y and velocity $v=dy/dt$ with the two equations $v'=-v(n/m)-y(k/m)$ and $y'=v$ which are in state-variable form (with y and v the state variables). In a similar way, any n^{th} order differential equation can be reduced to n first order equations creating a system of

equations to solve. This reduction in complexity may result in additional problem-solving steps.

The algebraic solution of the nonhomogeneous type of ODEs requires both the general solution to the corresponding homogeneous equation as well as a solution containing no arbitrary constants to address $r(t)$ in the general form (refer to Eqn 3.1). Techniques for obtaining this non-homogeneous specific solution include the limited method of undetermined coefficients and the more general method of variation of parameters which requires as many integration steps as there are constants in the homogeneous equation's general solution. Some of the difficulties associated with this solution process may be bypassed through the use of Laplace transforms which reduces the problem of solving a differential equation into an algebraic problem.

3.1.3 NonLinear ODE

The category of nonlinear differential equations is characterized by equations that

(1) have the dependent variable, y, or any of its derivatives occurring to the second degree or higher, or

(2) have products of y and/or any of its derivatives, or

(3) have transcendental functions of y and/or its derivatives.

Non-linear ODEs tend to be complex and require greater analysis in their solution. Outside of perturbation series, there is a dearth of systematic solving methods for nonlinear equations. It may be possible to approximate solutions via solvable linear equations. Algebraic techniques can be used to either convert them into one or more linear equations where possible or into a form with integrable combinations. For example, the equation $ty'' - (y')^3 - y' = 0$ can be substituted with $p = dy/dt$ to yield $t\,dp/dt - p^3 - p = 0$ which is a nonlinear separable equation whose solution can have p re-expressed as a function of t and integrated again to obtain a solution for y. This will arrive at the solution of $t^2 + (y - c_2)^2 = {c_1}^2$ [Fogiel, 1986, pp.1208]. Likewise, a

substitution of $p = dy/dt$ can both reduce the order and linearize $2yy'' = (y')^2$ into the separable homogeneous equation $dp/dy - (1/2y)p = 0$ that is linear in p. This can then be solved to the general solution $y = (c_1 t + c_2)^2$ [Fogiel, 1986, pp.1212]. However, such manipulations rely on trial-and-error. Consequently, numerical methods are often called upon to approximate the solution of nonlinear equations.

3.1.4 Drawbacks of Algebraic Methods

The preceding discussion indicates that one would need to be quite proficient in solution methods to correctly identify the best technique to use based upon the structure of the ODE and that even with a properly selected method, the solution steps can be tedious and laborious. These points are particularly valid for users of ODEs who are not mathematicians but scientists and engineers who require solutions but are not adept at solution methods. In comparison, the ODEASY database is a simple to use alternative requiring no special skills in solution techniques.

3.2 Numerical Methods of Solution

Numerical methods for approximating the solutions of ODEs are available for initial value problems of the first and second order and include techniques such as the improved Euler-Cauchy method, the fourth order Runge-Kutta method, the Adams-Moulton and Adams-Bashford multi-step method, and the Runge-Kutta-Nystrom method for second order equations.

In general, these step-by-step methods begin with the given initial condition, $y_0 = y(t_0)$ and proceed stepwise by step h. In the first step, an approximate value y_1 of the solution y at $t = t_1 = t_0 + h$ is determined; in the second step, the approximate value y_2 at $t = t_2 = t_0 + 2h$ is determined, and so on. In each step, one or more predictions are made for the value of the function based upon the step size and the previous value of the function. It is then refined or "corrected" to estimate the current value of the function. For example, the popular Runge-Kutta method incorporates four predicted or auxiliary

quantities into the new value y_{n+1} as illustrated in the following Runge-Kutta algorithm described in [Burrage, 1995]. It computes the solution of the initial value problem $y' = f(t, y)$ with $y(t_0) = y_0$ at equidistant points $t_1 = t_0 + h$, $t_2 = t_0 + 2h$, ..., $t_N = t_0 + Nh$ where f is assumed to have a unique solution on the interval $[t_0, t_N]$:

Input: Initial values t_0, y_0, step size h h, number of f steps N

Output:Approximation y_{n+1} to the solution $y(t_{n+1})$ at $t_{n+1} = t_0 + (n+1)h$,

where $n = 0,1,..., N-1$.

For $n = 0$ to $N-1$ do

$$k_1 = hf(t_n, y_n)$$

$$k_2 = hf(t_n + 0.5h, y_n + 0.5k_1)$$

$$k_3 = hf(t_n + 0.5h, y_n + 0.5k_2)$$

$$k_4 = hf(t_n + h, t_n + k_3)$$

$$t_{n+1} = t_n + h$$

$$y_{n+1} = y_n + (1/6)(k_1 + k_2 + k_3 + k_4)$$

Output t_{n+1}, y_{n+1}

endFor

These techniques replace the complicated computation of derivatives of $f(t, y)$, with the computation of f for one or several suitably chosen predicted values of (t, y), i.e., values of f at certain points, which, while computationally intensive, is straightforward and iterative in nature lending the methods to be ideal computer algorithms.

A basis for comparing and evaluating numerical methods is via the Taylor series which is applicable if $y(t)$ is a smooth function so that it is possible to describe $y(t+h)$ with

$$y(t+h) = y(t) + hy'(t) + (1/2!)h2y''(t) + (1/3!)h2y'''(t) + ... + (1/n!)hny^n(t) + ...$$

If h is small and $y(t)$ and its derivatives are all known, then the truncated series involving the $(n+1)$ terms can be used to estimate $y(t+h)$. When terms up to and

including $(1/n!)hny^n(t)$ are present in the series, the representation is said to be a Taylor series of order n. Note that the Taylor series itself is of limited practical value for general simulation since it requires obtaining the derivatives associated with the terms in $h_1, h_2, h_3, \ldots$, etc.

Variations of numerical techniques include variable step methods and multistep methods. The latter refines the corrected result by applying the correction process repeatedly via a k-step method that uses values of y and its derivatives at k prior time instants. Because of this, multistep routines require some other method such as the Runge-Kutta applied at the start to get the integration process underway. Widely used multistep methods for real-time continuous system simulation include the explicit Adams-Bashford method and the implicit Adams-Moulton method [Lambert, 1991]. Variable step algorithms are useful when different step sizes are appropriate in different operating ranges of the system as may be the case with nonlinear systems. Variable step methods such as the Fehlberg algorithm [Lambert, 1991], an order four method involving an algorithm of the Runge-Kutta type, incorporates the variable-step feature by determining the difference between the values of $y(t_k + h)$ calculated by fourth and fifth order procedures. This difference is taken as an estimate of the truncation error (see §3.2.1), and if this exceeds a given tolerance the step size is halved but if the difference is much less than the tolerance, the step size is doubled.

3.2.1 Drawbacks of Numerical Methods

Besides being limited to equations of the first and second order, numerical methods must have an appropriate step size in order to minimize truncation errors, round-off errors, and to ensure stability. Since the step size, h, impacts upon the size of the error in the approximation, the selection of h must consider the dynamic characteristics of the system modeled. Even so, systems with the stiffness property can not be easily solved by adjustments to h. Each of these problematic areas that can result from applying numerical methods to solve ODEs are addressed in turn.

Truncation error in the integration process involves an approximation that can be regarded as equivalent to truncation of a Taylor series after a finite number of terms. In general, the truncation error at each step is of the order of the first term dropped. For example, in the fourth-order Runge-Kutta method, the truncation error is of the order of h^5. While reduction of the integration interval would improve accuracy, a smaller step size increases another type of error. This error is due to the rounding off that occurs within the computer which represents quantities using a finite number of binary digits. Different compilers handle the rounding of numbers in different ways. Thus, while the truncation aspect of numerical errors are minimized with smaller step sizes, the round-off error becomes more significant with the decreasing integration step as the number of iterations required for a simulation run becomes larger. In general, truncation errors are dominant when large values of integration step size are used, while round-off errors are more important for small integration steps. This relationship is illustrated in Figure 3.1.

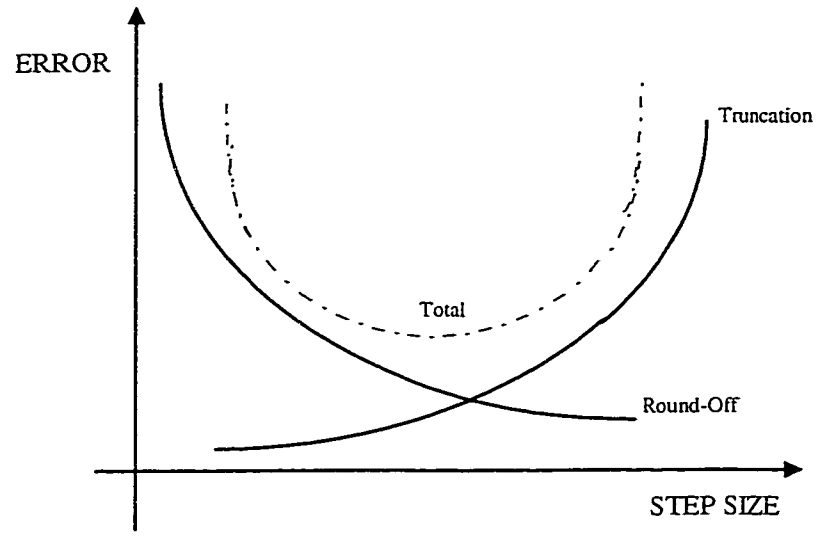


Figure 3.1 A sketch of the typical variation of solution error with step size.

The problem of numerical instability can occur quite independently from questions of inaccuracy due to truncation or round-off. Instability depends on the form of algorithm chosen and the step size adopted. Choosing a wrong step length can result in a divergent graph; as an example of this, [Murray-Smith, 1995] illustrates a system defined by the transfer function $Y(s)/U(s) = 1/(1+0.5s)$ approximated by the basic one-step Euler method with $y(t_k + h) = y(t_0) + h[2 - 2y(t_k)]$ and initial condition $y(t_0) = 0$ for step sizes $h = 0.8$ and $h = 1.01$ as shown in Figure 3.2 and Figure 3.3, respectively. In Figure 3.2, the simulated result stabilizes towards the real system after 8 seconds while in Figure 3.3 with the larger integration interval, the simulation shows an increasing oscillation that is clearly not characteristic of the real system.

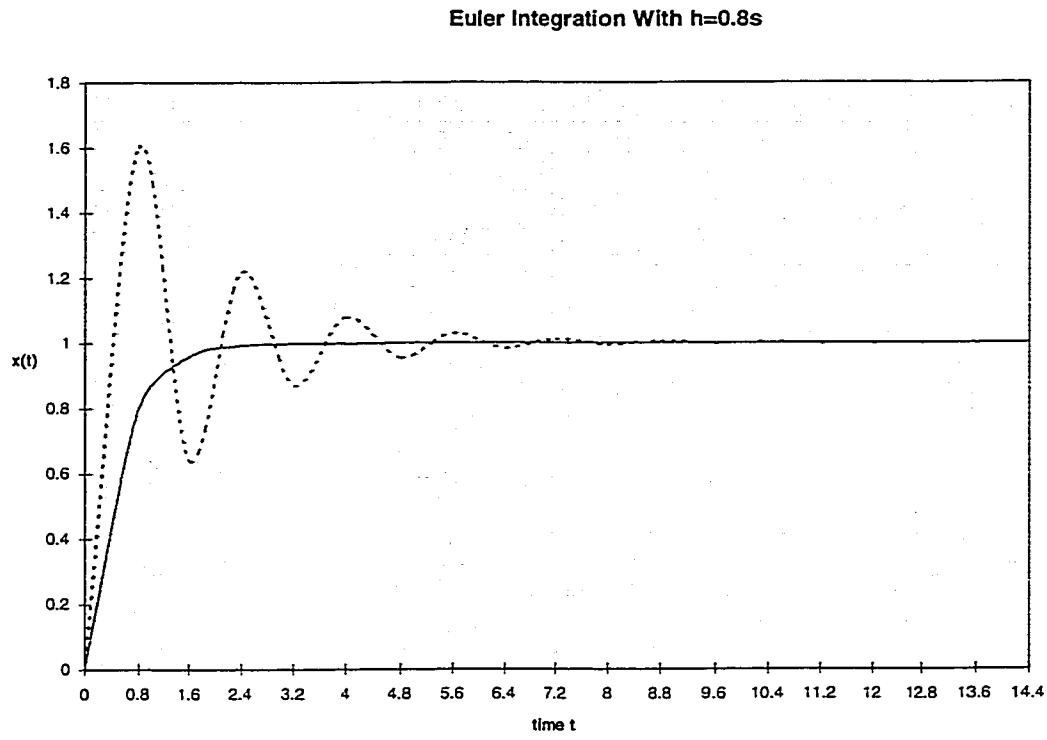


Figure 3.2 Step Size of 0.8 (dashed and continuous line indicate simulated response and exact solution, respectively).

As depicted by Figure 3.2 and Figure 3.3, the determination of appropriate integration step sizes can be problematic due to the limits that may exist in terms of the intervals for which a given integration algorithm will be stable. Of particular difficulty are nonlinear

problems that call for changes in the step size as the simulation progresses through the operating range necessitating the use of variable step methods as discussed in §3.2. However, the more complex integration methods also have a greater tendency towards numerical instability for chosen integration step sizes [Murray-Smith, 1995]. Furthermore, the use of complex integration algorithms incurs a greater computation time overhead that may present an excessive penalty for some time critical real-time simulations.

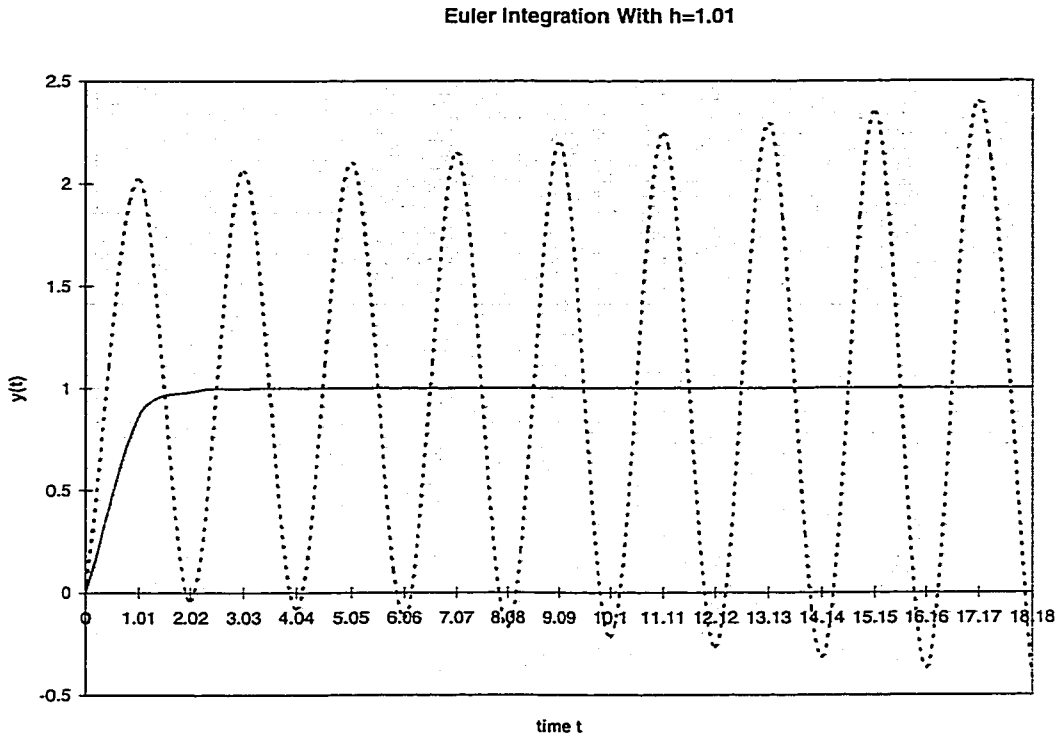


Figure 3.3 Step Size of 1.01 (dashed and continuous line indicate simulated response and exact solution, respectively).

Another area of challenge for numerical methods are stiff systems which involve differential equations with both very fast components and slow components. In the case of linear models this corresponds to a situation where the characteristic equation has roots that cover a very wide range of values. For example, the system $d^2y/dt^2 + (101)\,dy/dt + (100)\,y = 0$ has eigenvalues -100 and -1 so that the solution contains the "fast" component $a_1 e^{-100t}$ and the "slow" component $a_2 e^{-t}$. Since the

response contains one component that is very much faster than the other, the integration step size must be small enough to suit the most rapidly changing component. However, this small step size may be unacceptable in terms of the computing time needed to obtain the complete solution which must undergo the full-time history of the slow component of the response. Furthermore, the excessive number of steps required by classical numerical methods to deal with stiff systems may cause large enough round-off errors as to put the validity of the solution into question although the availability of high precision arithmetic has helped to reduce the occurrence of this. The complications of stiffness has spurred the development of special integration techniques such as [Cash, 1979] and an algorithm from [Gear, 1984] has become widely used as an option for stiffness in many continuous system simulation languages.

ODEASY can overcome the problems associated with numerical methods by replacing an equation with its known analytical solution so that issues such as truncation error, round-off error, stability and stiffness are no longer relevant. Furthermore, ODEASY can be consulted in the absence of initial conditions, a deficiency that numerical methods cannot overcome since they require initial values as priming data at startup.

While it is straightforward to replace a single equation by its corresponding solution, many applications are defined by multiple equations. The role that ODEASY may play in systems of equations is discussed next.

3.3 Systems of Equations

Larger problems including networks of many circuits, mechanical systems of several masses and springs and many others are characterized by multiple variables leading to systems of differential equations. If the equations are uncoupled, ODEASY can be consulted for each individual equation for its analytical solution. However, if there is coupling among the equations, then the applicability of ODEASY depends upon the degree of coupling. When it is applicable, ODEASY may serve to avoid stiffness in stiff systems and to expedite parallel numerical methods by removing a submodel from consideration.

3.3.1 Avoidance of Stiffness

Stiffness, as discussed in §3.2.1, may also affect several uncoupled equations of a system as indicated by the following example taken from [Gear, 1971] and given by [Ghasem-Aghaee and Oren, 1995]: the initial value problem $y' = -y, z' = -1000z$ with initial conditions $y(0) = 1$ and $z(0) = 1$ has eigenvalues -1 and -1000 which gives a ratio of stiffness of $s = 1000$. If this system is solved with traditional numerical algorithms such as the Euler method or the fourth order Runge-Kutta method, the stability requirements will force 1000h to be bounded. The level of stiffness will depend on the values of the parameters. With ODEASY, a consultation will provide the analytical solution $y = e^{-t}$ and $z = e^{-1000t}$ thereby avoiding the stiffness issue altogether.

3.3.2 Reduction of Parallelism

An active area of research for simulation studies of large scale continuous systems is the application of parallel solution methods for solving ODEs on multi-processor systems. This includes parallel algorithms such as block predictor-corrector (PBC) and parallel Runge-Kutta models [Ghassem-Aghaee and Birta, 1994; Birta and Cheng, 1995; Abou-Rabia et al., 1989; Birta and Abou-Rabia, 1987] and problem partitioning methods [O'Grady and Wong, 1987; Beuhrer et al., 1982]. These works are concerned with generating solutions to a set of ODEs having the form stated in §2.1, that is:

$$\frac{dy}{dt} = y'(t) = f(t, y(t)) \qquad \text{Eqn 3.2}$$

where, in general, y and f are vectors; t_0, t_f and $y(t_0)$ are given; and the objective is to generate the solution to Eqn 3.2 over the interval $[t_0, t_f]$.

A common underlying theme of these parallel methods is the need to efficiently utilize available processors by evenly distributing the workload so that all processors involved in the solution procedure are uniformly active. The problem partitioning method applies a classical integration procedure across the solution axis and achieves parallelism by distributing the computational tasks over the available processors. The derivative

vector f may be segmented at the individual equation level or at a finer level of granularity within the individual components of the f vector. Segmenting at the finer level of granularity is a generalized approach that permits greater flexibility in equitably distributing the workload among processors but it requires a decomposition procedure. Besides the need to ensure a proper workload balance, the partition method also requires a high bandwidth data exchange mechanism to support substantial data exchange among processors, especially for finer levels of granularity. The EMPRESS project at ETH, Zurich [Buehrer et al., 1982] which provides a compile-time decomposition feature is an example of the generalized partition method.

The other approach for the parallel solution of ODEs is via parallel algorithms. Also referred to as block methods, these algorithms do not partition equations as in the partition methods. Instead, a parallel algorithm generates a group, or block, of $k \geq 2$ equally spaced solution values with each application of its underlying formulas. To do this, the solution axis is divided into a sequence of adjacent and possibly non-overlapping intervals or blocks with each block containing a fixed number of internal points and each point allocated to a processor. Hence, there is a direct relationship between the number of points, M, and the number of processors, N, (e.g. M=N in PBC and M=N/2 in PPBC [Ghasem-Aghaee et al., 1994]). All processors carry out a sequence of computationally identical tasks and thus, avoids the workload balancing problem. Based upon the general configuration for parallel block methods shown in Figure 3.4 from [Birta and Cheng, 1995], each computational cycle advances the solution by k new points along the solution axis. It does so in the following manner: the multiple processors generate solution data at the points in block B_{n+1} using data from block B_n and possibly from blocks to the left of B_n. This is carried out in two steps where the first step involves using the values in block B_n to first generate a set of tentative solution values within block B_{n+1}. The second step constructs from the tentative values a set of final values in block B_{n+1}, possibly via an iterative procedure. As with conventional numerical methods, parallel algorithms also require initial values as priming data to begin the process.

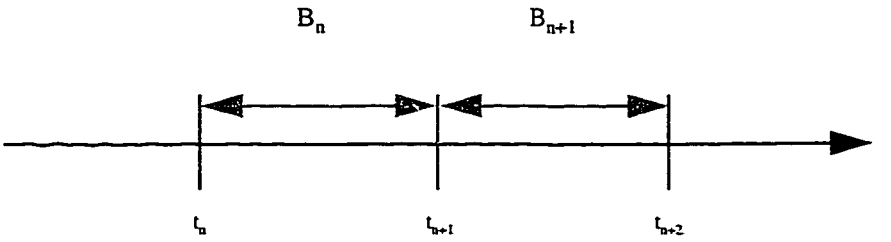


Figure 3.4 General Configuration for Parallel Block Methods.

Performance studies show that derivative function evaluations (as opposed to the evaluation of linear expressions that provide solution values) dominate the processing time when it comes to large problem size for both the BPC and the parallel block-predictor-corrector (PBPC) approaches [Ghasem-Aghaee et al., 1994]. The complexity arising from the derivative function evaluations was found to be the most significant aspect of the problem complexity. Parallel implicit Runge-Kutta methods offer improved stability properties but has the penalty of relatively higher computational overhead. Tentative results in [Birta and Cheng, 1995] indicate that the BPC method provides superior performance over that of the block parallel iterated Runge-Kutta (BPIRK) method by requiring fewer derivative function evaluations per processor. [Abou-Rabia et al., 1989] shows that performance of the BPC and PPC (parallel predictor-corrector) methods degrades to an unacceptable level if a low order of integration is used (3 in this study). Lastly, it does not appear that incorporating additional processors yields a consistent decrease in total computing time and that in some cases, the solution time might actually increase [Ghasem-Aghaee et al., 1994; Abou-Rabia et al., 1989].

ODEASY may expedite parallel processing by removing one or more of the equations in the problem set through replacement with known analytic solutions. If a coarse level of granularity is used in the problem partitioning approach, this will simplify the workload distribution and possibly reduce the number of processors required. With parallel algorithms, the removal of an equation will reduce the complexity of the derivative vector, f, leading to a reduction in the number of derivative function evaluations so that solution time is expedited. The degree to which this is possible depends on how closely

the equations of Eqn 3.2 are coupled. An equation that does not reference the state variables of any other may be checked immediately for a known analytical solution. A coupled equation would require an elimination method to express it solely in terms of its state variable although this may not be a simple task in highly coupled systems. Here, one needs to determine the sequence and the number of equations requiring parallel processing and the search into ODEASY may only be an intermediate step if no solution is found or if there are additional equations that require the parallel algorithm to complete the solution.

3.4 Solving with ODEASY

The case has been made in the preceding discussions that a database of analytic solutions such as ODEASY can offer positive benefits as an alternative or aid to the other methods of solving ODEs. In summary, these include:

- Providing a solution without requiring special skills from the user.
- Eliminating laborious efforts that are associated with obtaining a solution via algebraic methods.
- Offering solutions to higher order ODEs that tend to be problematic for other methods to derive.
- Avoiding stability issues, and truncation and round-off errors in numerical methods.
- Avoiding stiffness.
- Reducing model size to expedite processing with parallel methods.

4. SYSTEMS FOR THE SOLUTION OF ODES

Current tools available for the solution of ODEs include continuous simulation languages, knowledge-based systems, and mathematical software. Some of these are at the experimental stage of development while others such as ACSL, Mathematica, and Maple are commercially released products. This chapter presents an overview of some existing ODE solution systems and gauges them in terms of their adequacy as problem solving environments (PSEs). The role that ODEASY may play in a PSE is discussed in §6.4. Here, the concept of the problem solving environment is introduced followed by an examination of several ODE solution systems.

4.1 The Problem Solving Environment (PSE) Perspective

A PSE is a computer system that provides all the computational facilities necessary to deal with a specific class of problems [Gallopoulas et al., 1994]. These facilities would include tools and methodologies to help define the problem, to solve the problem and to analyze the results, all in a language that is natural to the problem domain. The PSE of the future is a multi-paradigm system supported by heterogeneous algorithmic infrastructures and which interacts with multiple devices that each process or contribute its own type of data. PSEs are characterized by their scope, power, and reliability [Gallopoulas et al., 1992]. Scope refers to the extent of the problem set that the PSE addresses while power refers to its ability to actually solve the problems posed. The more complex the problem class, the greater the likelihood that the PSE will fail on some problems. The reliability of a PSE is a measure of how often it produces correct answers.

A PSE should perform as an all-encompassing multi-media tool that can interact with users on their own terms. However, [Cornea-Hasegan et al., 1994] concede that current PSEs are not fully developed but do share some of the characteristics of an ideal PSE. Many have had major impacts in their fields by enabling users to make computations easily that are otherwise either very tedious or beyond the users' technical capability. Examples of popular PSEs include matrix laboratories (e.g. MATLAB), statistical packages (e.g. SPSS), and partial differential equation (PDE) based systems. A list of the

latter type of PSEs and whether they fulfill certain PSE functionalities is reproduced from [Gallopoulas et al., 1994] in Table 4.1. While the functionality features relate to PDEs here, they are generalizable to ODE systems.

PSE Functionality	RPI	ELLPACK	//ELLPACK	DEQSOL	VECFEM	ALPAL	PDE2D
Interactive I/O	x		x	x	x	X	
Graphical I/O	x	x	x	x	x		
Multimedia I/O							
Interactive Geometry			x	x	x		
Auto. Discretization	x	x	x	x	x		x
PDE Language	x	x	x	x	x	x	x
PDE Model Gen.							
PDE Solver Gen.		x	x	x	x	x	
Advising			*	*			
Explain/Tutor/Navigate							
Decision Making	x		*	*			
Parameter Estimation	x		*				x
Error Estimation	x						
Interactive Debugging				x			
Interac. Runtime Ctrl							
Symbol./Num. Computing	x		x			x	
Vector Processing		x		x	x	x	
Parallel Processing			x				
Performance Estimation		x	x				
Document Generation							
Openness		x	x		x	x	
Extendability							

Table 4.1 PSE Functionality of 7 PDE Systems

An x denotes the presence of the functionality and a * indicates it is under development.

All the PDE systems listed are either domain or method specific and are not easily expandable. They deal with one or a limited class of problems such as second order elliptic PDEs in two and three dimensions as in ELLPACK or parabolic PDEs in one and two space dimensions as in RPI. Each PSE employs numerical methods optimized to solve its particular class of problems and most require a large or multi-processor type

system that may potentially lead to scalability concerns. Our interest lies more in environments for solving continuous simulation systems but as stated above, the PSE functionalities listed in Table 4.1 would be equally desirable for continuous simulation. A literature search found one PSE type system for ordinary differential equations, ODEXPERT, which is discussed in §4.3.2. As with the PDEs, it is not an extendable system and its scope is currently quite limited. Thus, it is likely that analogies will exist between any other ODE type system and the inadequacies of the PDE type systems.

Gallopoulas et al. [Gallopoulas et al., 1992] assert that the lack of application specific knowledge bases and appropriate integrated infrastructures of symbolic, numeric, geometrical, artificial intelligence, and natural languages facilities present principal barriers to the development of general and application specific PSEs. However, the application domain that is most tractable for a mature heterogeneous PSE is in the field of classical mathematics, an area narrow enough to be feasible to undertake currently, and which can be based on numerous available handbooks and software components. Once completed, the system could become a generic PSE for many areas of science and engineering.

ODEASY fits in this vision as a supporting component that has incorporated solutions from available handbooks. It is capable of extending the power and scope of an ODE PSE or of a more general mathematics PSE. This is consistent with the goal of a PSE as a self managing environment with interconnections to supporting tools. Current systems, though not fully developed as ideal PSEs, are implemented in this manner; for example, ELLPACK and //ELLPACK act as an interface to various libraries of parallel elliptic PDE solvers.

4.2 Continuous Simulation Languages

Simulation languages are designed to solve the state-space models of the type $y' = f(t, y(t), u(t))$ by applying numerical integration algorithms. Examples of such languages include ACSL, DESIRE, DSblock, CSSL IV, Simnon, and SIMULINK. These languages come with a comprehensive command set for simulation purposes and most

offer graphics capability and an equation sorter that enables the user to specify the model equations in an arbitrary sequence.

The user must learn the set of language operators needed to perform basic functions such as integration and differentiation and become familiar with the necessary format of the program which for most simulation languages is composed of distinct segments (known as the initial, dynamic, and terminal sections). In some cases, the dynamic segment is subdivided to form a derivative section to enclose all integration statements and there may be multiple derivative sections each with their own integration algorithm and step size. ACSL and CSSL IV allow for the separation of the model definition from the runtime commands so that a saved model may be subsequently analyzed using any set of runtime commands, specified either interactively or from a batch file. The CSSL IV runtime monitor permits the user to carry out interactive processes such as changing parameters, collecting data, and presenting results. This language also incorporates analysis tools such as a steady-state finder, fast Fourier transform processing, and linearization routines. Like CSSL IV, ACSL offers interactive usage and has the additional capability of imposing no limits on the number of state variables, the number of symbols, or the number of labels in the simulation program and hence, is well suited for large simulation studies.

A deficiency of simulation languages is that they have difficulty resolving mutual dependencies such as algebraic, or implicit, loops. This is due to the fact that the structure of the input is examined to determine the proper execution sequence and the variable to solve for in each equation. This sorting scheme does not permit mutual algebraic relations between variables. For example, in the two equations

$$y = f(x)$$

$$x = g(y)$$

x must be known before y can be computed from the first equation, but y must be known in order to compute x from the second equation and consequently, neither of the two equations can be computed without the other. Algebraic loops among variables may reflect interconnections between different objects and are unavoidable but they may also

be due to bad modeling, or rather, a bad choice of variables. Linear implicit loops can be solved by symbolic formula manipulation while nonlinear loops generally cannot. Methods for eliminating the algebraic loop include the introduction of a small time delay or adding a first order transfer function with a small time constant. Some simulation languages, such as CSMP-III, provide a special facility for overcoming the algebraic loop problem through an implicit function [Murray-Smith, 1995].

Thus, while simulation languages are powerful tools for simulation modeling with each line of code generating the equivalent of 20-250 lines of code in a general-purpose high level language such as FORTRAN or C, their comprehensiveness imposes a difficult interface. The user must respect numerous language specific restrictions including program layout, naming rules for variables, limitations on the mixing of common data types such as integer and real, etc. that are neither intuitive nor convenient. The learning curve that is involved precludes them from being ideal PSEs. They would benefit from an interface extension with features such as submodel handling, and formula manipulation techniques to transform familiar relationships into the equations acceptable to the simulation language. One experimental tool for this latter purpose is Dymola [Cellier and Elmqvist, 1993] which is an object-oriented, continuous-system model manipulator that can accept a more intuitive, object-oriented input to generate models in several simulation languages such as ACSL and DESIRE among others.

In comparison, the ODEASY system lies at the opposite end of the complexity spectrum, offering a user-friendly interface to a database of known analytic ODE solutions. It circumvents the numerical process that is required in commercial simulation languages for generating solutions to the model equations. Where a submodel is autonomous, i.e., no cross-coupling terms with variables from elsewhere in the model, ODEASY may be consulted for the analytic solution which, if found, reduces the problem size of the simulation run. Furthermore, the ODEASY solution for the submodel is presented without error since it is in closed form. Although the solution for the model as a whole is only as accurate as its least accurate part, the exact solution from ODEASY can neither contribute inaccuracy nor compound inaccuracies that result from the numerical process.

4.3 Knowledge-Bases

Two knowledge-based approaches to solving differential equations are discussed in this section. The first one, known as EVE, is a PDE solver that offers an advanced user interface and is innovative in its adoption of an object-oriented paradigm for its expert system. The other is ODEXPERT, an ODE solver for selecting appropriate numerical methods for initial value problems.

4.3.1 EVE

EVE is an object-centered knowledge-based PDE solver which represents numerical algorithms, mathematical entities, and equation decomposition tasks as object classes [Baras et al., 1992]. Implemented in Lisp, EVE relies heavily on Shirka, a frame-based model, to represent the knowledge it exploits. A frame, commonly referred to in the field of artificial intelligence as a record-like data structure, is equated to a class by EVE. Each class contains slots that describe the properties of the class' instances and every class is a specialization of super-classes, to which it is linked through an ako (a-kind-of) slot. The resulting structure of the knowledge base is an acyclic graph, whose root is called object. Thus, rather than apply production rules as the inferencing mechanism (which is typical of most expert systems), knowledge is gained through the inheritance of slot descriptions from higher classes. All the mathematical entities manipulated by EVE, the tasks to manipulate them, and the methods to solve them are represented as classes. The entities include equations, variables, parameters, systems of equations, boundary conditions, domains and sub-domains, matrices and linear systems and so on. Several concepts of the object-oriented paradigm are relaxed in order to support this knowledge representation model. There is no encapsulation because the slots in a class must be visible from the outside of the class to support reasoning mechanisms, such as instance classification. Moreover, some slots have their values inferred using inheritance, procedural attachment, or default assignment rather than set as part of the instantiation process of the class.

The EVE system works by instantiating the component parts of the input equation to appropriate subclasses of the object class according to a knowledge base of equation

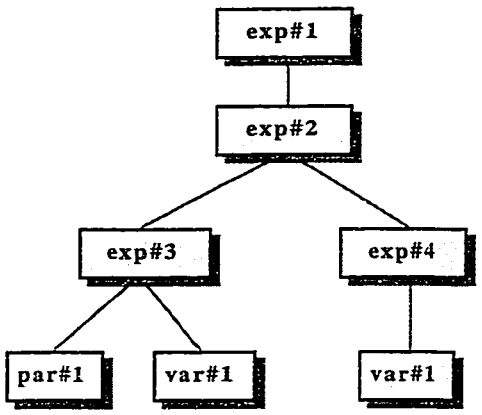
components. Based upon the characterization of each component, the reasoning mechanism deduces a task class which in turn, determines the most reasonable numerical methods class for solving the expression. The algorithms are retrieved from a knowledge base of numerical methods. Ultimately, all tasks are chained up in order to solve the current problem. Thus, the problem is one of finding the possible correct locations for the given objects in the class hierarchies whereby the classifying algorithm attempts to match object instances with the subclasses of the instantiation class. The knowledge on an object increases since the resulting potential classes are more specialized than its instantiation class.

At present, EVE accepts as input a fairly restrictive class of equations which can be described by the following generic formula in canonical form:

$$dy/dt - \mathrm{div}(A \overrightarrow{\mathrm{grad}}\, y) + \vec{b} \bullet \overrightarrow{\mathrm{grad}}\, u + cu = f \qquad \text{on} \quad \eta$$

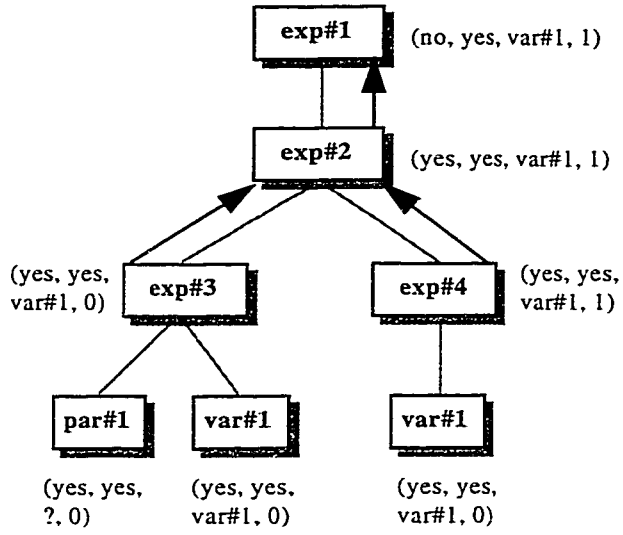
with appropriate boundary conditions, and where η is a bound set of $\mathbf{R}^n$ (n=1 or 2).

The components are instances of the classes operator and operand and of their subclasses. For example, Figure 4.1 shows a mathematical expression with its components represented as instances of the subclasses cos-operator, mult-operator, variable, and parameter. Only the pertinent slot descriptions are indicated in Figure 4.1. In fact, the subclasses all inherit additional slots from their superclasses and the values of these properties are updated and propagated back up to the root of the tree where they become the global properties of the equation as illustrated in Figure 4.2.




```
{exp#1
    is-a       = cos-operator
    operands = exp#2          }
{exp#2
    is-a       = add-operator
    operands = exp#3 exp#4    }
{exp#3
    is-a       = mult-operator
    operands = par#1 var#1    }
{exp#4
    is-a       = der/t-operator
    operands = var#1          }
{var#1
    is-a       = variable
    name       = "y"          }
{par#1
    is-a       = parameter
    name       = "a"          }
```

Figure 4.1 The EVE Representation (of the expression cos(a*y + *dy/dt*) where the instance exp#1=cos(a*y + *dy/dt*)).



Component instances inherit additional slots from the superclasses *operator* and *operand*. The last four slots of each instance below are inherited. Their propagation to the root is shown in the diagram.

```
{exp#1
    is-a       = cos-operator
    operands = exp#2
    linear     = no
    stationary= yes
    mono-var= var#1
    order      = 1                }

{exp#3
    is-a       = mult-operator
    operands = par#1 var#1
    linear     = yes
    stationary= yes
    mono-var= var#1
    order      = 0                }
```

Figure 4.2 Update of Component Instances in EVE.

The type of tasks and subtasks that are instantiated depend upon the properties determined at the expression and equation level. Similar hierarchies exist for the task classes and methods classes. The classification scheme and organization can be complex due to multiple inheritance which leads to very tangled hierarchies. Moreover, tasks and subtasks must be sequenced and their selection may depend on the results from the previous execution of tasks. For example, if the problem has been recognized as elliptic on a non-rectangular domain, a sequence operator will determine that the subtasks to

sequentially execute are mesh generation, interpolation, assembly of the matrix, and lastly, the solution process itself. The requirements for an extensive knowledge base and sophisticated classification and sequencing algorithms has thus far limited the capabilities of EVE to solving only linear systems of PDE in one and two dimension space. One of the main and more difficult extensions is to consider non-linear equations which would require a more powerful task planning algorithm in order to control the iterations of the solving process.

Despite its current limitations, EVE possess the potential of becoming a general PDE solver since the knowledge base is receptive for update and extensions. Knowledge is concentrated in the classes describing entities and in the classes describing methods rather than as a large number of simple, unstructured facts scattered across production rules typical of many expert systems. Incorporating details to existing knowledge is accomplished by adding slot descriptions to the corresponding class and the computation of new properties can be executed via procedures directly attached to the slots. Adding new information consists of creating a new subclass in the appropriate hierarchy. The advantage of inheritance will mean that only differences with existing knowledge need to be specified. This ability to assimilate new information provides potential for growth as a PSE. Once the organization of new information has been determined, knowledge and multi-media extensions largely involve classical software development, essentially writing or importing existing program modules, particularly if a multi-platform software environment becomes supported. The user interface is already fairly advanced in that it offers graphics capability, a domain editor, a display of the tree of tasks to explicitly represent the solution process, and a high level editor, shown in Figure 4.3, which accepts both point-and-click and keyboard entries. Operators appear as icons that can be selected and nested up to the desired level of complexity.

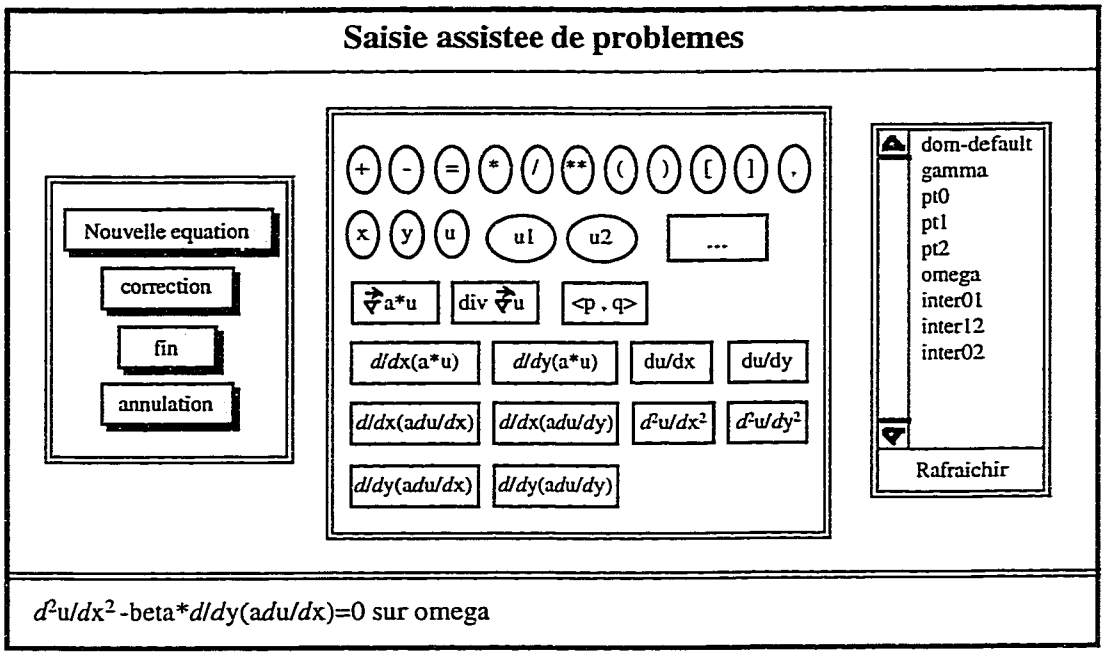


Figure 4.3 The EVE Editor of Equations

In the EVE equation editor, any icon clicked on by the user will have its corresponding operator appear in the bottom window to contribute to the formation of the mathematical expression. The user can then select an element of this expression and have it substituted by another operator or, in the case of a variable or parameter, by another name than the default one. Thus, the names of the variables and parameters are free and can be entered through the keyboard. The user also specifies the domain over which the equation has to be solved by selecting among a list of already known domains or by introducing the name of a new one. This same editor is also used to specify boundary conditions. This interface is advanced over those of other mathematical software in that no knowledge of commands is required to operate the system.

4.3.2 ODEXPERT

The second knowledge base system, ODEXPERT [Kamel et al., 1992], determines appropriate numerical solvers for initial value ODE problems by performing automated tests on the properties of the problem being studied. The various numerical techniques from different software packages that are applicable to solving initial value ODE systems presents a choice that may be difficult for non-expert users of ODE systems who need to

select the right solver to minimize errors and inefficient computations. This requirement for an intelligent solver selector provides the motivation for ODEXPERT which possess knowledge of several ODE solvers including commercial mathematical software libraries such as HARWELL, IMSL, NAG, and SLATEC and research codes such as LSODE, LSODES, LSODAR, RDEBD and SECDER. The system makes recommendations on appropriate solvers by investigating critical properties such as the linearity, stiffness, and structure of the input problem. ODEXPERT, shown in Figure 4.4 is a multiparadigm software system implemented in OPS83, a knowledge base development language, and interacts with components built in C, FORTRAN, and Maple.

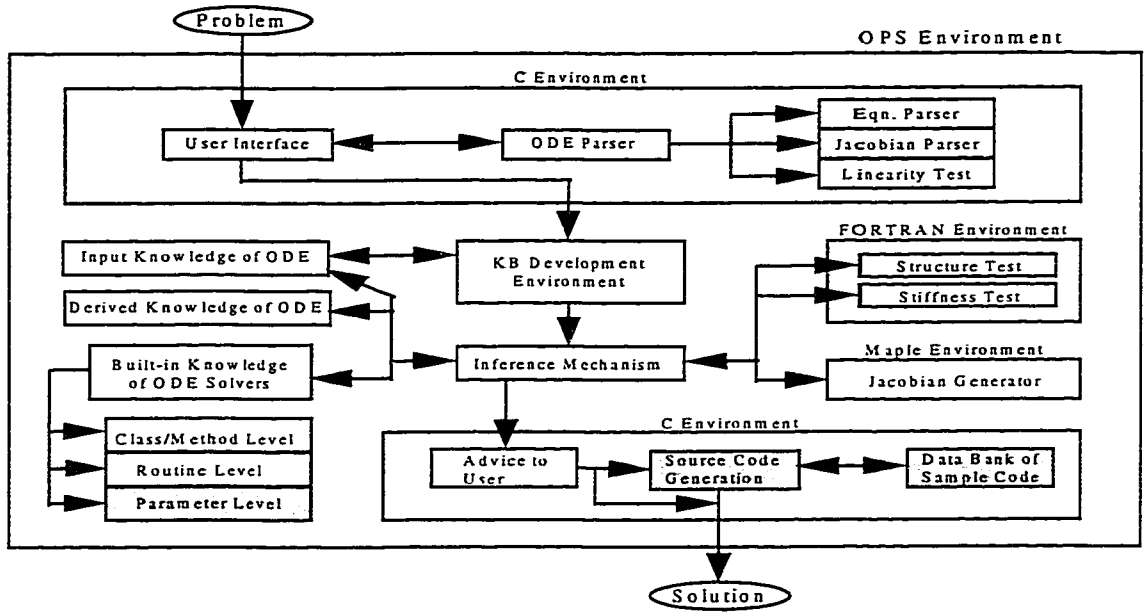


Figure 4.4 The System Structure of ODEXPERT.

Shaded areas of Figure 4.4 are planned for future development. The user interface accepts the problem specification via keyboard entry with the derivative term $d^n y/dx^n$ entered as y<n> except for first and second order derivatives which may be input as y' and y'', respectively. The syntax follows the language conventions of C. Equations must be in explicit format and be terminated by a semicolon. The ODE parser transforms higher order equations into equivalent first order ODE systems and changes all systems into FORTRAN syntax. For example, the input equation

$$y'' = (1 - y\text{^}2) * y' - y;$$

will be changed into the following system:

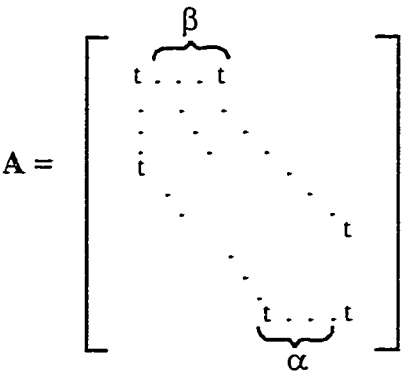
```
dy(1) = y(2)
dy(2) = (1 - y(1) **2.0) * y(2) - y(1)
```

Besides the input equations, ODEXPERT interactively requests from the user information on the properties of the input system. Any information not available from the user is determined through automatic testing. For example, the parser deduces the complexity of the problem in terms of the number of additions, subtractions, divisions, multiplications, exponentiations and function evaluations. It can also determine the linearity using the simple rule that an equation is linear if it can be expressed as:

$$\frac{d^n y}{dt^n} = f(t) + \sum_{k=0}^{n-1} f_k(t) \frac{d^k y}{dt^k}$$

where $d^0 y/dt^0 = y$ and $y(t), f_k(t)$ are functions of t. However, this rule is not foolproof because the parser does not perform any simplification or symbolic computation so that it is possible to misclassify a linear equation as non-linear.

The Maple symbolic computing environment is used to generate the Jacobian (where the Jacobian J is an n x n matrix such that Jij=dy_i'/dy_j for a first order ODE system of dimension n, i.e., $dy_i/dt = y_i' = f_i(t, y_1, y_2, ..., y_n)$ for $i = 1...n$. The Jacobian matrix is required for stiffness testing and to exploit the spatial structure of the system. The latter task is performed by examining the n x n matrix $\mathbf{A}=[a_{ij}]$ which provides the semi-bandwidths and where :



$$\alpha = \max_{\substack{a_{ij} \neq 0 \\ i \geq j}} (i - j)$$

$$\beta = \max_{\substack{a_{ij} \neq 0 \\ i \leq j}} (j - i)$$

The semi-bandwidths indicate the spatial structure of the matrix by their values, for example,

α value	β value	Structure
0	0	Diagonal
0	1	Upper Bidiagonal
1	0	Lower Bidiagonal
1	1	Tridiagonal
0	β>1	Upper Triangular
:	:	:
:	:	:

Outside of simply detecting the existing spatial structure of the system, ODEXPERT is able in some cases to permute rows and columns of a matrix to achieve certain desirable structures if no recognizable spatial structure is found in the original system. Using the maximal transversal algorithm and Tarjan's algorithm for block triangularization from the HARWELL library, the system may attempt to triangularize the matrix. It may also try to reduce the bandwidth of the matrix using an algorithm modified from the Reverse Cuthill-Mckee algorithm and that by [Gibbs et al., 1976].

The stiffness test uses the Jacobian in a non-stiff ODE solver, the DVERK subroutine from IMSL, to integrate the problem over a few steps backward starting from the end-point. By measuring the rate of change of the solution vector, a stiffness indicator is determined. This simple technique for detecting stiffness avoids the overhead that is associated with existing methods of dealing with stiffness during the course of problem solving [Petzold, 1983] since the purpose of ODEXPERT is to detect potential stiffness quickly so that the choice of an appropriate stiff solver can be recommended. However,

expediency is traded for some inaccuracy in the detection which has a success rate of 85.7% in classifying non-stiff problems and 93.5% in classifying stiff problems.

Once the essential knowledge of the input problem has been acquired, the inference mechanism relates the properties to an appropriate method. The inferencing is a standard rule based scheme with a collection of rules that interact with a global database or working memory. Each rule consists of a conditional expression on the left-hand-side (LHS) and an unconditional sequence of actions on the right-hand-side (RHS) which is executed if the conditions of the corresponding LHS are satisfied given the current contents of the working memory. In the event that there are more than one rule whose LHS match the working memory, the one with the most specific LHS is selected to ensure that special case rules supercede general case rules. This is similar to the instantiation of subclasses in EVE but without the organization of knowledge. Here, the knowledge is advanced by each execution of a RHS which effects changes in the working memory so that a new set of LHSs may be satisfied in the next cycle. Using the rule-based language of OPS83, a typical rule might be:

```
rule adams1 -- recommend Adams method for non-stiff problems
{  &1(kb_level1 stiffness=non_stiff; accuracy=high; continuity=continuous;
        method=undefined);
   --> modify &1 (method = Adams);  };
```

The name of the rule is "adams1" and it states that if the system is continuous, non-stiff and high accuracy is required for the solution, then the Adams method is recommended. In general, either the Runge-Kutta method or Adams method is recommended for non-stiff problems and Backward Differentiation Formulae (BDF) is suggested for stiff problems.

Following the determination of an appropriate method, the production system searches for subroutines that implement the recommendation. The final report to the user describes the system in first order form and its Jacobian, summarizes its properties and structure, and provides appropriate choices of subroutines from the various software

libraries along with interpolation drivers that may be used in conjunction. For example, the recommendation section of the report for a stiff system might read:

:

> BDF method is recommended.
>
> With pseudo code 1, Subroutine LSODE with MF = 21 or 22 can be used with interpolation driver 3.
>
> BDF method is recommended.
>
> With pseudo code 1, Subroutine DEBDF of SLATEC can be used with interpolation driver 2.
>
> BDF method is recommended.
>
> With pseudo code 1, Subroutine D02NBF of NAG can be used. Setup routine D02NVF and interpolation routine D02XKF should be used.
>
> BDF method is recommended.
>
> With pseudo code 1, Subroutine DC03 of HARWELL can be used.

:

:

The "pseudo code 1" refers to a program outline suitable to be used with the particular routine.

The contribution of ODEXPERT is its ability to advise users on the appropriate method and software to apply to a given ODE system. It succeeds in the PSE capacity of providing advice without relying heavily on user supplied information. The system is also advanced in its ability to accept higher order systems that it can translate into first order since most available ODE solvers accept first order ODE systems only. However, the possibility of erroneous classification on the critical stiffness property may detract from its practical application given that non-stiff ODE solvers tend to degrade drastically with stiff problems. Also, numerical methods have difficulty coping with discontinuities in the integration interval, a property for which there is no automatic testing and the system simply assumes continuity. Its potential as a PSE would improve with greater insight into these aspects of the input system. It might be possible to include a more thorough test for stiffness and one for potential discontinuity and have the user select

between expediency and accuracy. The utility of ODEXPERT would increase with the planned implementation of fuller recommendations with specific source code generation and appropriate parameter value settings rather than general pseudo code. The user interface of ODEXPERT could also become more uniform. The current environment accepts equation entries in C syntax where order n derivatives for n>2 must be keyed in as an unintuitive "y<n>"s. Inconsistent with the input, the output display employs FORTRAN syntax.

4.4 Mathematical Software

Mathematical software such as Maple [Burkhardt, 1994; Abell and Braselton, 1994] and Mathematica [Abell and Braselton, 1997; Gray, 1994], both PC-based products, provide a symbolic and algebraic computing system. With respect to differential equations, they are similar and provide functions to produce either an analytic solution or a numeric one. Both provide an identical command, although in different syntax, to perform differentiation. For example, the command to solve the differential equation "y'' – 5 * y' + 6 * y = 0" is dsolve(diff(y(x),x,x) – 5 * diff(y(x),x) + 6 * y(x)=0, y(x)) in Maple and dsolve[y''[x] – 5 y'[x] + 6 y[x] ==0, y[x], x] in Mathematica. The "diff" is the differentiation operator in Maple and is required to specify that the derivative is with respect to x. The last parameter indicates that the equation is to be solved for the function y(x). In a similar way, the Mathematica command needs both the dependent and independent variable to be explicitly specified as the last two parameters. Both packages also offer related features such as Taylor series, power series, and Laplace transforms, among many others, and can be used to solve for systems of differential equations and initial value problems.

In terms of actual functionality, these general math packages are geared towards a wide range of applications that are on a smaller scale than most practical simulation runs. The math packages tend not to be able to deal well with equations of order beyond 2, and their numerical solvers can be overwhelmed by problems of stiffness, discontinuities, and singularities. The Mathematica numerical solver handles the first two conditions better with an adaptive step mechanism that reduces the step size when the solution appears to

be varying rapidly in a particular region. However, this method cannot cope with a singularity. The Maple solver is based on the Runge-Kutta technique that can fall prey to any of the conditions. In fact, Maple has been cited as weak by [Heck, 1993] who claims "Maple's capabilities in solving ODEs are rather limited: in most cases, solutions are only found for ODEs of degree 1 and order smaller than 3. Little knowledge about special functions is incorporated in the differential equation solver (with the exception of the Bessel functions)". To be fair, these packages are moving targets with deficiencies that are eliminated with each successive version.

Although the input formats of Maple and Mathematica are not intuitive and requires an explicit indication of the state variables that should be possible for the program to deduce, both systems offer superior output display. They provide a two-dimensional, notationally accurate display of results with rational expressions reducible to lowest terms for the display. As a PSE, the interface of these packages is an improvement over the manipulations needed to input and process the same system with a simulation language. However, as discussed, their functionality falls short of the ideal PSE. Even though they are not intended as simulation languages which must offer features suited for complex simulation runs including the ability to handle hundreds of state variables and statements, and macro extensions, these mathematical software can be improved with additional features for general ODE solving.

4.5 The Role and Development of ODEASY

For continuous system simulation applications using simulation languages, ODEASY may be able reduce the size of the problem. The exact solution it provides for a submodel will not contribute inaccuracy or compound inaccuracies that occur as a result of the numerical processing of the remainder of the model.

The Eve knowledge-based PDE solver would benefit from a supporting tool such as a PDE equivalent to ODEASY. This will help to expand the scope of the problem class that it can support which currently is fairly limited. Similarly, a database of analytical solutions can address the deficiencies of mathematical software with respect to their abilities to deal with higher order equations and systems characterized by stiffness,

discontinuities, and singularities. As maturing PSEs, these systems should standardize data import/export rules for interfacing with supporting structures. As a supporting tool, ODEASY should be generic enough as to be an easily assembled software part.

The rationale for ODEASY has been established and issues of its development need to be investigated. Eve experienced limitations with its complex reasoning and sequencing schemes to characterize expressions and has only achieved this for a restricted class of equations. Likewise, ODEXPERT may misclassify on the linearity and stiffness properties, largely because its parser does not perform any simplification or symbolic computation. The design of ODEASY should avoid these problems since it is not necessary for it to achieve the depth of understanding needed by the knowledge-bases which must select a numerical method of solution. In fact, the success of ODEASY depends upon its ability to identify semantically equivalent ODEs entered in dissimilar forms as being the same. This matching dispenses with the need to recognize the properties of the equation. An equation indexing or element sorting method is required and is further explored in §5.

The advanced interface of EVE presents a benchmark for the ODEASY interface development. A point-and-click system with icons and the option to use any names for variables and parameters would assist in the user interaction.

5. MATCHING AN ORDINARY DIFFERENTIAL EQUATION

To access the right analytic solution in the ODEASY database, an input equation must be matched in some way with the equation in the database that corresponds to the analytic solution. For example, the system must be able to recognize that the following two entries are the same:

$$y' + \sin(t) * \cos(\tan(y) + h) = 0 \quad \text{and} \quad y' + \cos(h + \tan(y)) * \sin(t) = 0$$

Furthermore, these two ODEs must be seen as distinct from one that involves similar terms but that is mathematically different such as $y' + \sin(\tan(t) + h) * \cos(y) = 0$. If possible, the method for obtaining this awareness should avoid the detailed component decomposition associated with Eve as this may lead to a similar level of unmanageable complexity. The development of two methods for matching ODEs are discussed in the following two sections.

5.1 An Indexing Method

The common approach, popular in calculators and computing languages, of representing equations in postfix form via pushdown stacks and parse trees [Sedgewick, 1989] is inappropriate for a symbolic database like ODEASY. Where stacks and parse trees are structures that promote the efficient evaluation of arithmetic expressions, ODEASY performs no computations on the ODEs. For example, in a stack, the equivalent expressions (1) "5 * ((9 * 8) + 7)" and (2) "(7 + (8 * 9)) * 5" will have the same outcome through the following series of calls:

(1) "5 * ((9 * 8) + 7)" (5 9 8 * 7 + * postfix)

Push(5)

Push(9)

Push(8)

Push(Pop * Pop)

(2) "(7 + (8 * 9)) * 5" (7 8 9 * + 5 * postfix)

Push(7)

Push(8)

Push(9)

Push(Pop * Pop)

Push(7)	Push(Pop + Pop)
Push(Pop + Pop)	Push(5)
Push(Pop * Pop)	Push(Pop * Pop)

Both result in an answer of 395. Likewise, for the parse tree, it does not matter whether its structure is based on (1) or (2). The tree traversal evaluation will have the same outcome. However, with a symbolic database, the recognition of (1) is separate from that of (2) and the order of entry of the input can determine the success or failure of a search. Thus, some other representation for accessing the database needs to be explored.

One possibility is to employ an index that indicates the existence of functions and operators in an ODE. This would involve creating a classification system sufficiently comprehensive to distinguish the various types of operands and having distinct codes to represent the operators. The design of such an index is discussed next.

To serve as a key into the symbolic database, the index must be sensitive to the types of operands and the operations performed on them. Ignoring this relationship would be to treat the equation as a random collection of operators and operands and risk erroneous identification of similar equations. It is essential that the index capture the local sequencing of operators and operands. This would involve maintaining an awareness of the right operand of each addition and subtraction operation; that is, identifying such operands as either positive or negative elements. This can be simply done via an encoding of operator-operand pairs.

A more stringent requirement is necessary for the operations of multiplication, division, and exponentiation in that both the left and right operands must be tracked together. While this is obvious for the latter two operations, the case for multiplication can be illustrated by an expression such as "a * c + b" which would be confused to be equal to "a + b * c" if only the immediate left operators of each operand were considered. This last point means that the index must include an additional tagging system on the encoding for operator-operand pairs. This issue will be resolved later. For now, the encoding of each pair as operator code-operand code comprise the index. However, the index string must

have its elements in the same order irrespective of the sequence that the pairs appear in the equation. This is a requirement since the same equation can be entered with a different sequencing of terms. This is resolved by having an ordering on the index elements enforced through a sort code on each type of operand. Given that operand types are encoded, those codes may also serve as the sort codes. A set of codes for equation elements, operators, and structures for this index is shown in Table 5.1. Structures refer to parentheses and arguments of functions that must also be incorporated into the index.

Table 5.1 Operand codes for index method

Operand Code	Operand Type	Operand Code	Operand Type	Operand Code	Operand Type	Oprtr/ Strctr Code	Oprtr/ Strctr Code
1	y'''''	9	f(y)	17	arctan()	A	+
2	y''''	10	cos()	18	ln()	S	-
3	y'''	11	sin()	19	exp()	M	*
4	y''	12	tan()	20	e	D	/
5	y'	13	csc()	21	pi	E	^
6	t	14	sec()	22	alpha	P	bracket
7	y	15	arccos()	23	numeric	G	argument
8	g(t)	16	arcsin()			L	left operand
						R	right operand

Notes: (1) t denotes the independent variable, y denotes the dependent variable.

(2) "alpha" and "numeric" refer to alphabetic and numeric constants.

(3) Brackets and function arguments, denoted by codes P and G, respectively, are structural elements that must be considered by the index.

(4) "Oprtr" denotes operator and "Strctr" denotes structure.

Table 5.1 offers one possible sort order (via the operand codes) for the index. In fact, any arbitrary but fixed positioning that prevents the index from being a random interspersion of encoded operand-operator pairs would aid it to be an effective key into the database. The purpose of the system is to have a rigidly ordered index that is independent of the order in which the elements and operations appear in the equation and hence, places no restrictions on the sequence in which the equation elements may entered. The issue of maintaining the relationship of left and right operands in multiplication, division, and

exponentiation is dealt with by adding a numbered tag field to the operator codes. The tag field relates a right operand to the associated left operand by having the same (unique) number on the tags of the operator codes involved. This applies only to the operand of multiplication, division, and exponentiation. Since the sorting is by operand code, this will not impact the sort. The sort has been chosen to be in ascending order.

Following is the general scheme that apply the codes of Table 5.1 to generate the index:

1. The first element in the equation is assigned a left operator of "+" unless the equation begins with a negative element. In the index, operator-operand pairs in the equation are represented by their respective codes from Table 5.1 with each pair or combination of pairs separated by a period. A sequence of codes terminated by a period is referred to as an index substring. Each substring begins with the operand code.

2. After the operator code, a P code appears if the expression is enclosed in parentheses. The purpose of the P code is to indicate which operands occur at the various levels of bracketing in an expression. The tracking is performed by an associated integer field to the P code, the p_count, which shows the number of unmatched left parentheses at the point at which the operand occurs in the equation. The p_count begins at 1 for each new parenthesized expression (that is not nested within a longer parenthesized term).

   Within each expression, the p_count is incremented by 1 for every occurrence of a left parenthesis and reduced by 1 for every occurrence of a right parenthesis. When the p_count is reduced to zero, all bracket levels have been completed and the P code and p_count no longer appear in subsequent substrings. The P code and non-zero p_count is appended to every substring of operands nested within one or more levels of bracketing. This method captures the levels that operands occur at while preventing the sequence in which the operands are parenthesized in the input equation from producing alternative encodings in the index. The P code and p_count may be used in other substrings to reference the associated parenthesized term. As an example, the equivalent expressions "(t + (y + a))" and "((a + y) + t)" are encoded as

"6AP1.7AP2.22AP2." and "22AP2.7AP2.6AP1.", respectively. Both would be sorted to the same form, "6AP1.7AP2.22AP2.", in the index.

3. A subtraction of a parenthesized term has the negation captured in an auxiliary substring. Auxiliary substrings are placed at the end of the index and are a record of the negative parenthesized terms and negative multiplication terms in the equation. Each auxiliary substring is prefixed with a "z" (to ensure that it sorts to the end of the index).

4. Function arguments are denoted by the G code which follow any P codes in the index substrings. The purpose of the G code is to link functions to their parenthesized arguments. For functions with simple arguments composed of a single atomic element, i.e., a variable or a constant, the index representation is within one substring with the G code separating the left operator code of the function and the operand code of the argument. Encoding for functions of single elements are patterned in the following order:

   operand code of the function, operator code of its left operator, tag required by the left operator (optional), G code, operand code of the argument, P code and p_count (optional), and M code and m_count (optional).

   The tag is used to link operands encoded in different substrings. The P code and p_count occur if the function is within a parenthesized term while the M code and m_count is needed if the function is part of a multiplication operand. More on the M code later. As an example of simple function encoding, the expression "sin(y) – cos(t)" is coded as "11AG7.10SG6." which in sorted order would be "10SG6.11AG7.".

   For function arguments that extend beyond one atomic element and hence, require encoding in one or more substrings, the G code has an associated integer field, the g_count, which is incremented by 1 for each new non-atomic function argument in the equation. In the function substring, the G code and g_count occur after the P code and p_count (if they exists) but before the M code and m_count (if they exist). In the

argument substring(s), the G code and g_count are the last items. For example, the expression "sin(y + 1) + cos(t – y)" is encoded as "11AG1.7AP1G1.23AP1G1.10AG2.6AP1G2.7SP1G2." and be sorted to "6AP1G2.7AP1G1.7SP1G2.10AG2.11AG1.23AP1G1.". When the index is complete and sorted, the g_count fields are re-numbered, starting at 1, whereby all g_count fields of the same number will receive the same updated number. This process removes any dependency in the index on the ordering of the functions in the input equation without altering the semantics. For the example just sorted, the re-numbered g_count results in the index "6AP1G1.7AP1G2.7SP1G1.10AG1.11AG2.23AP1G2.".

5. In parsing the equation (from left to right), if the next operator is "*" and the current left operator is "+" or "-", then the current operator is encoded as "*". The purpose of this 'overwrite' of the left operator is to refrain from imposing a sequence on the multiplication operands since "a * b"="b * a" (commutative property) and the objective is to produce an index which is independent of the order of the equation elements. For example, a term such as "t * y" would have been encoded as "6A.7M." while the equivalent term "y * t" would be "7A.6M." or in the sorted index, it would be referred to as "6M.7A.". By changing the first operator code to "M" for each case, the index is consistent regardless of the order of operand entry.

   Thus, while it is important to track which operand types participate in the multiplication, the index avoids recognizing their order of occurrence. Instead, all substrings that belong to the same multiplication sequence are tagged with the same integer, referred to as m_count. The m_count begins at 1 and is increased by 1 for each multiplication sequence in the equation. If the term in the previous example is the first multiplication term in an equation, it is encoded as "6M1.7M1." and if it is the second term, it becomes "6M2.7M2.". After the index has been sorted, the m_counts are re-numbered, starting at 1, in order to remove any dependency of when the multiplication terms occur in the equation. All operands of the same m_count will receive the same re-numbered m_count. For example, the equivalent expressions "g(t) * y + f(y) * t" and "t * f(y) + y * g(t)" are encoded as "8M1.7M1.9M2.6M2."

and "6M1.9M1.7M2.8M2." which would sort to "6M2.7M1.8M1.9M2." and "6M1.7M2.8M2.9M1.", respectively. When the m_counts are re-numbered, both codes become "6M1.7M2.8M2.9M1." which reflect the operand types of two multiplication terms but is independent of the order of occurrence of the operands in each term and of each term in the expression.

6. If the multiplication term is negative, the auxiliary substring (refer to 3) for indicating negative terms is appended with the M code and the term's m_count. This preserves the knowledge that a particular term, which had its original left operator of "-" replaced by the code for "*", is negative. Auxiliary substrings are prefixed with "z" and are sorted to the end of the index. During the re-numbering process, m_counts in auxiliary substrings are replaced by their updated m_counts.

7. The multiplication of parenthesized terms is approached by first encoding each parenthesized term, then adding a substring beginning with the M code and m_count followed by the P code and p_count of the parenthesized term representing the operand. A separate substring is required for each parenthesized term participating in the multiplication. For example, the expression "(b * (t + 1) + t - y * a)" is encoded as "22M1P1.6AP2.23AP2.M1P2.6AP1.7M2P1.22M2P1.zM2." which sorts to "6AP1.6AP2.7M2P1.22M1P1.22M2P1.23AP2.M1P2.zM2.". The "M1P2." substring refers to an operand parenthesized within a second level of brackets, as denoted by "P2", that is multiplied with an operand related by "M1". The index shows that there are two positive elements within P2 of encoded operand types 6 and 23 and that the operand related by M1 for this multiplication sequence is of operand type 22. The auxiliary substring "zM2" indicates that the second multiplication term is negative. Lastly, the index is made generic through the re-numbering of the m_counts which produces the semantically identical index "6AP1.6AP2.7M1P1.22M2P1.22M1P1.23AP2.M2P2.zM1.".

8. For multiplication operands that are also division terms or exponentiation terms, e.g., a*b/c or a^(b) * c, the M code and m_count is appended to the term's substring in the case of simple division and exponentiation. If the operation involves one or more

parenthesized terms, then it cannot be represented by one substring. In this latter case, it is sufficient to append the M code and m_count to the denominator substrings or the exponent argument substrings as an indicator that the entire term is a multiplication operand.

9. With simple division terms where neither the numerator nor denominator is parenthesized, the entire term is represented by the combination of the left operand-operator pair and the right operand-operator pair in one substring. Unlike multiplication, the left and right operands of division are not commutable and hence, their order of occurrence is recognized by the index. Similarly, for the exponentiation operation where the base is not parenthesized, the entire term is captured in one substring with any enclosing parentheses of the exponent argument ignored by (since the absence of these brackets do not impact the semantics of the index expression). If the division or exponent term is either the left or right operand of a multiplication operation, then the numerator or the exponent base is encoded as if it is of the "+" operator (or of the "-" operator if it is a negative term). The M code and m_count is appended to the end of the operand substring. For example, the expression " - t/b * y^(a)" is encoded as "6SD22M1.7AE22M1.".

10. For division terms where either or both of the operands are parenthesized expressions, several substrings are required to encode the operation. The parenthesized operand is treated as a parenthesized term represented in two or more substrings and it is referenced via its lowest (outermost) P code and p_count which takes the place of the operand in the division encoding. For example, the division term "(a * t)/(y - b)" is encoded as "22M1P1.6M1P1.7AP2.22SP2.P1DP2." which becomes "6M1P1.7AP2.22M1P1.22SP2.P1DP2." when sorted.

This indexing scheme is successful in generating a unique yet generic index for ODEs that do not involve several non-nested parenthesized terms, (not including the simple arguments of functions). For example, the equivalent equations

"y' - b * sin(y) * ln(t) + sin(cost(t)) + a * y = 0"

and "y' + a * y + sin(cos(t)) – b * sin(y) * ln(t) = 0"

are both represented by the unique index
"1A.7M1.11AG10AG6.11M2G7.18M2G6.22M1.22M2.zM2.".

Similarly, the equivalent equations

"y' + t^(b) + (y + (a * t))/(t + b) = 0"

and "y' + ((t * a) + y)/(b + t) + t^(b) = 0"

are both encoded as "1A.6AE22.6AP3.6M1P2.7AP1.22AP3.22M1P2.P1DP3.".

The limitation on this index occurs when there are two or more non-nested parenthesized terms due to the ambiguity presented by the p_count. If the previous equation had a parenthesized exponent base as in "y' + (t + 1)^(b) + (y + (a * t))/(t + b) = 0", its index would be "1A.6AP1.6AP3.6M1P2.7AP1.22AP3.22M1P2.23AP1.P1DP3.P1E22.". Due to the two sets of parentheses, "6AP1.23AP1." and "7AP1.", being denoted by the same p_count, it is no longer clear whether the equation components should be

"(t + 1)^(b) + (y + (a * t))/(t + b)",

"(y + (a * t))^(b) + (t + 1)/(t + b)",

"(y + 1)^(b) + (t + (a * t))/(t + b)",

or some other combination. The alternative of not resetting the p_count to 1 for each new non-nested parenthesized term would resolve this ambiguity but defeat the purpose of tracking the bracket level that operands occur at. This produces an index unable to distinguish between multiple levels of bracket nesting versus a multiple number of first level parenthesized terms. For this reason, it is also not possible to use the m_count and g_count re-numbering process on the p_count since it destroys the nesting relationship indicated by the variable.

Thus, this indexing system cannot provide a unique key for all ODEs but does offer independence from the ordering of the equation elements. Methods to enforce a unique key would ultimately pay the price of independence from the equation ordering, a

sacrifice that would render the index inflexible to variations in the input form and places the onus on the user to follow rigid precedence rules in entering the equation elements. Alternatively, it is possible to have a sorted database whereby the results of all keys of the same code sequence are presented to the user as potential solutions. Note that the input ODE cannot be matched with accuracy against the database ODEs associated with these solutions due to possible differences in the ordering of the elements. Rather, it will be up to the user to judge which, if any, of the solutions is relevant.

5.2 A Method for Organizing Terms

The lack of precision of the indexing method requires that the user undertake some form of pattern matching between the input ODE and the database ODE. This suggests that an alternative strategy may be to use the input ODE to perform the indexing, provided that the equation can be made into a generic representation of all possible orderings of itself. By ordering the elements of the equation rather than generating an encoded description of them, the problem of non-uniqueness in the index is avoided. This can be realized by applying a predetermined sequence to re-order the equation elements.

A priority is assigned to each type of element so that all elements are placed in an ordered sequence in the equation. This system is data driven rather than operator driven – that is, the emphasis for sorting is placed on what the elements are and not on what algebraic operation is involved. The operators are respected insofar as the semantic of the equation is maintained; for example, a subtraction term is never erroneously sorted to become an addition term and consecutive multiplication elements never become separated by other operators through the sort. However, the operators themselves play no part in the sorting process. A straightforward approach to classifying elements is to provide a set of priorities that reflect the most commonly accepted format for the presentation of ODEs. This would include placing the highest order derivative at the leftmost position followed by the second highest order derivative, if any, and so on.

To enforce precision in the sort, certain rules apply. These rules include ranking the independent variable, say t, ahead of the dependent variable, say y, and the state variables

ahead of functions, and in turn, the functions of the independent variable ahead of functions of the dependent variable, and as the lowest rank, any constant values. The following transcendental functions are considered: cos(), sin(), tan(), arccos(), arcsin(), arctan(), sec(), csc(), ln(), and exp() and their ranking, as indicated in Table 5.2, is dependent upon whether they are a function of the independent variable, the dependent variable, some combination of both, or neither.

An ascending sort is used whereby the lowest priority element is ranked first placing it at the leftmost position of the equation. An expression composed of two or more elements is assigned the largest of its element priorities so that its place in the equation is dictated by the lowest rank among its elements. The arrangement of priorities for determining such a sort order is partially shown in Table 5.2. This ascending order set provides a priority for 238 categories of terms that include multiplication of transcendentals and state variables. Recognition of these multiplication sequences is necessary because, unlike the addition and subtraction operators that permit their left operands to be separated by the sort without impacting the semantics of the equation, the multiplication operator must apply to both its original left and right operands. For example, the equation $y' + \sin(t) * t - f(y) = 0$ cannot be sorted to become $y' * t - f(y) + \sin(t) = 0$ even though the priorities of the individual elements would have resulted in such a sequence. Hence, the need to treat consecutive multiplication elements and functions as one term.

The sort is further refined by setting variables and functions raised to a power to a higher ranking than their equivalent without the exponent, and by providing unique priorities to transcendental functions based upon various permutations of the state variables that comprise their arguments. These last two additional criteria causes a significant expansion in the number of categories because sequences of up to two transcendentals are considered. Each transcendental has exponentiation and non-exponentiation versions and for each version, additional refinements represent functions of seven different permutations of arguments based on the independent and dependent variables. These permutations may contain other elements and operators as denoted by the ellipses in the table; for example, the argument template "(...t...y...)" is satisfied by arguments such as "(a * t – b * y)", "(t + y)", and "(t * y + m)", among many others. The goal here is to be

able to classify a transcendental if it has an argument containing an independent variable first and a dependent variable second.

To use the priorities of Table 5.2, each equation element is compared with the element templates (shown in columns 1 and 3) until a match is found and the associated priority assigned. Any ties in priorities are resolved alphabetically during the sort. The templates are accessed in the sequence they are presented in Table 5.2 and once a match is found, the comparison does not continue with further templates. For this reason,

i. the derivatives are located at the top of the table since any term with a derivative must be sorted to the beginning of the ODE

ii. the next set of terms are transcendental functions with the multiplication sequences first and the single transcendentals at the end of the set. This indicates that if a multiple transcendental is not found, then the search attempts to match the term as a single transcendental.

iii. the third and fourth groups of templates are the functions of the state variables, and the state variables themselves. Note that the priorities, which have been increasing in the preceding transcendentals group, now revert to a higher ranking since these element types are to be ordered to an earlier leftmost position than the transcendental functions. Their position in the table following the transcendental functions reflects the need to verify that the term, which may be a multiplication sequence, does not already contain one or more lower ranking transcendental functions. If it does, then the term must be assigned the priority of the transcendental in accordance with the sorting rule that terms be assigned the lowest rank from among its subterms. For example, a term such as "t * sin(y)" would successfully match the template "trans(...y...)" and be assigned the corresponding priority of 225. The table search then terminates preventing a match with the higher priority "t" template that occurs later on in the table.

iv. the last group of templates with the lowest priorities, from 226 to 238, represents the different forms of constants. Located as the final set in the table, this group is

reached if no other matches are found. This will result in a sort that places constants at the end of the ODE and for multiplication sequences consisting of non-constants and constants, the constants will have no impact on the rank of the term. This is a deliberate relaxation of the rule that assigns to a term the lowest rank among its elements. Precluding the priorities of constants in such terms is necessary since the appearance of constants in multiplication sequences is common enough to mask the desired ordering if their priorities were to be considered here.

Table 5.2 Partial Listing of Term Sorting Priorities

Equation Element Template	Priority	Equation Element Template	Priority
y'	1	:	:
y''	2	:	:
y'''	3	:	:
y''''	4	trans^(...)(...t...t...)	214
y'''''	5	trans^(...)(...t...y...)	215
trans^(...)(...t...t...) * trans^(...)(...t...t...)	22	trans^(...)(...y...t...)	216
trans^(...)(...t...t...) * trans^(...)(...t...y...)	23	trans^(...)(...y...y...)	217
trans^(...)(...t...t...) * trans^(...)(...y...t...)	24	trans^(...)(...t...)	218
trans^(...)(...t...t...) * trans^(...)(...y...y...)	25	trans^(...)(...y...)	219
trans^(...)(...t...t...) * trans^(...)(...t...)	26	trans(...t...t...)	220
trans^(...)(...t...t...) * trans^(...)(...y...)	27	trans(...t...y...)	221
trans^(...)(...t...t...) * trans^(...)(...)	28	trans(...y...t...)	222
trans^(...)(...t...y...) * trans^(...)(...t...t...)	29	trans(...y...y...)	223
trans^(...)(...t...y...) * trans^(...)(...t...y...)	30	trans(...t...)	224
trans^(...)(...t...y...) * trans^(...)(...y...t...)	31	trans(...y...)	225
trans^(...)(...t...y...) * trans^(...)(...y...y...)	32	g^(...)(...t...)	18
trans^(...)(...t...y...) * trans^(...)(...t...)	33	g(...t...)	19
trans^(...)(...t...y...) * trans^(...)(...y...)	34	f^(...)(...y...)	20
trans^(...)(...t...y...) * trans^(...)(...)	35	f(...y...)	21
trans^(...)(...y...t...) * trans^(...)(...t...t...)	36	t^(...) * ... * y^(...)	6
trans^(...)(...y...t...) * trans^(...)(...t...y...)	37	y^(...) * ... * t^(...)	7
trans^(...)(...y...t...) * trans^(...)(...y...t...)	38	t^(...) * ... * y	8
trans^(...)(...y...t...) * trans^(...)(...y...y...)	39	y * ... * t^(...)	9
trans^(...)(...y...t...) * trans^(...)(...t...)	40	t * ... * y^(...)	10
trans^(...)(...y...t...) * trans^(...)(...y...)	41	y^(...) * ... * t	11
trans^(...)(...y...t...) * trans^(...)(...)	42	t * ... * y	12
trans^(...)(...y...y...) * trans^(...)(...t...t...)	43	y * ... * t	13

trans^(...)(...y...y...) * trans^(...)(...t...y...)	44	t^(...)	14
trans^(...)(...y...y...) * trans^(...)(...y...t...)	45	t	15
trans^(...)(...y...y...) * trans^(...)(...y...y...)	46	y^(...)	16
trans^(...)(...y...y...) * trans^(...)(...t...)	47	y	17
trans^(...)(...y...y...) * trans^(...)(...y...)	48	trans^(...)(...) * trans^(...)(...)	226
trans^(...)(...y...y...) * trans^(...)(...)	49	trans^(...)(...) * trans(...)	227
trans^(...)(...t...) * trans^(...)(...t...t...)	50	trans(...) * trans^(...)(...)	228
trans^(...)(...t...) * trans^(...)(...t...y...)	51	trans(...) * trans(...)	229
trans^(...)(...t...) * trans^(...)(...y...t...)	52	trans^(...)(...)	230
trans^(...)(...t...) * trans^(...)(...y...y...)	53	trans(...)	231
trans^(...)(...t...) * trans^(...)(...t...)	54	e^(...)	232
trans^(...)(...t...) * trans^(...)(...y...)	55	e	233
trans^(...)(...t...) * trans^(...)(...)	56	pi^(...)	234
trans^(...)(...y...) * trans^(...)(...t...t...)	57	pi	235
:	:	constant raised to a power	236
:	:	alphabetic constant	237
:	:	numeric constant	238

Notes: (1) t denotes the independent variable, y denotes the dependent variable.

(2) "trans" denotes transcendental and may be any one of "sin", "cos", "tan", "csc", "sec", "cot", "arcsin", "arccos", "arctan", "ln", "exp".

As an example, the terms in the ODE

$$y'' + \sin^{(2)}(m * t * y) * \cos^{(n)}(y) - t + \sin(y) + m = 0$$

where m and n are constants would be assigned the following priorities:

- priority 2 for y''
- priority 34 for sin^(2)(m * t * y) * cos^(n)(y)
- priority 15 for t
- priority 225 for sin(y)
- priority 237 for m

The sorted equation is: $y'' - t + \sin^{(2)}(m * t * y) * \cos^{(n)}(y) + \sin(y) + m = 0$.

The priorities of Table 5.2 provide a unique sort for many equations but does not recognize semantically identical multiplication sequences. The transcendental sequence, "sin^(2)(m * t * y) * cos^(n)(y)", from the preceding example will not be understood to be the same as "cos^(n)(y) * sin^(2)(m * t * y)" even though the priorities from Table 5.2 will sequence them in a correct equation ordering. Also, in certain cases where several

multiplication sequences of transcendental functions occur in an equation, the arbitrary ordering of the terms within each sequence will cause an arbitrary sorting of the equation. For example, the ODE y' + sin^(2)(t) * tan(y) + cos^(2)(m * y) * sin(y) + t = 0 will be sorted to y' + t + sin^(2)(t) * tan(y) + cos^(2)(m * y) * sin(y) = 0 but if the same ODE were entered as y' + tan(y) * sin^(2)(t) + cos^(2)(m * y) * sin(y) + t = 0, the result would be y' + t + cos^(2)(m * y) * sin(y) + tan(y) * sin^(2)(t) = 0.

6. THE ODEASY SYSTEM

6.1 The Implementation System

ODEASY is implemented in Microsoft Excel 97 with Microsoft Visual Basic in a Windows'95 environment. The sorting strategy exploits the properties of the spreadsheet by using its columns as lists containing terms that are to be ordered and takes advantage of the built-in Excel sort function for data to perform the actual sort once the term priorities have been assigned. The Excel sort utility can sort entire columns of cells in ascending or ascending order according to a chosen key column which, in the case of ODEASY, will contain the priority numbers.

In general, the extensive data manipulation and mathematical functions of spreadsheets are spurring them to gain wider usage in science and engineering applications. These range from the solution of a three-dimensional transient heat conduction problem which found the spreadsheet ideal in deriving a numerical solution based upon the finite-difference method [Kharab, 1997] to the representation of neural network grids in the simulation of long-term semantic memory [Russell and Fatmi, 1995]. The spreadsheet provided a natural grid for a neural network grid and was easily translatable into rules, frames, and logical expressions. Other examples include using the spreadsheet to dynamically visualize simulation results [Adamy et al., 1996] and solution of the three dimensional Laplace equation.

For the user interface, Visual Basic offers the ability to dynamically re-size, create, and delete dialogboxes and their objects. This versatility can be utilized to fit dialogboxes to particular conditions, for example, a parameter list for an equation, which varies from one query instance to another. In conjunction, the Excel worksheet can be customized to become an entry form with menus, toolbars, and buttons that are readily programmable for the interface. The worksheet has the added benefit of being able to accept very long entries due to the 240 character capacity of each cell (which is well within the requirements of ODEASY). The majority of query forms, including the Visual Basic dialogboxes, would have difficulty accommodating large inputs such as long differential

equations that the user can easily enter and see at once. In fact, Excel has been found to be an effective substitute for other widely used graphical user interface (GUI) by [Ng and Crockett, 1997] who assessed the package on its GUI potential based upon criteria such as cost, compatibility, flexibility, training, maintenance, and implementation effort. A second study also concluded that Excel, in comparison to other spreadsheet packages (QuattroPro and Lotus 1-2-3), is well-suited for data entry and presentation of solutions in the modeling of linear and nonlinear examples of machine design problems [Kahn-Jetter and Sasser, 1997].

The sorting strategy of ODEASY could have been implemented in C with multiple linked lists or any other high level language with dynamic memory features. Using these alternative languages would necessitate the creation of additional sort routines as well as explicit memory management for controlling the term list structures. The efficiency offered by dynamic memory allocation is an important consideration since languages which permit only static declarations, as is the case with Visual Basic, will inevitably waste memory when a sufficient number of data structures of large enough capacity must be predefined. This would have been the situation with the ODEASY sorting algorithm were it not for the combination of Visual Basic with Excel which dispenses with both dynamic and static memory management issues as the cells of the spreadsheet can serve as contiguous data structures. This is a particularly useful option given that the ODEASY term lists can grow and shrink, and are created and destroyed as the equation becomes sorted.

In the following discussions, just as a list refers to an Excel column, nodes refer to the cells. Adjacent cells may serve as additional fields required by the node data structure. The pliable nature of spreadsheet cells permits this superimposition of data structure although the enforcement of the structure depends entirely on software control. Since the algorithm could be implemented in other systems and languages, the more generic terminology of 'lists' and 'nodes' which is commonly used to describe such methodology is adopted.

6.2 The ODEASY Sorting Strategy

The ODEASY sort extends the term sorting method presented in §5.2 by sorting at a finer level of granularity. The extended sort applies priorities to individual elements, including multiplication elements, in the input ODE. These priorities define a canonical form for the ODE that enforces a unique ordering of the ODE elements irrespective of the sequence of these elements in the input. This canonical form is recognized by the ODEASY database. The result is an equation that can be used to perform a straightforward pattern matching with the equations of the database and in this way, the equations themselves act as the indices to their own solutions. Thus, equations that are semantically identical are sorted precisely to the same pattern regardless of how they are entered.

As an example, consider the following three inputs to ODEASY,

y' – ((y * a) + (b * t))/(tan(y) + (t^(2) * b)) + sin(y + a) * cos(y) + t * (b + 1),

y' + cos(y) * sin(a + y) – (((t * b) + a * y)/(b * t^(2) + tan(y))) + (b + 1) * t,

t * (b + 1) + (sin(y + a) * cos(y)) – ((y * a) + b * t)/(tan(y) + b * t^(2)) + y'.

All three are sorted to become

y' + (b + 1) * t + cos(y) * sin(a + y) – (b * t + a * y)/(b * t^(2) + tan(y)). (1)

In the following discussion, reference will be made to the elements, terms and subterms of the equation. The sort considers the equation as basically a string of tokens to be ordered and the atomic entities are referred to as elements. Elements include constant and variable names, single functions, brackets, and operators. Elements, parenthesized elements, and elements connected by the "*" operator comprise the terms of the equation and terms themselves may contain other terms. Terms contained within other terms are subterms. Some examples of terms include: y, sin(y) * cos(t) * a, (y + t – a * sin(t)/(t^n + a)), and t * (y^(cos^(m)(t) + m) * b). The demarcation point for a term in an equation is the occurrence of a plus or minus sign once all the term's left parentheses, if any, have

been matched with corresponding right parentheses or if the end of the equation is reached.

The inability of the method in §5.2 to identify the equality of identical terms composed of different orderings of subterms points to the need to apply an ordering to the elements of multiplication sequences. This addresses the ambiguity in sorting which result due to complex argument combinations or with longer multiplication sequences in equations comprised of several terms of three or more functions. For example, should an expression such as sin(y) * csc(t) * tan(a * y) + sin(y) * cos(t + y - 1) + sin(y) * csc(t) * tan(b * t) + sin(y) * cos(t – 1 + y) be sequenced as sin(y) * cos(t –1 + y) + sin(y) * cos(t + y -1) + sin(y) * cos(t) * tan(b* t) + sin(y) * cos(t) * tan(a * y) or perhaps some other variation? Expanding the number of categories is impractical due to the proliferation of cases that must be considered with each additional function, element, or argument permutation in a sequence. The approach must sort yet maintain the multiplication operands together during the equation ordering.

ODEASY eliminates the requirement for many of the sorting categories in Table 5.2 concerned with multiplication sequences by providing a separate sort ordering for multiplication. The ODEASY sort priorities, shown in Table 6.1, consists of 34 categories with a separate set of priorities for multiplication (in column 3) and for addition/subtraction (in column 4). These two sets of priorities, now possible due to the separation of processing of multiplication and addition/subtraction, permit operator specific rankings that reflect the conventional format for equations.

The multiplication priorities reflect the standard practice of placing constant coefficients to the left of functions which in turn, are positioned to the left of derivatives, if any. On the other hand, constant coefficients also serve as the right operand in common addition/subtraction (a/s) expressions such as "y – 1" or "t + 2". These constants are positioned accordingly via the Table 6.1 column 4 priorities which assigns the lowest ranking to the numeric constant template. Similarly, while the multiplication priorities ranks a derivative to the rightmost position of a multiplication term, the a/s priorities

ensures that the derivative is sorted to the leftmost position of the a/s sequence. This complies with the common practice of beginning ODEs with the highest derivative.

Table 6.1 ODEASY Sort Priorities

Category No.	Equation Element Template	Multiplication Priority	Addition (Sub'n) Priority
1	y"''	30	1
2	y"''	31	2
3	y'''	32	3
4	y''	33	4
5	y'	34	5
6	transcendental(c)	29	19
7	transcendental(...y ...)	28	18
8	transcendental(...t ...)	27	17
9	transcendental(...y ...y ...)	26	16
10	transcendental(...y ...t ...)	25	15
11	transcendental(...t ...y ...)	24	14
12	transcendental(...t ...t ...)	23	13
13	transcendental^(...)(c)	22	12
14	transcendental^(...)(...y ...)	21	11
15	transcendental^(...)(...t ...)	20	10
16	transcendental^(...)(...y ...y ...)	19	9
17	transcendental^(...)(...y ...t ...)	18	8
18	transcendental^(...)(...t ...y ...)	17	7
19	transcendental^(...)(...t ...t ...)	16	6
20	f(...y ...)	15	23
21	f^(...)(...y ...)	14	22
22	g(...t ...)	13	21
23	g^(...)(...t ...)	12	20
24	y^(...n ...)	10	26
25	y	11	27
26	t^(...n ...)	8	24
27	t	9	25
28	e^(...)	6	28
29	e	7	29
30	pi^(...)	4	30
31	pi	5	31
32	numeric constant	1	34
33	c^(...)	2	32
34	c	3	33

Notes: (1) t denotes the independent variable, y denotes the dependent variable.

(2) A transcendental and may be any one of "sin", "cos", "tan", "csc", "sec", "cot", "arcsin", "arccos", "arctan", "ln", "exp".

(3) c refers to any constant.

Using the operator specific priorities of Table 6.1, ODEASY applies a local ascending sort on the terms based on the operators of multiplication and addition/subtraction. The sort continues to be resolved alphabetically in the event that two or more terms are tied in their priorities. As with Table 5.2, the equation element templates are traversed from top to bottom until a match is found with the term to be sorted. This comparison is considerably faster than that of Table 5.2 since there are at most 21 cases to consider for each operator type. The algorithm optimizes the search by skipping categories 7 to19 once it determines that an element is not a transcendental function so that no time is wasted in checking again to determine argument characteristics. If it is a transcendental, categories 7 to 12 are skipped if a peek ahead on the element reveals that it is raised to a power. The worst case scenario for a search occurs for alphabetic constants which are compared against categories 1 to 6, then 20 to 34, or a total of 21 cases per operator.

In processing an equation, ODEASY first refers to column 3 of Table 6.1 to perform a sort on all consecutive multiplication terms and ignores the a/s elements in this initial sorting phase. For example, terms such as

"y'' * sin(y) * cos(y) * tan(t) * y" and "t^(2) * 3 * exp(y)"

are re-arranged to

"y * tan(t) * cos(y) * sin(y) * y'"" and "3 * t^(2) * exp(y)".

In the first term, the element "y" is sorted to the beginning of the sequence via its highest rank (or lowest priority number) followed by the transcendental function of the independent variable, "tan(t)". The next two transcendentals, both of priority number 28, have their precedence resolved through a further alphabetical sort that places "cos(y)" ahead of "sin(y)". The rightmost position in this sequence belongs to the second derivative with its priority of 31. In the second term, the numerical coefficient, 3, with a priority number of 1 is moved to the leftmost position and the priorities of the remaining subterms leaves their relative positions unchanged.

Once the multiplication terms are sorted, ODEASY performs the second sort phase where multiplication sequences are treated as entities for addition/subtraction, and the column 4 priorities are applied to the a/s terms to complete the ordering of the ODE. As before, the sort is in ascending order where elements of lower priorities are placed to the left of those with higher priorities. For example, if the two terms indicated earlier in the multiplication sort, "y * tan(t) * cos(y) * sin(y) * y''" and "3 * t^(2) * exp(y)", were addition terms comprising a complete ODE, then this phase will place them in ascending order relative to each other. Since the derivative term is associated with a priority number of 4 while the other term a priority number of 23, the result is the sorted equation "y * tan(t) * cos(y) * sin(y) * y'' + 3 * t^(2) * exp(y) = 0".

This two-phase algorithm accommodates equation complexity by methodically organizing the subterms of every term and then organizing the terms within the equation. By working at a finer level of granularity, this process can remove extraneous brackets and treat all elements within internal parentheses of a term in the same manner as the term itself. For example, if a term is entered as tan(ln(y * t) * cos(y * t)), the algorithm re-arranges the innermost arguments to become "t * y", then the tan function argument to become "cos(t * y) * ln(t * y)" and finally, generates the resulting sequence tan(cos(t * y) * ln(t * y)). At the conclusion of the second phase, each term detail has been considered and all terms have been put in their proper place in the equation according to their priorities.

6.2.1 The ODEASY Sort Algorithm

The ODEASY sort algorithm makes two passes over the input equation, one for each of the operator based sort phase. As indicated in §6.2, multiplication is considered in the first phase and addition/subtraction in the second phase. In each phase, Table 6.1 is accessed in sequential order from top to bottom until a match is found. Thus, if a match is not made until case 28 or higher, then the term is a constant because any state variable multiplication factors or state variable exponents would have caused the term to be successfully matched in an earlier case.

The operators of division and exponentiation are ignored since their operands are not interchangeable without impacting the semantics of the equation and thus, any sorting with respect to these operators applies only to the ordering of elements within parenthesized operands (numerators and denominators, or exponent base and exponent). For the other 3 operators, the rules of associativity and commutativity are upheld during term simplification and sorting. These two rules for multiplication and addition [Durbin, 1985] are as follows:

- $a * b = b * a$ Commutative law for multiplication
- $a * (b * c) = (a * b) * c$ Associative law for multiplication
- $a + b = b + a$ Commutative law for addition
- $a + (b + c) = (a + b) + c$ Associative law for addition

Given these rules, the operands of the "+" and "*" operators may be re-ordered. Furthermore, according to the associative law, matching parentheses may be removed from consecutive multiplication elements and from consecutive addition elements. ODEASY takes advantage of this by stripping all extraneous brackets in order to eliminate variability in the input; for example, a term entered as "a * (b * c)" or "(a * b) * c" will be simplified to "a * b * c". Reducing all unnecessary brackets helps to ensure consistency in the representation of the equation as a key in the symbolic database.

The subtraction operator is neither commutative nor associative but it is possible to re-order a series of subtraction operands that are preceded by "-", i.e., it is equivalent to sorting a series of negative terms. For example, a subtraction sequence such as "- y - sin(t) - t" will be sorted to "- t - y - sin(t)" based upon the Table 6.1 column 4 priorities. However, if the sequence had been "y - sin(t) - t", then a re-ordering would require that each term, including the positive "y" element with a "+" operator, be explicitly associated with its left operator during the sort. The resulting sequence " - t + y - sin(t)" which, while correct, is an awkward representation for the expression since it is generally accepted that subtraction expressions begin with the positive element, if there is one. Thus, even though it is contrary to the rankings of Table 6.1, switching the position of the

first and second element would produce the more natural expression "y - t - sin(t)". The approach taken by ODEASY is to link each operand with its left operator during the a/s sort phase and to not begin a sorted expression with a negative element (unless all elements in the sequence are negative). Instead, to conform to convention, the algorithm will swap any negative term that begins a sorted sequence with the first positive term in the sequence. In such cases, except for the first term, all terms in the expression continue to follow the rankings of Table 6.1.

Note that the method of associating the left operator with operands only applies to the a/s sort phase, is unnecessary in the multiplication phase, and cannot be used to combine the two phases into one by tracking the operators. Otherwise, violation of the associative property of multiplication could result since expressions such "t * ln(t) + y" would erroneously sort to "t + y * ln(t)". It is neither simple nor efficient to merge the purposes of the two phases and initial attempts at term sorting had in fact, been devoted to developing an approach that considered all the operators during a sorting process which ultimately required three convoluted passes over the equation.

The sorting strategy cannot, as of yet, process for the distributive law which states the following equalities:

- $a * (b + c) = a * b + a * c$
- $(b + c) * a = b * a + c * a$
- $a * (b - c) = a * b - a * c$
- $(b - c) * a = b * a - c * a$

Part of the problem is that, outside of the fair amount of additional processing required, it is not clear whether a term entered in the format of the LHS of the equalities should be converted to the format of the RHS, or vice-versa. The LHS, being more compact, might be more practical and hence, likelier as the entry format. If this is correct, then future work on ODEASY could seek out sequences resembling the format of the RHS and process them into the format of the equivalent LHS. Currently, although there is no comprehension by the system of the equalities stated above, an entry in the LHS format of

either the first or second expression would be sorted to the same structure by the commutative law. That is, whether an expression is entered as "a * (b + c)" or "(b + c) * a", the sorted outcome is "a * (b + c)" assuming that a, b, c are constant variables (which all receive the same priority with the precedence relationship resolved alphabetically). Likewise, under the same assumption, the LHS of the next two equality would sort to "a * (b – c)"; the RHS of the first and second equality would be ordered as "a * b + a * c", and the RHS of the third and fourth equality would also sort to an identical structure. This, however, does not compensate for a lack of understanding of the distributive law itself. The incorporation of this rule could possibly be built into a pre-sort phase to distribute elements to the same format and should comprise one of the topics to explore in future enhancements.

6.2.1.1 The Multiplication Sort Phase

This phase parses the ODE from left to right and extracts the elements into multiplication lists and non-multiplication lists. Each list is considered to occupy a level and from level n, the creation of a new list occurs at level n+1. In general, consecutive elements are entered in nodes that are appended to a list until there is a change in sign from '+' or '-' to '*' or vice-versa, from '*' to either '+' or '-'. In the former case, the sign change causes a multiplication list to begin at level n+1 that will be appended with nodes of consecutive multiplication terms until the next sign change, the second case, at which point the multiplication list becomes sorted. The sort is based upon the priorities assigned to each node term according to the templates in Table 6.1 as described in §6.2. The node structure in the multiplication phase consists of two fields as follows:

TERM PRIORITY	Term

Once re-ordered, the level n+1 list is represented by one sorted expression and the level is 'collapsed' back to level n with the sorted term placed at the end of the level n list where it is treated as an addition (or subtraction) term. Where the sort occurs for a level n=1 list, the result is placed in the first node and the rest of the list is cleared. The processing

continues with the next term, which is the right operand of the sign change, being appended as a new node of the level n list.

Besides a change in sign, list creation and deletion also occurs for parenthesized terms. A new level begins each time a left parenthesis is encountered and a level is removed, or collapsed, with each right parenthesis. If the left and right brackets are equal in a term, it is said to be a 'complete' subterm. On the other hand, if the number of right brackets exceeds the number of left brackets, the term is referred to as a 'closing' subterm. The expression may be partitioned among the nodes of the level and the entire level may be referred to as having a complete or closing subterm.

Complete and closing subterms in a non-multiplication level n, n>1, cause the level to collapse to level n-1. Such a collapse involves combining the node terms back to their original expression and placing it at the end of the level n-1 list.

A complete or closing subterm in a multiplication level n, n>1, cause a sort followed by a collapse to level n-1 except in the instance where brackets are stripped and the next operator is "*" or level n-1 is a multiplication list. In processing the multiplication level, superfluous brackets are stripped prior to the sort to ensure that equivalent expressions with and without brackets are represented in the same format. Excess parentheses entered by the user to visually group the logical order of expressions has no impact on the semantics of the equation and aesthetics in this case is supplanted by the requirement to generate a rigidly ordered generic representation of the ODE. By isolating parenthesized terms into one level, unnecessary brackets are easily identified by left bracket(s) in the first node and corresponding right bracket(s) in the last node. If the next operator is not "*" and the prior level is not a multiplication list, or if the list continues to be parenthesized, (i.e., complete bracket reduction is not possible), then the multiplication level is sorted and collapsed as before. For the case of complete bracket removal in level n, the level n terms are transferred to level n-1 if level n-1 is also a multiplication level. If not, but the next operation continues to be multiplication, then the bracket removal permits the current multiplication level to continue accepting additional terms.

Since parentheses lead to the combine of a non-multiplication list and a sort of a multiplication list, they are added before the sort by the Parser to enclose the LHS of the input ODE. (All nonzero RHS elements have been moved to the LHS of the ODE by the Parser before the enclosing brackets on the LHS are added). This outside set of brackets on the ODE serve as a trigger to either combine or sort the nodes of the last list after the final term has been entered. Taking advantage of the existing parentheses processing in this manner eliminates the extra step that would have been required to explicitly acknowledge and force the last term into a combine or sort operation.

To implement the multiplication sort phase, four status flags are required per level as follows:

1. MultLvl(level n) – A boolean indicator which is set to True if level n contains a multiplication list and is False otherwise.

2. ApLvl(level n) – When a level n+1 expression is collapsed to the end of level n, the expression is either appended with the last non-empty node or written in an appended node of the list. This distinction is necessary since arguments to functions, subterms with nested left brackets, exponents of exponentiations, and denominators of the division operation must be included with their functions, left brackets, exponent base, and numerator terms, respectively, prior to sorting. Hence, such expressions are concatenated to the function name, left bracket, exponent base, or numerator that is in the last node of the level n list. For example, a collapsed level n+1 expression of "(x * y)" representing the argument to a level n function, say, "exp()", must be concatenated to become "exp(x * y)" for a proper sort. In such cases which include a level n last element in the following set ["(", "/", "^", "f", "g", and transcendental functions], the boolean flag of ApLvl(level n) is set to True to indicate that an append of the level n+1 term with the last element of level n is required. For all other cases, ApLvl(level n) is set to False to indicate that a new level n node is to be added for the collapsed term which must be sorted as a separate entity of level n's grouping.

3. LbracLvl(level n) – A counter of the number of left parentheses at level n.

4. RbracLvl(level n) – A counter of the number of right parentheses at level n. This, together with LbracLvl indicates whether the level n subterm is a complete or closing subterm.

At the start of each new level, the default settings for the boolean indicators is False and for the bracket counts is zero.

The general scheme and rules for the multiplication phase follows:

1. A list node entry, described in extended Backus-Naur Form [Louden, 1993], is of the form {"("} [<expression>] as defined by the following recursive productions:

<expression> ::= <term> | <term> "+" {<term>} | <term> "-" {<term>} | <term> "/" {<term>} | <term> "^" {<term>} | <term> "*" {<term>}

<term> ::= "(" <expression> ")" | <expression> | <subterm> {")"}

The subterm is the terminal structure and may be any one of the template forms listed in Table 6.1. A restriction on this grammar is that only the top list node may contain one or more left parentheses that are not followed by an expression. This case arises in the parsing of bracket nested terms where a level promotes a new level with each successive left parenthesis (refer to Rule 3).

During the build-up of a list, the node content terminates with the operator whenever the term is the left operand of addition or subtraction. The occurrence of either of these operators causes a new node to be appended to the list that accepts the right operand. For example, the parsing of the expression "(y' + t)" will result in a two node list with "(y' +" in the first node and "t)" in the second.

The division and exponentiation operators have no impact on list formation but are simply appended to their left operand during list construction. The right operand may be promoted to a new list if it is parenthesized but are otherwise immediately concatenated with the their division or exponentiation operator.

As for the multiplication operator, it does not appear unless an expression has been sorted. The parsing places multiplication operands, one to a node, in a list of their own eliminating the need to specify the operator explicitly and hence, restricting these operand types to node entries of the form {"("} <expression> {")"}. Once a multiplication list is sorted, the operator is inserted to regenerate the expression. Thus, a node may terminate with a parenthesis, (a left bracket for the top node and a right bracket for all other nodes), or an operator from the set ["+", "-", "/", "^"] but never with a "*".

2. If "*" is the first operator in the ODE, then the first level contains a multiplication list and MultLvl(1) is set to True. For n>1, a level n multiplication list arises from one of two causes. The first is a parenthesized multiplication term which is automatically promoted to level n from level n-1 due to the occurrence of the left parenthesis (refer to Rule 3). The second cause is an addition or subtraction operation that is followed by a multiplication operation.

   Specifically, when the right addition/subtraction operand is the left operand of multiplication, the operand is promoted from its level n-1 non-multiplication list to the first node of a new multiplication list in level n. A second node is created to accept the next multiplication operand. The subsequent sorting is valid given that multiplication has precedence over addition and subtraction. For example, the ODE "y' + t * b" is processed as follows:

   Level 1

   y +

   t

   The next operator of * promotes "t" to a new multiplication level:

   Level 1 Level 2 (MultLvl(2)=T)

   y + t

   The next multiplication operand, "b", is entered in a new node in Level 2:

Level 1	Level 2
y +	t
	b

Level 2 is then sorted to "b * t" which is collapsed down and appended to the first node of Level 1 to form "y' + b * t". MultLvl(2) is reset to False.

3. The occurrence of a left parenthesis in a non-empty level n starts a new level n+1 with the left parenthesis placed in the first node of the level n+1 list. If the rightmost element in the last node of level n is the name of a transcendental or state variable function, an exponent base, a numerator, or a left bracket (as part of a nested term), ApLvl(n) is set to True. The LbracLvl(n+1) count is set to 1.

4. In a non-multiplication level n, n>=1, the occurrence of a closing or complete subterm causes the list nodes to be combined back to their original expression which is then inserted in the first node of the level n list. The first node also receives the highest priority of the combined nodes. The combine operation connects the node terms together in the order that they were entered. All nodes except the first are deleted.

5. For a multiplication level n, n>=1, the addition of a right parenthesis in the last node updates the RbracLvl(n) count. If the number of left and right parentheses now balance for the level then it is possible that they are superfluous and could be removed. An equal number of left brackets is removed from the top node as the number of right brackets from the last node of the list provided the parenthesized expression is not any one of the following:

    - A numerator or denominator term since the removal of brackets alters the semantics of the expression. For example, the term "(t + y)/(m + 1)" cannot be simplified to "t + y/m + 1".

    - An exponent base or exponent for the same reason as the division operation.

- An argument to a transcendental or state variable function as indicated by a True value in ApLvl(n-1). This is valid only for n>1 since parenthesized arguments never begin in level 1. Note that a True value in ApLvl(n-1) may also refer to a bracket nested term whose brackets may be reduced.

Once extraneous leading and trailing parentheses are removed, the multiplication list is always sorted if the last node remains a closing subterm since appending additional terms following a node with a right parenthesis risk altering the meaning of the expression by the sort. For example, a list may contain an expression such as "y * a)" with "y" in the first node and the closing subterm "a)" in the second node. (Such an expression could be part of a function argument that began in the previous level). Adding an element, say "t", to a third node, would result in a sorted expression of "a * t * y)" or "a * t) * y" depending on how the parenthesis is managed. Both cases erroneously change the meaning of the parenthesized term.

However, if the last node of the list contains a complete subterm, or one without parentheses due to bracket removal, then two other conditions are considered with the sort proceeding if neither condition is met. The first condition, which applies to level n>1 is satisfied when level n-1 is a multiplication level. If it is, level n is collapsed to the end of the level n-1 list so that a complete sorting of consecutive multiplication terms can take place. The level n nodes are transferred in the sequence they occur to the tail of the level n-1 list. If ApLvl(n-1)=True, the transfer begins with the concatenation of the term in the first node of the level n list to the end of the term in the last non-empty node of the level n-1 list. This situation arises when the level n-1 list contains "(" as its only element and the level n list was its nested term. Following the transfer, the ApLvl(n-1) is reset. If ApLvl(n-1) had been False, the transfer begins at a new node of level n-1. For example, "y * (t * b)" is processed as follows:

Level n	Level n+1
y	(t
	b)

The bracket reduction leaves:

Level n	Level n+1
y	t
	b

The collapse transfers the level n+1 terms to level n:

Level n
y
t
b

Within the same level, the expression can be sorted as "b * t * y" according to the priorities of Table 6.1.

The other condition, which applies to any multiplication level n, n>=1, is satisfied when the next operator is "*". If it is, then the sort is not performed but processing continues in the normal build-up of the level n list with the addition of a new node for the next multiplication operand.

6. Every sort is preceded by an attempt to remove superfluous brackets for terms with balanced parentheses. When a sort, which re-inserts the "*" operator between the terms to form the result, is performed on multiplication level n, the result is placed in the first node of the level n list and the remaining nodes are deleted. The priority associated with the sorted expression is updated with the largest priority of the elements comprising the expression and lastly, the MultLvl(n) flag is reset.

7. If a complete or closing subterm is in the first and only node of level n, n>1, which arises either from a single element parenthesized term or a sort/combine operation, then the expression is collapsed to the end of the level n-1 list. If ApLvl(n-1)=True, the collapse is a concatenation with the contents of the last non-emtpy node, otherwise, the subterm is placed in a new node at the end of the level n-1 list. A

collapse into a new node updates the node's priority with the subterm's priority. A collapse into a non-blank node provides the node with a priority based upon the new expression composed of the collapsed term with the node's existing elements. The node receives the highest priority of the terms comprising its new expression. The LbracLvl(n-1) and RbracLvl(n-1) counts are updated accordingly and ApLvl(n-1), LbracLvl(n), RbracLvl(n) are reset. If n=1, the level is not collapsible and only the RbracLvl(1) is updated.

8. The collapse of a term from level n+1 to level n will trigger either a sort or combine operation (depending upon the type of level) if it updates the level n bracket counts such that the last node of the level is a closing subterm. The resulting expression may in turn force a sort or combine and a collapse of level n to level n-1 and in this manner, successive collapses can occur until n=1. A collapse that results in a complete subterm in the first node of level n is brought about when level n consists of a function name or left parenthesis and level n+1 holds the function argument or nested subterm. In this case, level n cannot be a multiplication level and neither a sort nor a combine is required following the collapse. For n>1, successive collapses may also occur in this way until n=1. A collapse to level n that results in a complete subterm in the last node, for a level n with two or more nodes, does not cause a sort or combine and hence, will not immediately lead to a further collapse. This non-first node complete subterm can be treated as one entity for subsequent sorting or combining. In this case, processing continues at the last node of level n.

9. A change in sign (to "+" or "-") in a multiplication level n triggers a sort on the level. No collapse occurs for a sign change induced sort since there are no parentheses to deal with. The addition/ subtraction sign is appended to the sorted result and a new blank node is appended to the list.

10. For a non-multiplication level, a "+" or "-" operator extracted from the input equation is appended to the term in the current level n node and causes a new blank node to be appended to the list.

11. A "*" operator is never appended to a list operand. For an existing multiplication level, it causes a new blank node (that will accept the next operand) to be appended to the list. For a non-multiplication level n, n>=1, one of two effects occur. If the level n list consists of only one node and the node's term does not end in an operator, then MultLvl(n) is set to True and level n becomes a multiplication level. A new blank node is appended to the list. This case occurs if the initial operation on a level is multiplication or when a parenthesized term, possibly the argument of a function, is collapsed to a non-multiplication level that now contains the complete left operand of a multiplication operation and hence, the MultLvl(n) flag is updated. The other possible effect of a "*" operator on a non-multiplication level n is to promote the term in the last node of the level n list to the first node of a new level n+1 list which becomes a multiplication list. This is the situation indicated in Rule 2 where a right operand of addition or subtraction is also the left operand of a following multiplication operation. Since the latter operation takes precedence and its operands may be sorted within a multiplication expression without impacting the equation's meaning, the operand is moved to participate in a new multiplication level.

The multiplication phase is complete once all LHS elements of the input ODE have been extracted and the resulting expression is at the first node of the level 1 list. An example ODE, shown below, will be used to illustrate the multiplication and addition/subtraction sort phases. The equation is represented in the format prepared by the ODEASY Parser which has moved all non-zero elements to the LHS and enclosed the LHS in brackets. The equation is:

ODE (1): (y' * y^(1 + n) + y'' – (((y – b) + a)/((t + b) * (a – b))) + (y/n) * (b + (y – n)^(t)))

A second example with the equation:

ODE (2): (y' + y * (t^(1 + a) + b) + ((y * t^(a)) + (t * b))/((t + b) – a)^(2) + 1/(1 – sin(y)) * exp(y))

is presented in Appendix I.

The application of the multiplication rules is shown next. The result of this phase is the starting point for the addition/subtraction phase where the example is continued following a discussion of the rules for the second phase.

In the example, the MultLvl and Aplvl flags are shown above each column level and the values True and False are represented as T and F, respectively. The sort priority field, with default of zero, is provided on the left of each term. The applicable rule numbers are indicated by "R #" in the comments at the base of each level. A new empty node that is created and pointed to is indicated by "<=".

ODE (1):

(y' * y^(1 + n) + y'' – (((y – b) + a)/((t + b) * (a – b))) + (y/n) * (b + (y – n)^(t)))

MultLvl(1)=T, ApLvl(1)=F	MultLvl(1)=T, ApLvl(1)=T	MultLvl(2)=F, ApLvl(2)=F
0 (y' 0 <=	0 (y' 0 y^	0 (1 + 0 n)
The first "*" changes level 1 to a multiplication level since the term occupies the first node of the list and the element is not trailed by a "+" or "-" (R 2). A new node is created due to the "*" (R 11).	The "(" for the exponent argument begins a new level 2 list and is inserted in its first node. The last node of level 1 contains the exponent base (R 1) and hence, ApLvl(1) is set T (R 3).	Since level 2 is not a mult. level, the "+" is added to node 1 (R 1) and a new node 2 is appended (R 10). When node 2 receives the ")", level 2 has a complete subterm which causes a combine operation that places the expression in node 1 (R 4).
MultLvl(2)=F, ApLvl(2)=F	MultLvl(1)=T, ApLvl(1)=F	MultLvl(1)=F, ApLvl(1)=F
0 (1 + n)	0 (y' 0 y^(1 + n)	34 (y^(1 + n) * y' + 0 y'' - 0 <=

The complete subterm in node 1 in level 2 (>1) causes a collapse to level 1. Since ApLvl(1)=T, the collapse is to the last non-empty node and ApLvl(1) is reset (R 7).	The collapse of the complete subterm to node 2 does not cause an immediate sort (R 8). The next "+" triggers a sort with the result placed in node 1 (R 9). Priority is updated with the largest priority of the elements (34 for y') and MultLvl(1) is reset (R 6).	Following the sort, the "+" is concatenated to the result and node 2 is created (R 9). The next operand-operator pair is inserted in node 2 (R 1) and a node 3 is created (R 10).
MultLvl(2)=F, ApLvl(2)=T	MultLvl(3)=F, ApLvl(3)=T	MultLvl(4)=F, ApLvl(4)=F
0 (	0 (	0 (y - 0 b)
The "(" begins the level 2 list (R 3). The next "(" begins the level 3 list and sets ApLvl(2) to T since it is a nested bracket of level 2 (R 3).	The 3rd "(" begins the level 4 list and sets ApLvl(3) to T since it is a nested bracket of level 3 (R 3).	The "-" causes a new node 2 to be appended (R 10). When node 2 receives the ")", level 4 contains a complete subterm that will be combined and put in node 1 (R 4).
MultLvl(4)=F, ApLvl(4)=F	MultLvl(3)=F, ApLvl(3)=F	MultLvl(3)=F, ApLvl(3)=F
0 (y – b) 	0 ((y – b) 	0 ((y – b) + 0 a)
The complete subterm in the first node triggers a collapse to level 3 (R 7).	The collapse to level 3 is a concatenation with node 1 since ApLvl(3) = T and ApLvl (3) is reset (R 7). No further collapse occurs since it is neither a complete nor closing subterm.	It is not a mult. level so the next "+" is added to node 1 and creates node 2 (R 10). A complete subterm exists when node 2 receives ")". This triggers a combine operation (R 7).
MultLvl(3)=F, ApLvl(3)=F	MultLvl(2)=F, ApLvl(2)=T	MultLvl(3)=F, ApLvl(3)=T
0 ((y – b) + a)	0 (((y – b) + a)/	0 (

The complete subterm in the first node triggers a collapse to level 2 (R 7).	The collapse to level 2 is a concatenation with node 1 since ApLvl(2) = T and ApLvl(2) is reset (R 7). The "/" sign is appended to the term (R 1).	The "(" of the denominator triggers a new level 3 and sets ApLvl(2) to T (R 3). The next "(" triggers a new level 4 and sets ApLvl(3) to T since it is a nested subterm of level 3 (R 3).
MultLvl(4)=F, ApLvl(4)=F	MultLvl(3)=F, ApLvl(3)=F	MultLvl(3)=T, ApLvl(3)=F
0 (t + 0 b)	0 ((t + b)	0 ((t + b) 0 <=
It is not a mult. level, so the "+" is concatenated to node 1 and node 2 is created (R 10). The ")" in node 2 creates a complete subterm for level 4. This triggers a combine followed by a collapse to level 3 (R 4, 7).	The collapse is a concatenation with node 1 since ApLvl(3)=T. ApLvl(3) is reset. No further collapse since node 1 expression is not a complete or closing subterm (R 8).	The "*" changes level 3 to a mult. level since the list contains only one node and it does not end in an operator. MultLvl(3) is set T and a blank node 2 is created (R 11). The next "(" begins a new level 4 (R 3).
MultLvl(4)=F, ApLvl(4)=F	MultLvl(3)=T, ApLvl(3)=F	MultLvl(3)=T, ApLvl(3)=F
0 (a - 0 b)	0 ((t + b) 0 (a – b)	0 ((t + b) 0 (a – b))
The "-" is added to node 1 since its not a mult. level and a new node 2 is created (R 10). The ")" triggers a combine (R 4) and collapse of the complete subterm to level 3 (R 7).	The collapse is to the blank node 2 since ApLvl(3)= F (R 7). The collapse of this complete subterm to a non-first node does not cause a sort (R 8).	The addition of ")" in node 2 creates a complete subterm in mult. level 3. Brackets cannot be reduced since the expression is a denominator term (ApLvl(2)=T). The list is sorted and the result placed in node 1 (R 5, 6).
MultLvl(3)=F, ApLvl(3)=F	MultLvl(2)=F, ApLvl(2)=F	MultLvl(1)=F, ApLvl(1)=F
9 ((a – b) * (t + b))	11 (((y – b) + a)/((a – b) * (t + b)))	34 (y^(1 + n) * y' + 0 y'' - 11 (((y – b) + a)/((a – b) * (t + b))) + 0 <=

The priority of the sorted subterm is updated and the MultLvl(3) flag is reset (R 6). Since node 1 contains a complete subterm, the level is collapsed to level 2 (R 8).	The collapse is a concatenation in node 1 since ApLvl(2)=T. ApLvl(2) is reset. Node 1 is assigned a priority based upon its new expression (R 7). The next ")" changes expression into a complete subterm and triggers collapse to level 1 (R 7).	The collapse is to node 3 since ApLvl(1)=F and updates the node 3 priority (R 7). Since the collapsed complete subterm is not in node 1, it does not trigger a combine (R 8). The next "+" is added to node 3 and node 4 is created (R 10).
MultLvl(2)=F, ApLvl(2)=F	MultLvl(1)=F, ApLvl(1)=F	MultLvl(2)=T, ApLvl(2)=F
0 (y/n)	34 (y^(1 + n) * y' + 0 y'' - 11 (((y – b) + a)/((a – b) * (t + b))) + 0 (y/n)	0 (y/n) 0 <=
The next "(" begins a new level 2 (R 3) which is collapsed back to level 1 when the ")" is received forming a complete subterm (R 4).	Collapse of level 2 is to node 4 since ApLvl(1)=F (R 7). Level 1 is not suitable as a mult. level due to the non-mult. terms. The next "*" promotes the node 4 term (left operand of "*") to a new mult. level 2 (R 11).	MultLvl(2) is set T and node 2 is created (R 2). The "(" begins level 3 (R 3).
MultLvl(3)=F, ApLvl(3)=F	MultLvl(4)=F, ApLvl(4)=F	MultLvl(3)=F, ApLvl(3)=T
0 (b + 0 <=	0 (y - 0 n)	0 (b + 0 (y - n)^
Level 3 is a non-mult level, so "+" is appended to the node 1 term and creates node 2 (R 10). The next "(" begins level 4 (R 3).	The "-" is added to node 1 and creates node 2 (R 10). The ")" of node 2 triggers a combine and collapse to level 3 (R 4).	Collapses to node 2 since ApLvl(3)=F (R 7). The collapse is a complete subterm to a non-first node and hence, does not trigger a combine (R 8). The "(" of the exponent argument starts a new level 4 and sets ApLvl(3) to T (R 3).

MultLvl(4)=F, ApLvl(4)=F	MultLvl(3)=F, ApLvl(3)=F	MultLvl(2)=T, ApLvl(2)=F
0 (t)	0 (b + 0 (y - n)^(t))	0 (y/n) 0 (b + (y - n)^(t)))
Level 4 is collapsed back when the ")" is received forming a complete subterm (R 7).	Collapses to non-blank node 2 and resets ApLvl(3) (R 7). The complete subterm in node 2 does not trigger a combine (R 8). The next ")" creates a closing subterm in node 2 causing a combine (R 4). Resulting complete subterm collapses (R 7).	Collapse of the level 3 subterm is to node 2 since ApLvl(2)=F (R 7). The complete subterm does not trigger a sort (R 8). The next ")" creates a closing subterm in node 2 causing a sort (R 8). Since brackets do not balance, no reduction done (R 5).
MultLvl(2)=F, ApLvl(2)=F	MultLvl(1)=F, ApLvl(1)=F	MultLvl(1)=F, ApLvl(1)=F
11 (b + (y - n)^(t)) * (y/n))	34 (y^(1 + n) * y' + 0 y'' - 11 (((y – b) + a)/((a – b) * (t + b))) + 11 (b + (y - n)^(t)) * (y/n))	(y^(1 + n) * y' + y'' - (((y – b) + a)/((a – b) * (t + b))) + (b + (y - n)^(t)) * (y/n))
MultLvl(2) is reset and the node 1 priority is updated (R 6). The closing subterm is collapsed to level 1 (R 7).	The collapse of the closing subterm from level 2 triggers a combine on level 1 (R 8).	This last combine completes the multiplication phase. The node term priorities are not used since it is a combine operation.

The multiplication phase changes the equation

ODE (1):

(y' * y^(1 + n) + y'' – (((y – b) + a)/((t + b) * (a – b))) + (y/n) * (b + (y – n)^(t)))

into the following by re-ordering the multiplication terms according to the Table 6.1 priorities and removing superfluous parentheses enclosing multiplication expressions:

ODE (1'):

(y^(1 + n) * y' + y'' - (((y – b) + a)/((a – b) * (t + b))) + (b + (y - n)^(t)) * (y/n))

6.2.1.2 The Addition/Subtraction Sort Phase

This phase takes the result of the multiplication phase and parses the equation from left to right to form addition/subtraction (a/s) lists. As with the first phase, each list occupies a level and from level n, the creation of a new list occurs at level n+1. Consecutive a/s terms and their left operators are extracted into nodes that are appended to a list until the occurrence of a closing parenthesis or an opening one. In the former case, the list is sorted according to the priorities of Table 6.1 and in the latter case, a new level is begun for the parenthesized term. Unlike the multiplication phase, a change in sign does not induce the creation and deletion of lists. Instead, all terms are ultimately addition or subtraction operations and only these expressions are considered while all other operators are appended with the operands in each node. The node structure in the a/s phase consists of three fields as follows:

TERM PRIORITY	Term Sign	Term

The term sign field holds either the "+" or "-" operator associated with the term. This sign field isolates the operators from the terms so that a sort of terms with the same priority can be resolved alphabetically. Otherwise, all terms beginning with the "-" operator would automatically be ordered ahead of the same priority addition terms. The sign field also serves to remember the sign of parenthesized terms that are promoted to new levels.

In the sorting of an expression with both negative and positive elements, a negative term is never ordered to the leftmost position even if its priority is the lowest among the priorities of terms in the expression. This reflects the common practice of not beginning expressions with a negative element; for example, a common term such as "t – 1" is not usually represented as "- 1 + t". ODEASY sets the second position of expressions as the leftmost position for a negative term to occupy unless all terms are negative. Whenever

possible, the leftmost position is reserved for the lowest priority addition term and the remainder of the equation terms are sorted according to their template priorities as before.

Similar to the multiplication phase, the sort is preceded by the reduction of redundant parentheses from complete subterms of a/s. However, in addition to not removing the innermost enclosing brackets from (i) function arguments and (ii) operands of division or exponentiation, the innermost enclosing brackets are not stripped from negative parenthesized terms or from parenthesized terms that serve as multiplication operands. To remove these parentheses is to change the meaning of the equation since there is no processing for the distributive law in ODEASY.

As for superfluous brackets in other operations, the case of parentheses enclosing multiplication terms has already been dealt with in the multiplication phase. With division and exponentiation, the immediate parentheses enclosing their operands are not removed but excess brackets of these operands as well as those enclosing the entire operation itself are stripped. The multiplication phase focussed solely on multiplication expressions and did not treat the excess brackets of other operations. If it did, the phase would have had to consider the lack of the distributive law for other operations as well as process non-multiplication lists. To economize on code and processing, the removal of superfluous parentheses for division and exponentiation is performed in the second phase where all remaining operations are considered.

The sorting of an a/s level n list, n>1, produces an ordered expression that is collapsed to the end of the level n-1 list. If n=1, the result is placed in the first node of the list.

To implement the a/s phase, three status flags are required: ApLvl(n), LbracLvl(n), and RbracLvl(n). They serve the same purpose as in the multiplication phase. A phase specific boolean flag, Neg_Mult, is needed to track whether a parenthesized expression in a sort is a subtraction term or a multiplication operand. A True value in Neg_Mult prevents inappropriate bracket reduction and causes the term to be treated as one entity during sorting.

The general scheme and rules for the a/s phase follows:

1. List node entries are of the form {"("} [expression] and is defined by the same grammar as in the multiplication phase (Rule 1). As in the first phase, only the top list node may contain one or more left parentheses that is not followed by an expression.

   The addition or subtraction operator is written to the node's sign field during the parsing of the equation. The sign field for the first term of the equation is assigned the addition operator unless the equation is negative. As in the multiplication phase, the occurrence of an addition or subtraction operator triggers a new node to be appended to the list that accepts the right operand. Once a list is sorted, the a/s operators are re-inserted with the node terms to regenerate the expression and hence, these operators do not appear in a node expression unless the expression has been sorted. The remaining operators of multiplication, division, and exponentiation, are included with the left operand during list construction. Their right operand may be promoted to a new list if it is parenthesized but are otherwise immediately concatenated with the operator. When a non-parenthesized non-constant operand of multiplication, division, or exponentiation is appended in an existing (non-empty) node, the priority associated with the operand replaces the node's priority if it exceeds the node's priority.

   Thus, an a/s list node may terminate with a parenthesis, (a left bracket for the top node and a right bracket for all nodes), or an operator from the set ["*", "/", "^"] but may never end with "+" or "-".

2. The occurrence of a left parenthesis at a non-empty level n causes a new level n+1 to begin with the left parenthesis starting the first node of the level n+1 list. The sign field of this first node is set to addition regardless of the actual sign of the parenthesized expression that is to be extracted next. This permits level n+1 to be bracket reduced and sorted without the need to process for the distributive law; that is, it avoids possible re-assignment to the sign field of each term within the parenthesized expression. The expression's true sign is placed in the last node of level n where the promoted expression will eventually be collapsed back in.

If the rightmost element in the last node of level n is the name of a transcendental or state variable function, an exponent base, a numerator, a left multiplication operand, or a left bracket (as part of a nested term), ApLvl(n) is set to True. The LbracLvl(n+1) count is set to 1.

3. The occurrence of a right parenthesis causes an a/s list to be bracket reduced if possible (refer to Rule 4), and should the last node remain a closing subterm in a list of two or more nodes, the list will be sorted. If it is a list of one node and the right parenthesis forms a complete subterm, an attempt at bracket reduction takes place as before followed by an assignment of priority for the term if it remains parenthesized. If the one node list becomes a closing subterm following the addition of the right parenthesis, then only priority assignment takes place. (This second situation arises when parentheses are removed from a complete subterm in the first node). In both situations, the first node receives a priority from among the priorities of the terms it contains. If all terms are constants, then the priority assigned to the node is the highest priority from the constants. However, if one or more terms are non-constants, then the value assigned is the highest priority of the non-constant terms. This approach prevents the constant priorities from dominating the ordering since many terms are associated with constants. When this level is collapsed, the node's priority may update the receiving node (refer to Rule 5).

   Unless all terms are negative, a lowest priority negative term is inserted in the second position of the expression and the leftmost position is reserved for the lowest priority positive term. Following a sort, the operators from the sign fields are appended to the corresponding node terms in re-creating the ordered expression that is inserted in the first node of the list. All other nodes of the list are deleted. The priority associated with the sorted expression is updated with the largest priority of the elements comprising the expression.

4. If the addition of a right parenthesis creates a complete subterm for level n, then the possibility of superfluous brackets exist. An equal number of left brackets is removed

from the head of the subterm as from the tail of the subterm provided the parenthesized expression is not any one of the following:

- A numerator or denominator term.
- An exponent base or exponent.
- An argument to a transcendental or state variable function.
- A multiplication factor as indicated either by the next operation in the equation being multiplication or by the last node in level n-1 terminating with the multiplication operator.
- A negative term as indicated by a subtraction operator in the sign field of the last node in level n-1.

The first three restrictions are identical to those in the multiplication phase while the last two restrictions, enforced by a Neg_Mult value of True, prevent the removal of parentheses enclosing multiplication factors and negative terms. For a level n with two or more nodes holding a complete subterm satisfying these restrictions, an equal number of brackets is reduced from the first and last nodes of the list until one of the two nodes is without parentheses.

If a term involves division or exponentiation, bracket reduction is not performed until both the left and right operands are in the same node. Furthermore, only the first three restrictions are applicable in these operations, i.e., enclosing parentheses may be removed from division or exponentiation terms which are also multiplication factors or negative expressions. For example, a sequence such as

"y * ((t + 1)/y) – (y^(2))"

can be simplified to "y * (t + 1)/y – y^(2)".

Note that only the outside brackets for the division and exponentiation are removed and the parenthesized numerator and exponent remain as such in accordance with the first two restrictions. One, both, or neither operands of the operation may be

parenthesized. Bracketed operand(s) must be collapsed from higher levels before the enclosing parentheses of the operation can be removed. Thus, bracket reduction for division and exponentiation terms is attempted both upon an append of ")" in a one node list (refer to Rule 3) and when a division or exponentiation expression is collapsed (refer to Rule 5).

Reducing parentheses in division and exponentiation operations involve parsing backward from the operator for the left operand, referred to as LeftOp, and parsing forward from the operator for the right operand, referred to as RightOp. In general, a bracket is marked for removal if it is an extra in the LeftOp that has a corresponding extra in RightOp and vice-versa.

For the LeftOp, any rightmost parenthesized term is bypassed by the backward parse that seeks additional left brackets beyond a balanced set. Any excess left brackets found is a candidate for elimination if it is not part of a larger division, exponent, or function argument term. For example, the LeftOp may be one of the following structures:

- term^(.../ or term^((...)/...
- term^(...^ or term^((...)^...
- term/(.../... or term/((...)/...
- term/(...^... or term/((...)^...
- fn(.../... or fn((...)/...
- fn(...^... or fn((...)^...

The LeftOp is the expression before the rightmost "/" or "^". The "fn" denotes a transcendental or state variable function name. In the processing, the system moves backward past any bracketed term "(...)" to look for excess left brackets. In every case, the leftmost bracket is part of an enclosing set of parentheses for a larger operation and should not be removed. However, the reverse is possible where the operations or functions are imbedded in a larger operand. For example, in the

structure "((... * (...)^(...))/(...))", three sets of brackets must be parsed in the numerator before reaching the excess left bracket. Thus, a left bracket is counted as superfluous once all right brackets have been balanced. The stopping condition for reading the LeftOp is either the beginning of the term or a blank space when all right brackets have been matched. A similar case exists for the RightOp in counting excess right brackets.

Although it appears redundant to assess a division or exponentiation operation contained within a larger operation that has already been parsed, each operation must be examined when it is encountered to consider the parentheses for that particular operation. For example, an operation such as "term/(((...)/...))" cannot have the parentheses in its left operand reduced based upon its role as the denominator to "term". However, the division operation in the denominator will have a LeftOp with excess left brackets that can be matched with excess right brackets in its RightOp. Reduction in parentheses results in the structure "term/(...)/...".

Once extraneous leading and trailing parentheses are stripped, the list is always sorted if its last node contains a closing subterm. If, on the other hand, the last node of level n holds a complete subterm or level n is without parentheses due to bracket removal, then for n>1, the level is collapsed to the end of the level n-1 list. The level n nodes are transferred in the sequence they occur to the tail of the level n-1 list beginning with a concatenation to the term in the last non-empty node if ApLvl(n-1)=True. ApLvl(n-1) is then reset. If Aplvl(n-1)=False, then the transfer begins with a new node in the receiving list. For n=1, the collapse step is bypassed and for all levels, the sort is delayed until a closing subterm occurs.

5. If a complete or closing subterm is in the first and only node of level n, n>1, which can arise either from a single element parenthesized term or a sort operation, then the expression is collapsed to the end of the level n-1 list. If ApLvl(n-1)=True, the collapse is a concatenation with the contents of the last non-empty node, otherwise, the subterm is placed in a new node appended to the level n-1 list. A collapse may update the priority field of the receiving node. If the collapsing subterm consists

entirely of constant terms, the priority field of the receiving node is not updated. If the subterm contains non-constant terms and it is collapsed to a blank node, the node receives the highest priority of the non-constant terms. However, if the collapse is to a non-empty node, the node is assigned the highest priority from the non-constant terms comprising its new expression. This ensures that subsequent sorts remain fine enough to distinguish between different elements that are associated with constants. For example, terms such as "a * sin(y)", "t^(a)", "(y + a)" and innumerable others that are associated with a constant, ("a" in this case), would all have the priority number 33. This would prevent the sort from ordering the structures according to their relative rank to each other. Thus, a node is only assigned the priority number of a constant (of priority number 30 and above) if does not contain non-constant terms.

For the collapse of a division or exponentiation expression that results in a complete subterm in the receiving node of level n-1, an attempt at bracket reduction is performed. This is required because the removal of superfluous brackets enclosing such expressions cannot take place until both the numerator and denominator or exponent base and exponent are in the same node (refer to Rule 4).

The LbracLvl(n-1) and RbracLvl(n-1) counts are updated accordingly and ApLvl(n-1), LbracLvl(n), RbracLvl(n) are reset. If n=1, the level is not collapsible and only the RbracLvl(1) is updated.

6. The collapse of an expression from level n+1 to level n will trigger a sort if it creates a closing subterm in the last node of level n. The resulting expression may in turn force a sort and a collapse of level n to level n-1 and in this manner, successive collapses can occur until n=1. A collapse that creates a complete subterm in the first node of level n is further collapsed to level n-1, n>1. A collapse to level n that results in a complete subterm in the last node, for a level n with two or more nodes, does not cause a sort since the complete subterm can be treated as one entity for subsequent sorting. In this case, processing continues at the last node of level n.

7. A parenthesized expression that begins with a negative term in level n will have the expression less the left parenthesis promoted to level n+1. This is because the sign

field associated with a left parenthesis ("(") node entry must contain the addition operator. (It would clearly be erroneous to apply a subtraction operator in the sign field of the node containing "(" due to a negative first element within the bracketed expression). The promotion accommodates the leading negative sign and its operand in the sign and term fields of the first node in level n+1. Aplvl(n) is set True. This is different from the case of a negative term which will have the subtraction sign entered in the last node of level n while the entire parenthesized expression is promoted to level n+1 as a positive term (refer to Rule 2).

The example ODE began in the first phase is continued next to illustrate the application of the a/s rules. In the example, the ApLvl flag is shown above each column level, and in each row are displayed the 3 fields for the sign, priority and term. As before, the priority is given a default of zero; a new empty node is indicated by "<="; and the applicable rule numbers are indicated by "R #' in the comments at the base of each level.

ODE (1'):

(y^(1 + n) * y' + y'' - (((y – b) + a)/((a – b) * (t + b))) + (b + (y - n)^(t)) * (y/n))

ApLvl(1)=T	ApLvl(2)=F	ApLvl(2)=F
0 + (y^	0 + (1 0 + n)	34 + (n + 1)
The start of every parenthesized expression begins with an addition operator in the sign field (R 2). The next "(" starts level 2 and ApLvl(1) is set to T since level 1 contains the exponent base (R 2).	The "+" in node 1 causes a blank node 2 to be appended (R 1). The ")" creates a closing subterm in node 2 causing a sort of level 2 (R 3). The term cannot be bracket reduced since it is an exponent base, i.e. ApLvl(1)=T (R 4).	The sorted result is placed in node 1 and the priority number is updated (R 3). Since the result is a complete subterm, it is collapsed to level 1. The collapse is an append with the level 1 node 1 term and ApLvl(1) is reset (R 5).
ApLvl(1)=F	**ApLvl(2)=T**	**ApLvl(3)=T**
8 + (y^(n + 1) * y' 0 + y'' 0 - <=	0 + (	0 + (

ApLvl(1) is reset. The receiving node is assigned the highest priority of its non-constant terms (R 5). The mult. operator and operand is appended in node 1 and node 2 is created by the 3rd "+" (R 1). The "-" adds a new node 3 (R 1). The next "(" starts level 2 (R 2).	The next "(" starts level 3 for the nested subterm of level 2 which sets ApLvl(2) to T (R 2).	The next "(" starts level 4 for the nested subterm of level 3 which sets ApLvl(3) to T (R 2).
ApLvl(4)=F	ApLvl(4)=F	ApLvl(3)=F
0 + (y 0 - b)	0 + y 0 - b	0 + (y 0 - b 0 + a)
The "-" appends a new node 2 (R 1). The ")" creates a closing subterm in node 2 and a complete subterm for the level. Bracket reduction takes place despite Aplvl(3)=T since the term is a nested expression of level 3 (R 4).	Level 4 is without parentheses following bracket reduction so collapse to level 3 (R 4). Collapse begins at level 3 node 1 due to ApLvl(3)=T. ApLvl(3) is then reset (R 4).	The 2nd "+" adds new node 3 (R 1) and the ")" entered in it creates a closing subterm for the node. A peek ahead finds the division operator so bracket reduction is not performed (R 4). The closing subterm triggers a sort.
ApLvl(3)=F	ApLvl(2)=T	ApLvl(3)=T
9 + (y + a – b)	9 + ((y + a – b)/	0 + (

The sort resolves the identical priority constants "a" and "b" alphabetically and updates the node's priority with the highest priority of the elements (16 for y) (R 3). The complete subterm in node 1 is collapsed to level 2 (R 5).	The collapse is a concatenation with node 1 and ApLvl(2) is reset. Node 1 is also updated with the higher priority (R 5). The "(" for the denominator starts a new level 3 and sets ApLvl(2) to T (R 2).	The next "(" starts level 4 to hold the nested subterm and sets ApLvl(3) to T accordingly (R 2).
ApLvl(4)=F	ApLvl(4)=F	ApLvl(3)=T
0 + (a 0 – b)	33 + (a – b)	0 + ((a – b) *
The "-" appends a new node 2 (R 1) and the closing ")" entered in node 2 leads to a sort (R 3, 4). A peek ahead indicates that this term is a multiplication factor that precludes bracket removal (R 4).	The complete subterm in node 1 is collapsed to the last non-empty node in level 3. The node priority is not updated since the collapsed subterm consists entirely of constants. Aplvl(3) is reset (R 5).	The next "(" starts a new level 4 and sets ApLvl(3) to T since level 3 contains the left multiplication operand (R 2).
ApLvl(4)=F	ApLvl(4)=F	ApLvl(3)=F
0 + (t 0 + b)	33 + (t + b)	7 + ((a – b) * (t + b))
Node 2 begins due to the 2nd "+" (R 1) and when it receives the closing ")", bracket reduction is not performed because level 4 is a multiplication operand (R 4). The closing subterm triggers a sort (R 3).	The sorted result in node 1 is a complete subterm that is collapsed to level 3. The collapse is to the last non-empty node, node 1, since ApLvl(3)=T. The node's priority is updated with the highest priority of the terms in its new expression. ApLvl(3) is reset (R 5).	Adding the last ")" creates a complete subterm in node 1 which is collapsed to level 2. With ApLvl(2)=T, the collapse is to the last non-empty node. This node's priority is then updated with the highest priority of all its terms. ApLvl(2) is reset (R 5).
ApLvl(2)=F	ApLvl(2)=F	ApLvl(2)=F
9 + ((y + a – b)/((a – b) * (t + b))	9 + ((y + a – b)/((a – b) * (t + b)))	9 + (y + a – b)/((a – b) * (t + b))

The collapse does not result in a complete subterm so bracket reduction not done for division (R 5). Also not sorted since it is not a closing subterm (R 6). The next ")" creates a complete subterm that undergoes bracket reduction for division (R 4).	The bracket reduction process first verifies that the division term is not a function base or an exponent operand. (Does so by checking ApLvl(1) and pattern-matching). Outside set of matching parentheses is removed (R 4).	Following bracket reduction, the complete subterm in node 1 is collapsed to the first empty node, node 3, of level 1. Priority of node 3 is updated (R 5).
ApLvl(1)=F	ApLvl(1)=F	ApLvl(2)=F
8 + (y^(n + 1) * y' 0 + y'' 9 - (y + a – b)/((a – b) * (t + b))	8 + (y^(n + 1) * y' 0 + y'' 9 - (y + a – b)/((a – b) * (t + b)) 0 + <=	0 + (b 0 +
The collapse of the complete subterm does not lead to a sort (R 6). The node 3 expression has changed from a positive term in level 2 back to a negative one due to the memory of the node 3 sign field (R 2).	The next "+" appends a new node 4 (R 1). The following "(" starts a new level 2 (R 2).	The 2nd "+" appends a new node 2 (R 1). The following "(" starts a new level 3 (R 2).
ApLvl(3)=F	ApLvl(2)=T	ApLvl(3)=F
0 + (y 0 - n)	0 + (b 9 + (y – n)^	7 + (t)
The "-" appends a new node 2 (R 1). The closing ")" in node 2 triggers bracket reduction and sort. Peek ahead shows term is an exponent base so no brackets are removed. Proceeds with sort (R 3).	The sorted result is a complete subterm collapsed to node 2. The collapsed term's constant priority (33) is discarded. Instead, node 2 is assigned the highest priority of its non-constant terms (R 5). The exponent begins level 3 and sets ApLvl(2) to T (R 2).	As an exponent, the complete subterm cannot be bracket reduced (R 4). When the complete subterm is formed in node 1, the node is assigned the subterm's priority (R 3). The complete subterm is collapsed to level 2 (R 5).

ApLvl(2)=F	ApLvl(2)=F	ApLvl(1)=T
0 + (b 9 + (y – n)^(t))	33 + ((y – n)^(t) + b)	8 + (y^(n + 1) * y' 0 + y'' 9 - (y + a – b)/((a – b) * (t + b)) 9 + ((y – n)^(t) + b) *
The collapsed term from level 3 is added to node 2 and ApLvl(2) is reset (R 5). No excess brackets exist on the exponent term so none removed (R 4). The closing ")" in node 2 causes a sort (R 3).	Peek ahead shows the term is a mult. factor which precludes bracket removal on the complete subterm (R 4). Term is collapsed to level 1 and the receiving node is assigned the highest priority of its non-constant terms (R 5).	The complete subterm collapsed to node 4 does not trigger a sort (R 6). The "*" is appended in the current node (R 1). The next "(" begins a new level and sets ApLvl(1) to T since level 2 will have the mult. factor.
ApLvl(2)=F	ApLvl(2)=F	ApLvl(2)=F
0 + (y/t)	0 + y/t	9 + y/t)
The closing ")" forms a complete subterm which undergoes bracket reduction for division. Brackets may be removed from a division term that is also a mult. factor (R 4).	Brackets have been removed from the term (R 4).	The next ")" creates a closing subterm in node 1 which is assigned the highest priority of its elements (9 for "y") (R 3). The closing subterm is then collapsed to level 1 (R 5).
ApLvl(1)=F	ApLvl(1)=F	ApLvl(1)=F
8 + (y^(n + 1) * y' 0 + y'' 9 - (y + a – b)/((a – b) * (t + b)) 9 + ((y – n)^(t) + b) * y/t)	8 + (y^(n + 1) * y' 4 + y'' 9 - (y + a – b)/((a – b) * (t + b)) 9 + ((y – n)^(t) + b) * y/t)	(y'' + (y^(n + 1) * y' - (y + a – b)/((a – b) * (t + b)) + ((y – n)^(t) + b) * y/t)

The collapse is a concatenation with node 4 and ApLvl(1) is reset (R 5). The closing subterm created in node 4 triggers a sort (R 6).	The sort first assigns any missing priorities. Nodes with identical priority, nodes 3 and 4, have the ordering of their terms resolved alphabetically.	This completes the a/s phase and the sort.

The original equation of (ODE 1):

(y' * y^(1 + n) + y'' – (((y – b) + a)/((t + b) * (a – b))) + (y/n) * (b + (y – n)^(t)))

has been sorted to (ODE 1''):

(y'' + (y^(n + 1) * y' - (y + a – b)/((a – b) * (t + b)) + ((y – n)^(t) + b) * y/t)

by sorting the elements according to the priorities of Table 6.1 in the multiplication and a/s sort phases and by removing extraneous brackets. Other arrangements of the terms in ODE 1 will lead to ODE 1'' as long as the semantics of the equation are unchanged.

6.3 Development of the ODEASY System

The ODEASY system is an Excel workbook harnessed to a Visual Basic program that controls the user interface and processes the inputs into the database. The workbook contains worksheets for the following purposes:

- Forms for equation entry, solution entry, and display of search result. These forms include on-sheet menu buttons that activate Visual Basic procedures.
- Storage for the default values of dialogboxes, for the values of variables, and for the priority table.
- A work area for the system to perform the sort to convert the ODE into the canonical form recognized by the database.

- To hold the database of ODEs and their solutions.

The software that manages the ODEASY system is contained in 8 modules as detailed in §6.3.3 to §6.3.3.9. The database of ODEASY is discussed next.

6.3.1 The Database

The database is comprised of worksheets that contain the equations and their related information. A sheet exists for each order and linearity of ODEs and is formatted as follows:

First Order Linear Ordinary Differential Equations

index	Condition	General Solution (y=)	Integration Constant (k=)	Reference	Additional Notes
y' - a * y	b=0	k * exp(a * t)	yo/exp(a * to)		
y' - a * y + b	b<>0	k * exp(a * t) - b/a	(yo + b/a) * exp(-a * to)		
y' - b * t^(a)		b/(a + 1) * t^(a + 1) + k	yo - b/(a + 1) * to^(a + 1)		
y' - a * y + b * t		k * exp(a * t) - (b/a) * t - (b/a)^2	exp(-a.to).(xo + (b/a).to + (b/a)^2)		

An equation may have several solutions depending upon the values of certain parameters. In this case, one or more conditions comprising the condition set for one solution is specified in the Condition column and the associated solution begins in the same row as the first condition of the set. The ODE is an index of the database and each of the ODE's solutions is an entry for that index. Once an index is known, it is a simple matter to extract any solution by examining the condition set.

6.3.2 Incorporating Initial Conditions

As indicated in §6.3.1, a Conditions column holds the condition(s), if any, pertaining to each solution. These conditions may be conditions on parameters, ranges for the independent variable, and initial conditions. The specification of initial conditions is particularly relevant for some non-linear equations for which the form and possibly, the existence, of an exact solution depend upon the initial condition.

Conditions are specified when entering the solution for an ODE. The system will not permit the entry of a solution for an ODE which has one or more existing solutions in the database unless the new solution is differentiated by a condition set that is unique from

the condition set(s) of the existing solution(s) of the ODE. A successful query in ODEASY permits the user to view all the solutions and associated conditions for the matching ODE. Please refer to §6.3.2.5 and §6.3.2.6 for details on the handling of conditions in the ODEASY Search and Add modules.

6.3.3 Software Architecture

The ODEASY system is comprised of the modules shown in the structure chart below:

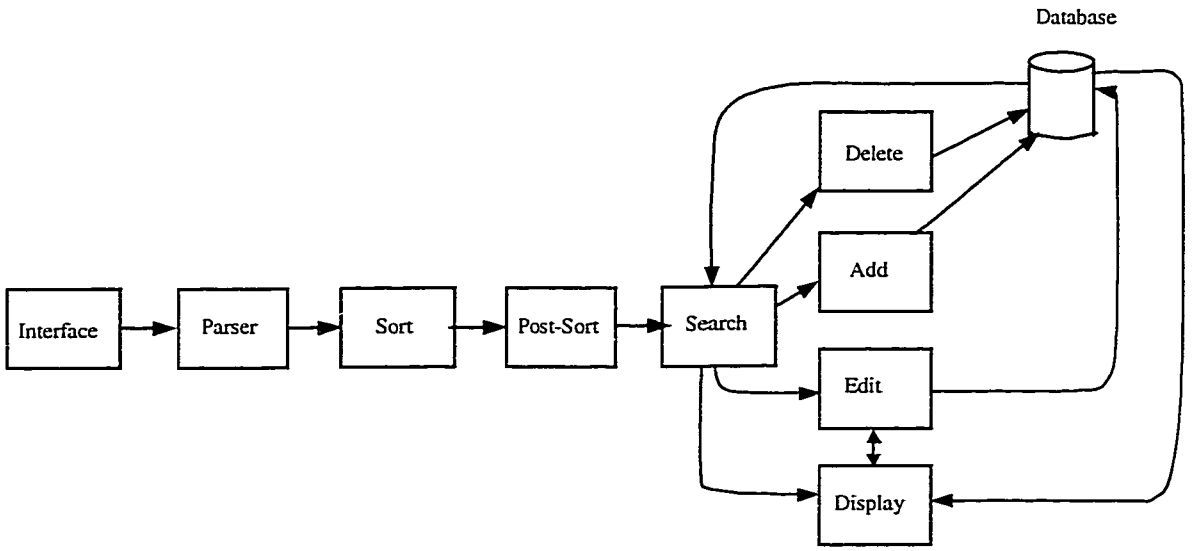


Figure 6.1 ODEASY Architecture.

An overview of the features of each module is discussed next.

6.3.3.1 Interface

When the ODEASY application is double-clicked, an auto-start procedure in the interface sets up the ODEASY environment (eliminating such things as gridlines, headings, scrollbars, book tabs, etc., normally associated Excel) and activates the ODEASY logo followed by the Main Menu, shown in Figure 6.2. The application returns the environment to the previous Excel settings when it is closed.

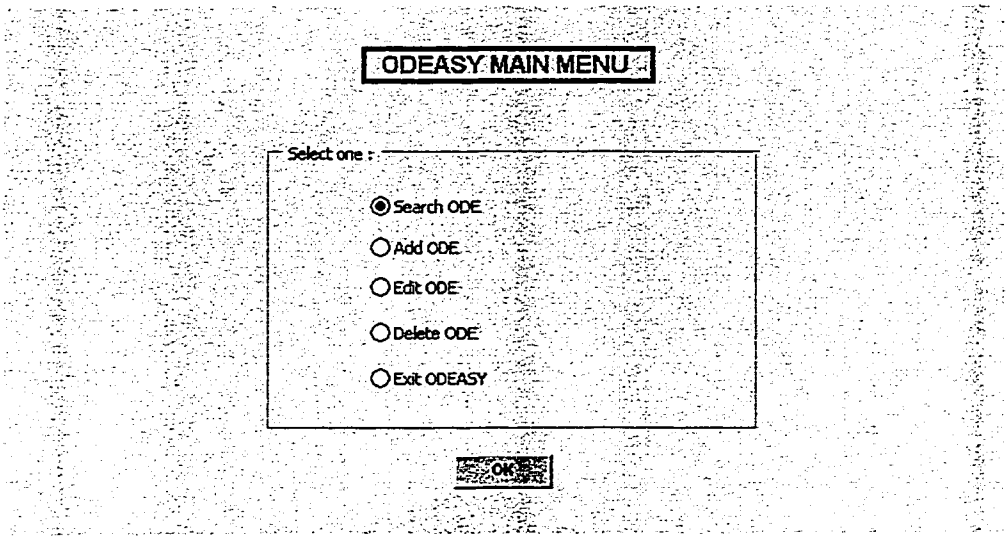


Figure 6.2 ODEASY Main Menu

All menu selections except 'Exit' brings up an identical ODE Entry Form except for customization on the title. The form, shown in Figure 6.3, offers both point-and-click and manual entry of the equation. Every component of the equation is available as buttons or button activated dialogboxes that the user can click on to enter the input.

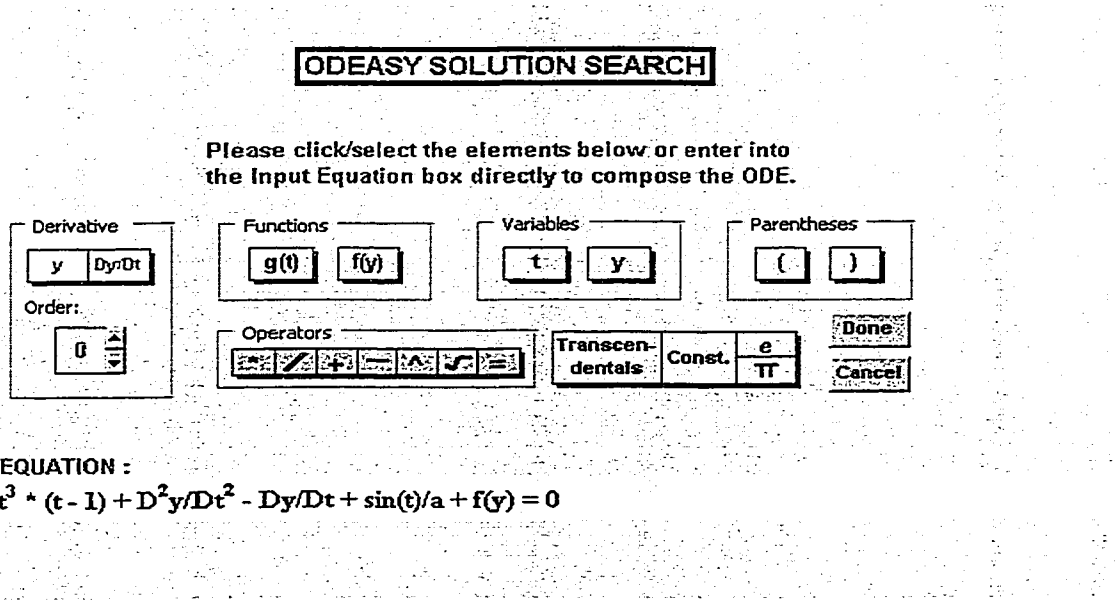


Figure 6.3 ODE Entry Form

In point-and-click entry, the derivative buttons, "Dy/Dt" and "y", are disabled until a derivative order has been selected in the associated spinner textbox below. In Figure 6.3, the "Dy/Dt" button is used repeatedly to enter an equation in the entry bar. The system disables the derivative buttons once a derivative is written but accepts additional derivatives provided an order selection is made beforehand. This minimizes erroneous entries; however, the system is flexible enough to accept both derivative forms in the same equation. The user is made aware of the enabled/disabled status of buttons via colour changes on the buttons from light to bold when a button becomes available.

A click on the "Transcendentals" button brings up the Transcendental Functions dialogbox shown in Figure 6.4, while selecting the "Constants" button activates the Constants dialogbox shown in Figure 6.5. The Constants dialogbox is sensitive to the side on which the mouse is clicked by automatically setting the appropriate radio button at the base of the dialogbox.

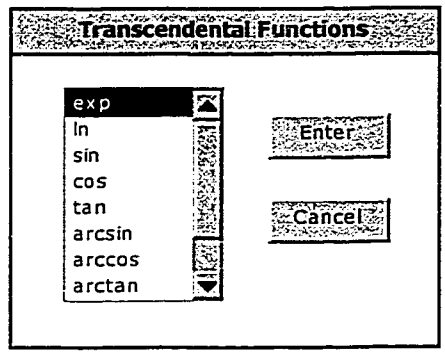


Figure 6.4 Transcendental functions dialogbox

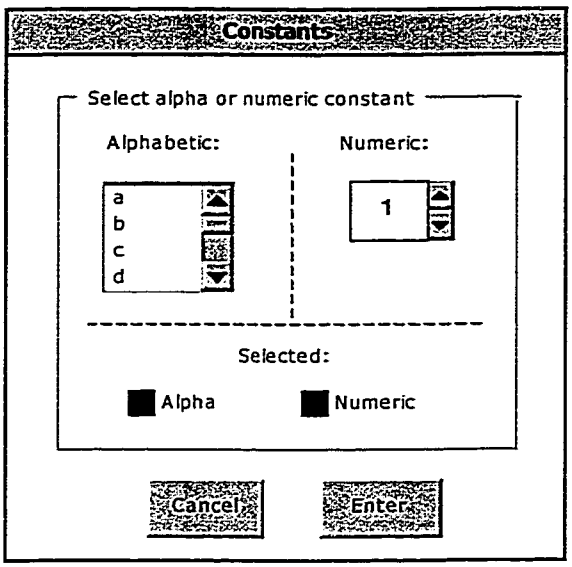


Figure 6.5 Constants dialogbox

Any editing or manual entry can be performed directly by the user in the entry bar. The user is not restricted to the alphabetic characters presented in the Entry Form to designate the various entities. Via manual entry, the user may pick any other alphabetic characters to represent each of the variables. The interface is capable of accepting an ODE in any form with terms either on the LHS, RHS, or both sides of the equation. Even if the "=" sign has not been entered, the system assumes the equation is complete once the "Done" button is clicked which sends the input equation to the Parser. A click on the Cancel button cancels the current entry and returns the user to the Main Menu.

Other aspects of the ODEASY interface are activated by the specific actions that are to be taken and are presented with the overview of those actions. The next module, the Parser, affords the equation entry its flexibility.

6.3.3.2 Parser

The Parser manipulates the input equation into a format that the sort can recognize. Its parsing duties include:

- Ensuring that the input is an ODE.

- Querying the user to obtain the name of the independent variable (for equations entered in "y'" form).
- Changing equations entered in "Dy/Dt" form into the equivalent "y'" form.
- Moving all terms to the LHS of the equation, removing the "=" sign, ensuring uniform spacing, and enclosing the equation in parentheses.
- Isolating variables from function names in the equation text.
- Mapping the dependent and independent variables and state variable functions in the input equation to their internal representation.

The last point permits any variable name to represent the state variables, functions of state variables, and constants. Internally, the ODEASY database understands the dependent variable to be "y", the independent variable to be "t", the function of the dependent variable to be "f", and the function of the independent variable to be "g". Each of these names, referred to as the internal variables, could be used in the input to represent an element other than what they identify in the database. For non-constant variables, the mapping of the input variables, also referred to as display variables, to the internal variables is performed prior to the sort. This is a prerequisite for the sorting algorithm which must match terms with priority templates (i.e., in Table 6.1) that are defined via the internal variables. The Parser maintains a Variable List, shown in Table 6.2, of the internal variables and their corresponding display variables. The independent variable has been determined earlier either from the user if the entry is in "y'" form or from the equation if it is entered in "Dy/Dt" form. The dependent variable is determined by parsing for the prime symbol ("'") and the names of the functions of the state variables are deduced from the names of the dependent and independent variables. For example, the Variable List in Table 6.2 contains the mapping for an equation such as "w' + sin(r) + k(w) + l(r) = 0" where the user has specified that "r" is the independent variable:

Table 6.2 Variable List

	Internal Variable	Display Variable
Dependent variable	y	w
Independent variable	t	r
Fn1: g(t)	g	l
Fn2: f(y)	f	k
Constants:	a	
	b	
	c	
	d	
	h	
	k	
	m	
	n	
	q	

Display constant variables that are not represented by internal variables must be mapped following the sort (refer to PostSort in §6.3.3.4). Not re-assigning constants before the sort has no impact on the ordering since the priority of constants in terms that also have non-constants are ignored in favour of the highest non-constant priority in the a/s sort phase. Thus, the ordering of the terms is dictated by the non-constants in the final a/s sequence of the equation. During the parsing, constant names found are entered in a Constants List so that the substitution for these constants in the PostSort stage can be performed by seeking the names rather than identifying each term in the equation to find constants.

In the equation, the Parser converts the dependent variable, then the independent variable, followed by the function of independent variable, and lastly, the function of the dependent variable. In the event that an internal variable is used to represent an entity other than what is understood by the database, an intermediate step in the conversion is required. A predefined symbol exists for each of the internal variables as shown in Table 6.3. The symbols are arbitrarily selected and are non-alphabetic (to prevent coinciding with other variable names).

Table 6.3 Predefined Symbols

Internal Variable	Predefined Symbol
y	@
t	~
g	!

f	#

The intermediate step replaces all instances of an internal variable in the equation with its predefined symbol. For non-constants, this symbol is entered in the Variable List at the display variable position associated with the internal variable that should be in the equation. Following this intermediate replacement, the conversion continues with the next variable type. In the case of an internal variable used to represent a constant in the input, the internal variable's symbol is substituted in the equation but the variable list is not updated since constants are re-assigned at the PostSort stage. The system interprets any predefined symbols left in an equation as constants. For example, in the equation "t' + g + h * sin(y) * f(t) + y" where the dependent and independent variables are switched from their database meaning, and "g" is a constant rather than a function name, the conversion steps are as follows:

⇨ ~' + g + h * sin(y) * f(~) + y

⇨ ~' + g + h * sin(@) * f(~) + @

⇨ ~' + ! + h * sin(@) * f(~) + @

The updates to the Variable List are in Table 6.4.

Table 6.4 First update to Variable List

	Internal Variable	Display Variable
Dependent variable	y	~
Independent variable	t	@
Fn1: g(t)	g	g
Fn2: f(y)	f	f

Without this intermediate step, the conversion is ambiguous since the same variable name would refer to different entities in the equation; e.g., a direct substitution for the dependent variable changes the representation to "y' + g + h * sin(y) * f(y) + y" making it impossible to distinguish the dependent variable from the independent variable for the next substitution on the independent variable from "y" to "t".

The intermediate step is followed by an exchange of the predefined symbols in the equation with the internal variables that they are associated with in the Variable List:

⇨ y' + ! + h * sin(@) * f(y) + @

⇨ y' + ! + h * sin(t) * f(y) + t

The sort produces "y' + t + h * f(y) * sin(t) + !" with the two constants exchanged in the postSort stage to obtain the database index "y' + t + a * f(y) * sin(t) + b". The Variable List for this equation is shown in Table 6.5.

Table 6.5 Second Update to Variable List

	Internal Variable	Display Variable
Dependent variable	y	~
Independent variable	t	@
Fn1: g(t)	g	g
Fn2: f(y)	f	f
Constants:	a	h
	b	!

If a solution is available, its database representation in the internal variables can be converted back to the display variables for the solution to the user by consulting the Variable List. Any predefined symbols are replaced by their associated internal variable. For example, the dependent variable "y'" with a display variable of the predefined symbol "~" would be replaced by the symbol's internal variable, "t", in the result.

6.3.3.3 Sort

The Sort module accepts the parenthesized equation from the Parser and treats it with the two phase sorting process involving the multiplication phase and the addition/subtraction phase described in the ODEASY Sort Algorithm in §6.2.1.

6.3.3.4 PostSort

The PostSort module removes the sorted equation's enclosing brackets and converts its constant variables into the internal variables. This mapping of constants is performed after the sort to preserve an ordering that must follow the sequence in which the database

expects constant names. Constants are identified in a database equation in the sequence that they appear in the Internal Variable column of the Variable List. Thus, a database equation with three constants must have them referred to, in order of appearance, as "a", "b", and "c". Note that if the equation's constants were converted prior to the sort, the result may not subscribe to the accepted sequence. For example, if an equation such as "y' + m * sin(y) – n * y + p * t = 0" with constants "m", "n", and "p" were to be completely substituted with internal variables before sorting, it becomes "y' + a * sin(y) – b * y + c * t". The sort on this equation would leave it as "y' + c * t – b * y + a * sin(y)" which cannot be recognized in the database. Instead, the correct constant sequence is enforced by performing the constant conversions subsequent to the sort to obtain "y' + a * t – b * y + c * sin(y)".

Extracting the constants from the sorted equation has been made more efficient by the Parser which left the names of contants it found in a Constants List. The PostSort module traverses this list and searches the equation for a match. Matches found are verified to not be parts of transcendental function names.

6.3.3.5 Search

By the time an equation arrives at the Search module, it has been changed into a database key, also referred to as an index equation. If a solution exists for this equation, it will match an identical key in the database. The Search module determines the highest order derivative in the equation to target the correct order database sheet. It then finds the equation's linearity by:

- Searching for the dependent variable and its derivatives that occur beyond the first degree in the equation. This includes looking for the dependent variable in any denominator terms.
- Searching for transcendental functions that have the dependent variable or its derivatives as an argument.
- Searching for products of the dependent variable and its derivatives.

Should any of the three searches succeed, the Search module goes to the non-linear version of the order sheet, otherwise, it will seek the solution in the linear sheet of the correct order.

If a search fails, the user is notified along with a query on whether another ODE should be tried. For a successful attempt, the Search module sends the solution location and any associated conditions to the Display module. For an ODE with several solutions, each uniquely identified by the conditions (on parameters, initial conditions, or ranges on the independent variable), the Display for Search Result will enable menu buttons to scroll to the next or previous solution. For example, an entry in the ODE Entry Form of

$$t' - y^{(m)}/(t^{(q)} * (n + y^{(1 + m)})^{(p)}) * z = 0$$

is sorted and converted to the canonical form

$$y' - a * t^{(b)}/((t^{(b + 1)} + c)^{(d)} * y^{(h)}) = 0$$

which is identified as a first order non-linear ODE that matches the sixth equation in the first order non-linear database sheet. This ODE has 2 solutions depending upon the value of h (represented as q by the user). The display of the result is shown in

Figure 6.6.

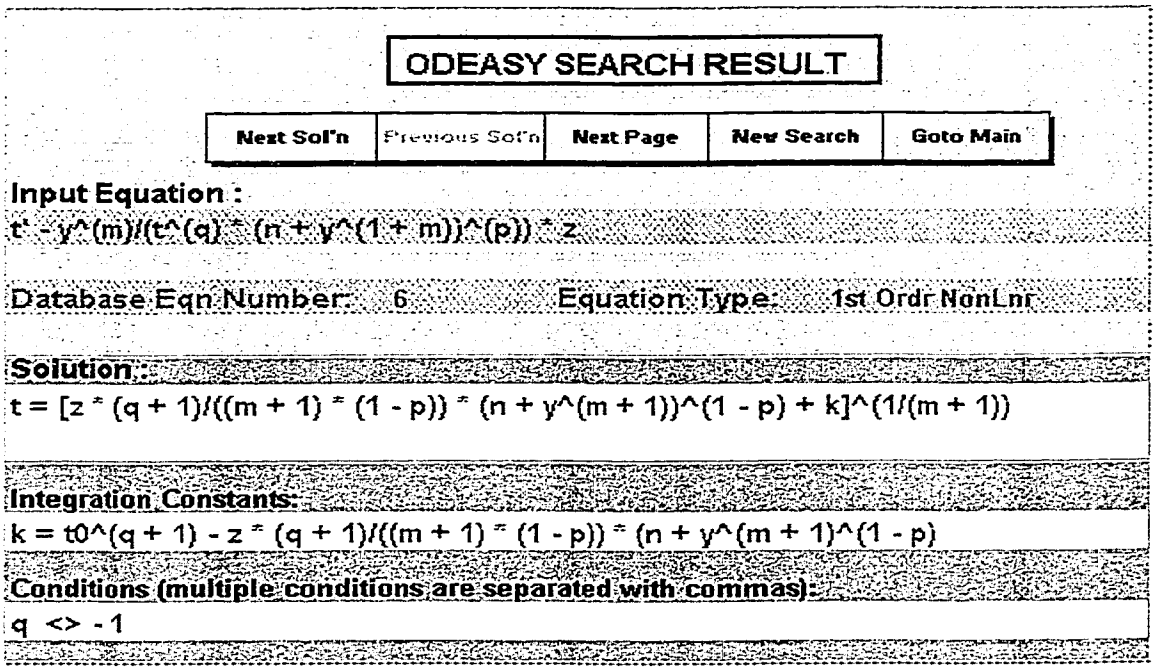


Figure 6.6 ODEASY Search Result for the first solution

Note that the solution, integration constant, and condition are displayed in the variables that the user has selected for the equation entry rather than the variables used by the database. Since there is another solution available for this ODE, the "Next Sol'n" button is enabled to indicate this to the user. Clicking this button updates the display to the next solution as shown in Figure 6.7.

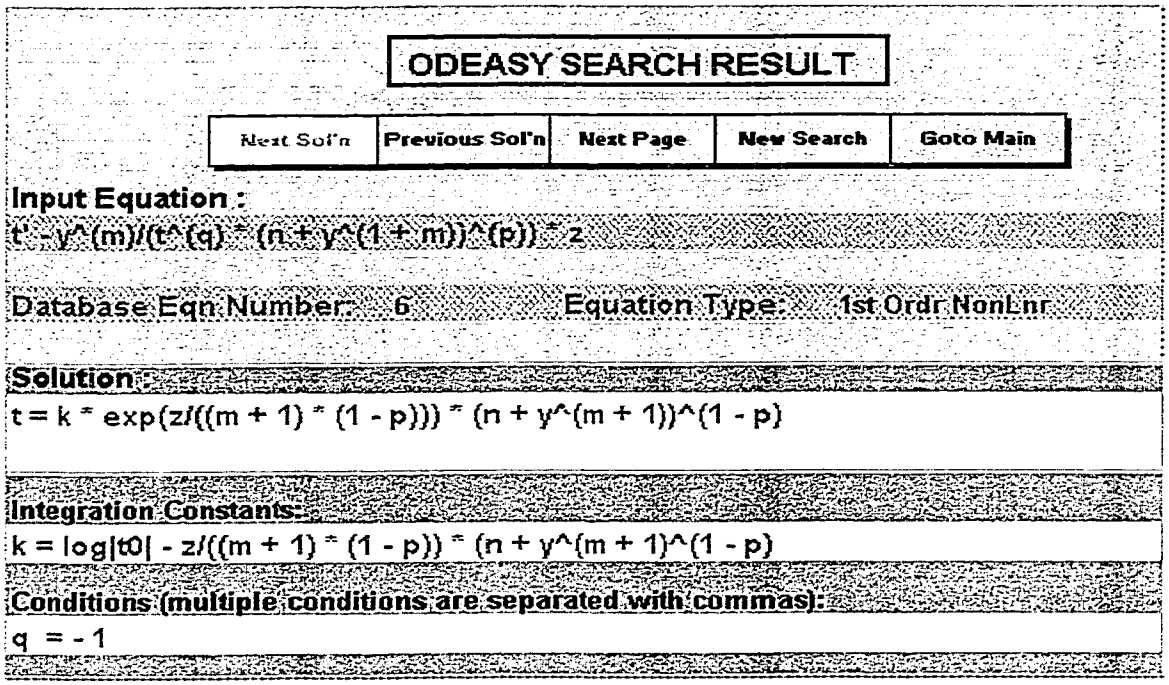


Figure 6.7 ODEASY Search Result for the last solution

Figure 6.7 shows the last solution for the query as indicated by the disabled "Next Sol'n" button. The user may access the previous solution by clicking the "Prev Sol'n" button. If only a single solution exists, both the "Next Sol'n" and "Prev Sol'n" buttons are disabled and faded.

For longer length solutions or condition sets (beyond the length of 1 screen) the horizontal scrollbar (not shown) can be used to view to the end of the string. The capacity of each of the display bars is 256 characters.

Additional information on an ODE can be obtained by clicking on the "Next Page" button as shown in Figure 6.8. This permits an equation and solution to be traced back to the source.

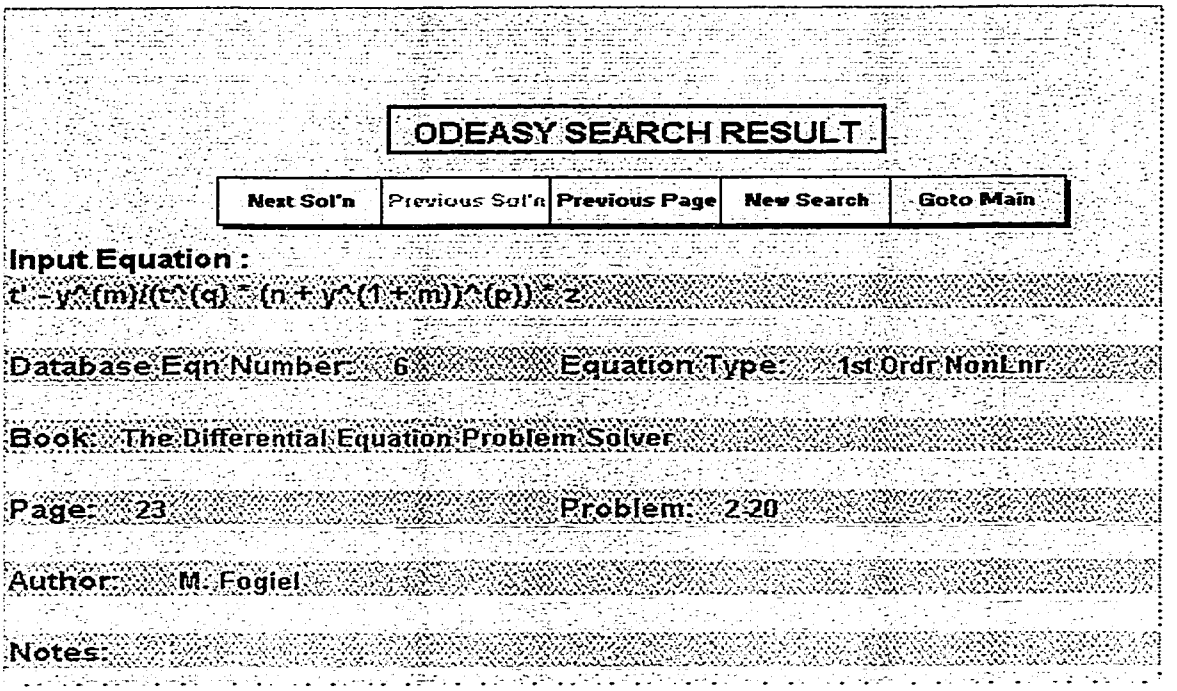


Figure 6.8 Second page of Search Result provides additional information

The "New Search" button dismisses the Search Result display form and returns the user to the ODE Entry Form to obtain an ODE for a new search.

The processing path for an ODEASY search is provided in the control flow diagram of Figure 6.9.

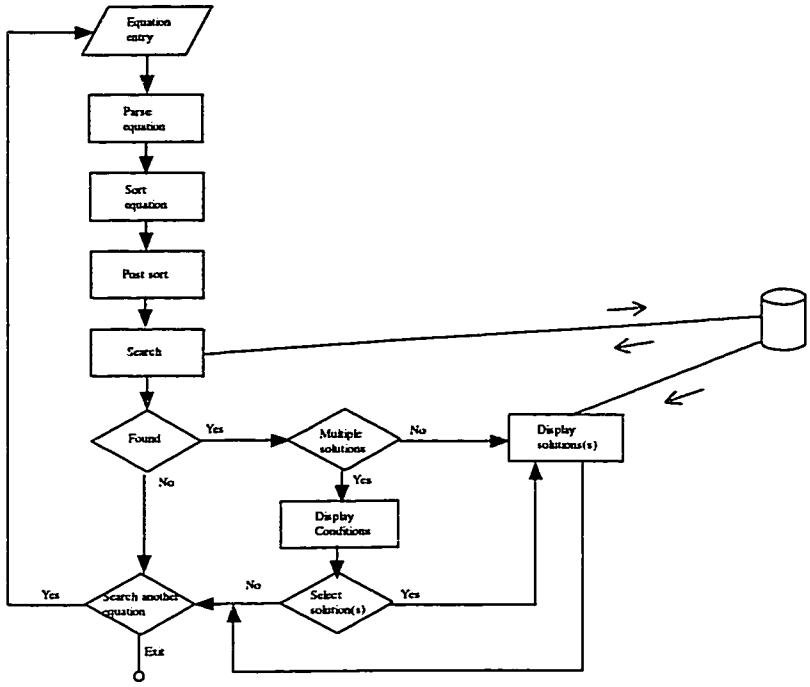


Figure 6.9 The Search Operation

6.3.3.6 Add

The Add module adds an entry into the database if it does not already exist. If the index equation is found in the database, the Add module will not permit the entry of a new solution unless it is given a condition set that is unique from the condition set(s) of the other solution(s) for the index equation.

The Add module provides a blank Add Entry Form, shown in Figure 6.10, for the user to enter the new data. The input equation, as entered by the user, is repeated at the top. (This original equation from the ODE Entry Form is kept in a global variable that is accessed by both the Add and Display modules). Unlike the Search and Edit functions, the Display module is not required by the Add function since it has no existing data to display. The input equation is also shown in the canonical form of the database.

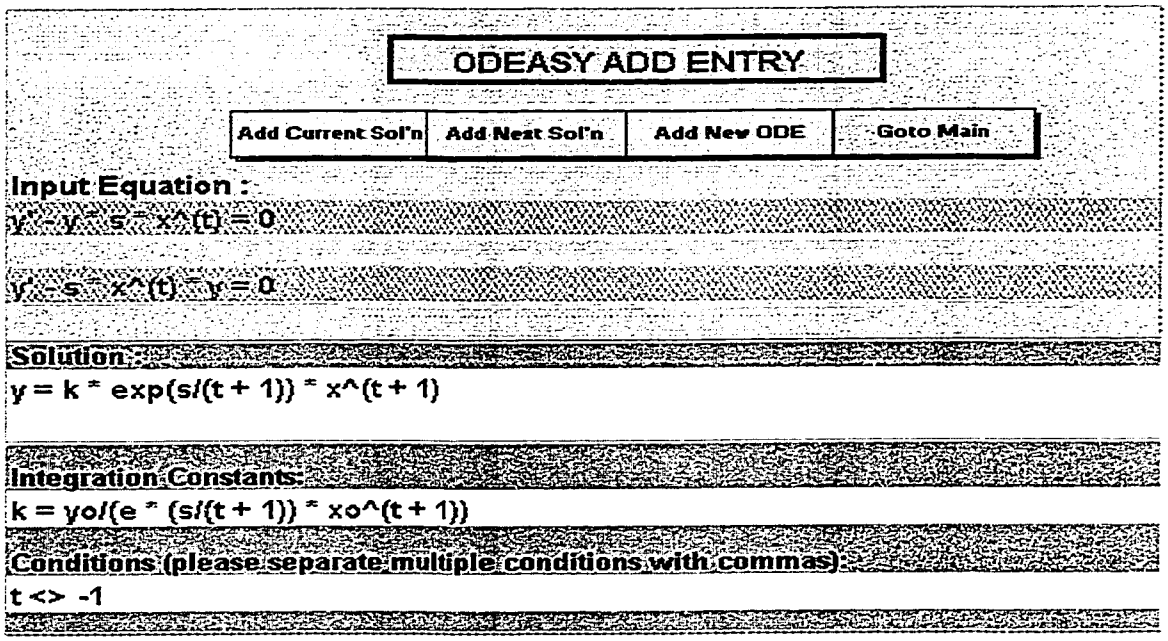


Figure 6.10 Add Entry Form

In the Add Entry Form, the highlighted bars are available for data entry. With the menu buttons across the top, the "Add Next Sol'n" button allows for an ODE repeated entries of new solutions based upon unique condition sets. If the "Add New ODE" button is clicked, the Add Entry Form is dismissed and the user is returned to the ODE Entry Form to enter a new equation. The "Add Current Sol'n" button sends the current data in the form for error-checking and processing. The solution and conditions are converted to their internal variable representation before writing into the database. A dialogbox, shown in Figure 6.11, is invoked to obtain additional information if it is a valid entry about to be committed into the database.

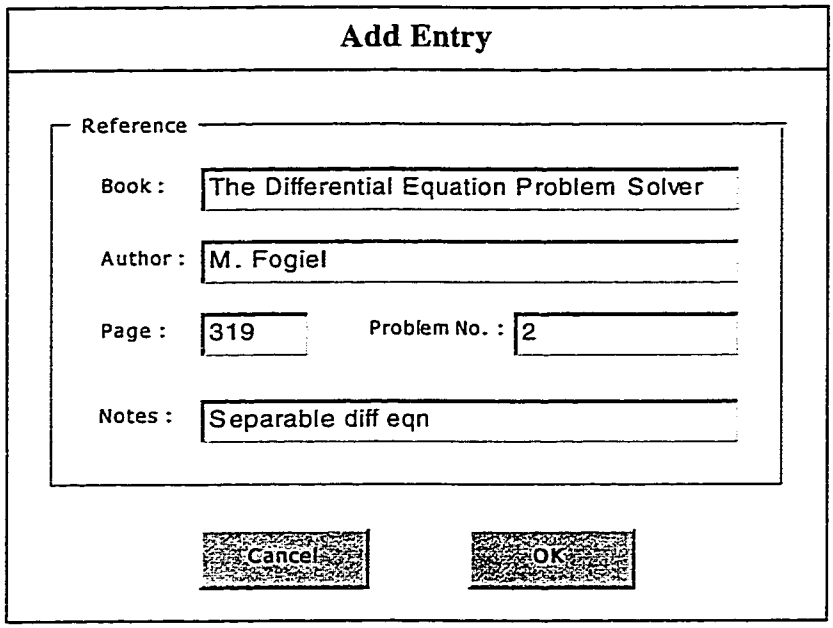


Figure 6.11 Additional information for Add Entry

If an entered solution already exists, or if the solution is not entered or not valid, the user is notified and queried as to whether another ODE is to be added. The control flow diagram for the Add function is provided in Figure 6.12.

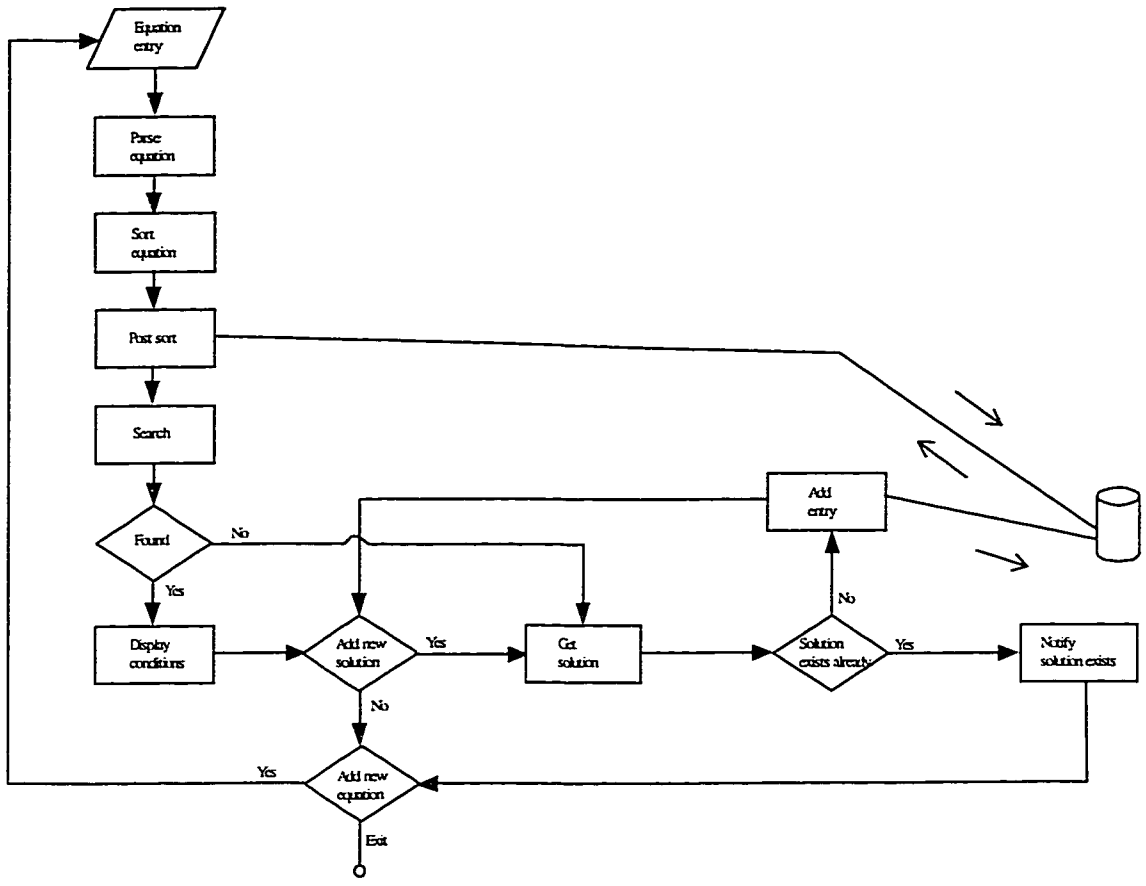


Figure 6.12 The Add Operation

6.3.3.7 Edit

The Edit module permits the user to modify one or more solution that is associated with an input equation. If multiple solutions exist, a dialogbox similar to the one in Figure 6.13 is activated. This is a dynamic dialogbox with a parameter list that customizes to the conditions and display variables of the input equation. The condition options are taken from the Conditions column for the solution in the database sheet. Three options are indicated here but this list as well as each list item may expand or shrink depending upon the particulars of the condition set for the index equation. This feature takes advantage of Excel dialogboxes and dialogbox objects which can be dynamically created, re-sized, and deleted. To ensure that the user can relate to the variables indicated, they have been

converted back to their display names (from the database representation) by consulting the Variable List.

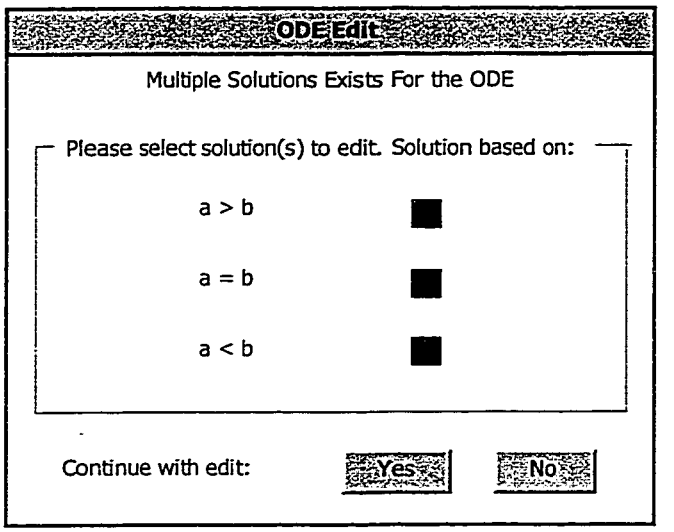


Figure 6.13 Multiple solutions dialogbox for Edit

Once one or more solutions have been chosen for update, the Edit module calls the Display module with the location of the index equation and one condition set for one specific solution if multiple solutions are involved. The Display module, which converts the selected database entry into its display variable equivalent, brings on screen the Edit Entry Form shown in Figure 6.17. Each item displayed, other than the "Input Equation" (at the top), may be edited. The menu selection is restricted to canceling the edit or confirming that the edit is complete. This is because changes to the database is processed one entry at a time. The Display module re-converts the update into its internal variable representation and sends the information back to the Edit module. When one edit has been processed, the Edit module sends the next entry location to the Display module if multiple solutions are involved.

If the user has changed the index equation, it must be parsed, sorted, and post-sorted to ensure its validity as an index equation. An altered index equation is first searched in the database and is accepted only if the search is unsuccessful. An update is not performed if

a changed index equation already exists in order to protect the integrity of the database. The user is notified that such an update should be based on the altered index as the original input equation for the edit. Where a changed index is not found in the database and it is a single solution equation, its current sheet location is simply replaced. For a multiple solution equation, the edited solution is deleted from its current location and a new entry is added into the database with the altered index and its solution.

If an input equation is not found or if no solution is chosen for a multiple solution entry, the user is queried on whether another database entry should be edited. The control flow diagram for the edit function is shown in Figure 6.14.

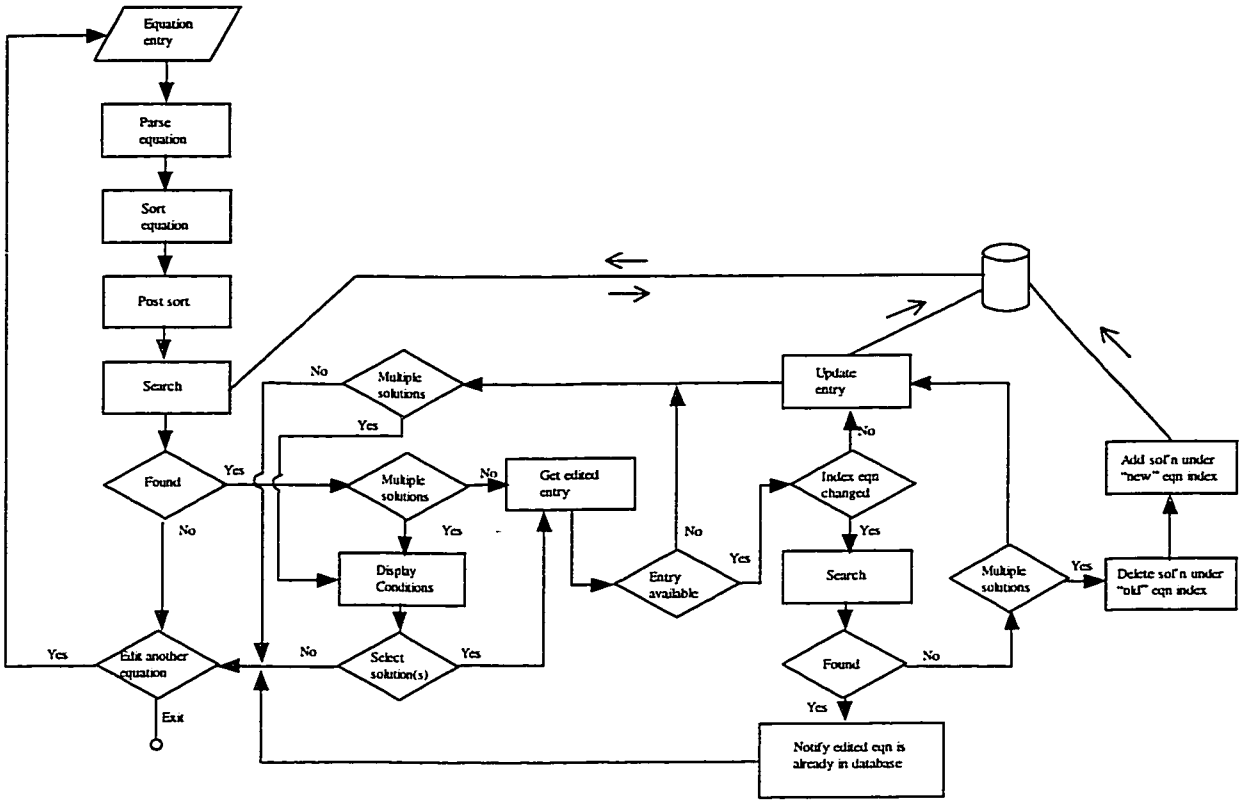


Figure 6.14 The Edit Operation

6.3.3.8 Delete

If an input equation to be deleted is found in the database, the Delete module will remove the rows associated with its entry. The index equation and all its related fields are removed for single solution equations. For multiple solution equations, the index remains unless all of its solutions are to be deleted. In the case of multiple solutions, a dialogbox similar to the one in Figure 6.15 is activated to query the user on the solutions to delete.

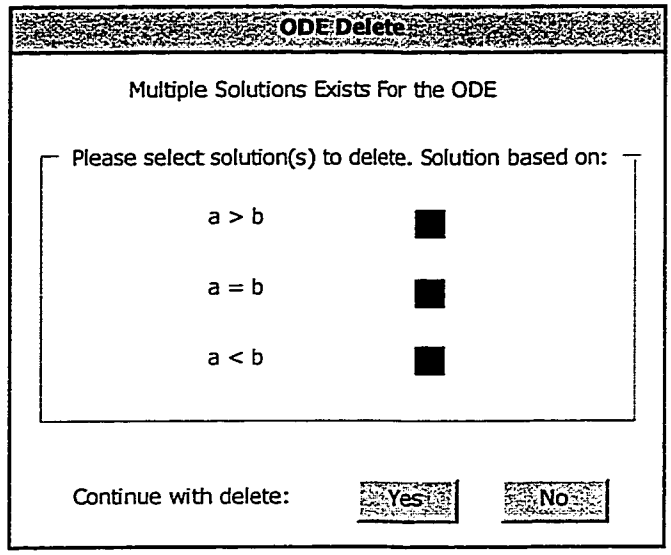


Figure 6.15 Multiple solutions dialogbox for Delete

Like the Edit multiple solution dialogboxes, this dynamic dialogbox is updated with the particular condition sets and display variables pertaining to the query. If no solutions are to be deleted or if the input equation is not found, the user is queried as to whether another equation is to be entered. The control flow diagram for the delete process is shown in Figure 6.16

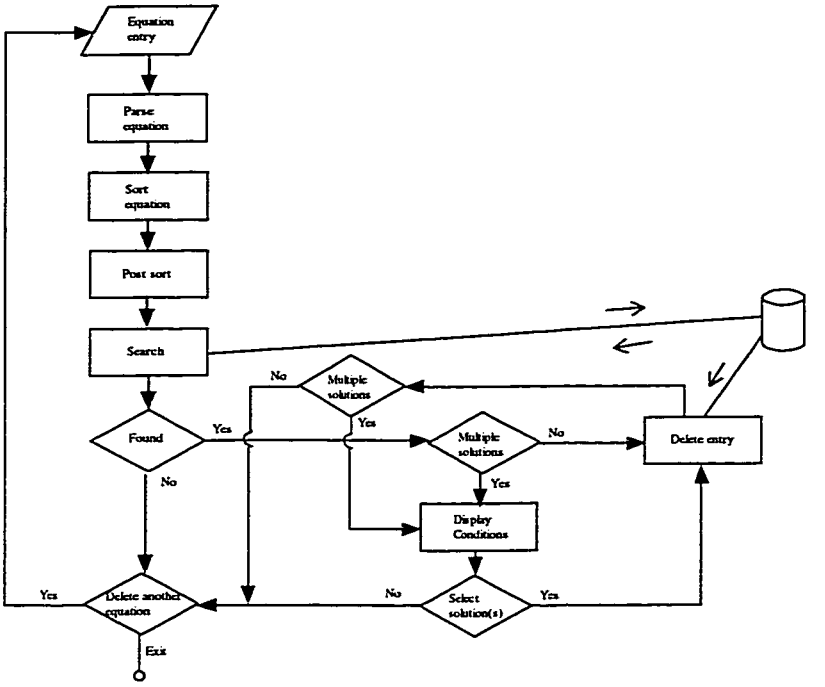


Figure 6.16 The Delete Operation

6.3.3.9 Display

The Display module accepts input from the Edit and Search modules. From the Edit module, it obtains the database location of an index and up to one parameter set while the Search module will send it a location with any number of parameter sets (including none). The location information indicates the database sheet and cell where the index can be found. If multiple solutions exist, the parameter conditions are used to target a particular solution within the solution set of the index equation. The primary purpose of Display is to extract the specified database entry and convert the internal representation to the display variables by consulting the Variable List.

For edit, the Display module brings on the Edit Form shown in Figure 6.17. The original input equation is repeated at the top as a reminder to the user since it may appear quite different from its sorted version in the index equation. The existing information from the database populates the highlighted entry bars which may be edited by the user. When an edit is complete, indicated by a click on the "Edit Complete" button, the entry is re-converted into the internal representation before being sent back to the Edit module for

further processing. A click on the "Edit Cancel" button dismisses the Edit Form and returns an empty form to the Edit module.

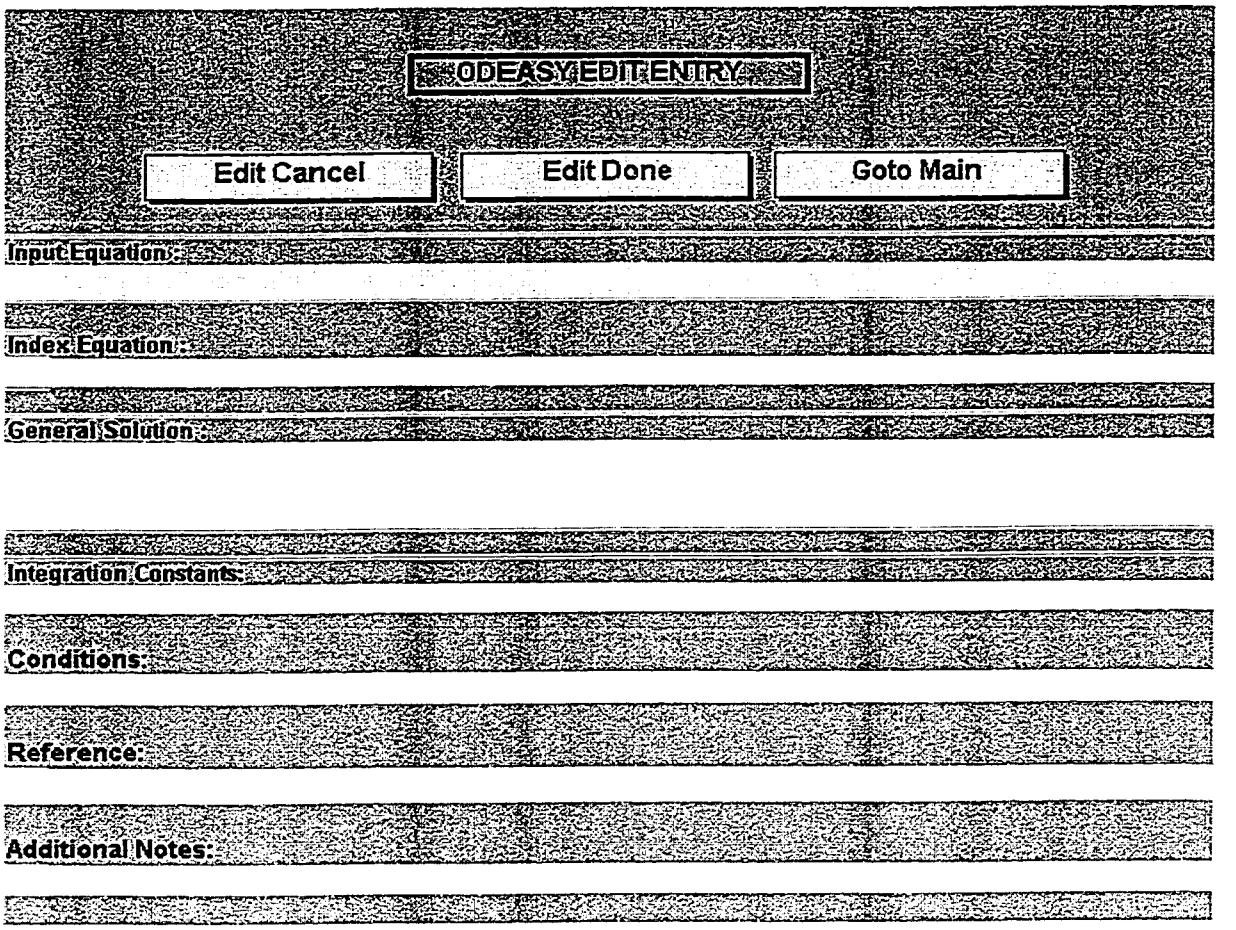


Figure 6.17 The Edit Entry Form

For search, the Display module presents the Search Display Form shown in Figure 6.6-Figure 6.8. Currently under development is an additional menu button for initializing and setting parameters. This feature would permit the user to enter or compute parameter values and to have these values replace the parameters in the solution. A dialogbox queries the user on the value or basic arithmetic formula of each parameter. The arithmetic formula may compute a value with another parameter with widely known constants such as Pi or e. For solutions of first order equations, the entire solution may be solved. An efficient method for this calculation is to convert the equation into postfix form via a pushdown stack and then to re-use the stack to compute the postfix equation.

The infix to postfix conversion is straightforward as indicated by the following pseudo-code:

```
Initialize the stack;
Repeat
   Skip blanks; Read char;
   Case char
        ")" :  write(pop);
        "+", "-", "*", "/", "^" :  push(char);
        else :  prompt for the value of the operand;
                write the operand;
   EndCase;
   If char<>"(" then write (' ');
Until end of equation
```

Parentheses, which are not required in the postfix form, are eliminated. Each right parenthesis indicates that both arguments for the last operator have been output, so the operator itself can be popped and written out. Operands that are functions may require that an additional procedure be provided to the user to compute the function values. The calculations on the postfix string is performed on the stack by pushing down operands and having operators replace the top two items on the stack by the result of the operation as follows:

```
Intialize the stack;
Repeat
   x <- 0;
   Skip blank;  Read char;
   Case char
        "*" :  x <- pop * pop;
        "+" :  x <- pop + pop;
```

```
                :
            other operators
        EndCase;
        Push(x);
    Until end of equation;
    Writeln(pop);
```

While it may seem incongruous to incorporate computation abilities into the Display module, the display of the solution is causal to the requirement for this feature. It involves additional dialogue with the user based upon the result information that is currently on the screen.

6.4 The ODEASY Prototype

The following areas have been implemented in the ODEASY prototype:

- the ODEASY interface with the main menu (Figure 6.2)
- the point-and-click ODEASY Entry Form with built-in dialogboxes for transcendentals and constants for flexible equation entry (Figure 6.3)
- The Parser for error-checking the input, obtaining the name of the independent variable (for equations entered in y' format), changing the equation into y' format, moving all terms to the LHS ensuring uniform spacing, extracting function names from the equation string, mapping the state variables and state variable functions to their internal database representation, and updating the Variable List and Constants List (§6.3.3.2)
- the 2-phase sorting algorithm with the multiplication phase followed by the addition/subtraction phase to convert the equation into the canonical form defined by the sort and recognized by the database (§6.2)

- the PostSort module to map constants to their internal database representation following the sort and to update the constants section of the Variable List (§6.3.3.4)
- the Add module to add ODEs and their solutions into the database (Figure 6.10-6.12). This includes accepting multiple solutions for the same ODE provided each solution is associated with a unique set of conditions (e.g., initial conditions, parameters, range of the independent variable).
- the Search module to search for solutions of ODEs. This includes finding multiple solutions to an ODE (Figure 6.6-6.9).
- the Display module to re-map the database variables to those used by the user by consulting the Variable List (§6.3.3.9)

The source code for the implementation is available in Appendix V. The ODEASY software system has been used to populate the database with 83 ODEs: 5 first order linear, 37 first order non-linear, 25 second order linear, 8 second order non-linear, and 8 third order linear. Six of the ODEs have multiple solutions. These equations, their solutions, and associated conditions are listed in Appendix II.

6.5 Validation of the ODEASY Prototype

In the development of the Parser, the Sort, and the PostSort Modules, 118 equations and equation components have been used to exercise the code and ensure its correct behavior. A list of these testcases is provided in Appendix II. Code coverage is complete with the following types of equations and equation components:

- equations without a derivative (flagged as error)
- equations with irregular spacing, with and without the "=" sign, all or some of the terms on the LHS, in Dy/Dt, in y' format, mix of Dy/Dt and y' format
- selection of an independent variable name which is the same as the dependent variable name (flagged as error)

- use of randomly selected variables to represent the dependent variable, independent variable, function of the dependent variable, function of the independent variable, and constants in the input equations
- use of the database's dependent variable (y), function of the dependent variable (f), function of the independent variable (g), to represent the independent variable in the input equations
- use of the database's independent variable (t), function of the dependent variable (f), function of the independent variable (g), to represent the dependent variable in the input equations
- use of the database's independent variable (t), dependent variable (y), function of the dependent variable (f), function of the independent variable (g), to represent constants in the input equations
- equations with multiple occurrences of nested parentheses, parenthesized addition terms, parenthesized multiplication terms, mix of addition and multiplication elements within parentheses, negative terms, and negative parenthesized terms
- equations with nested division terms, parenthesized numerator with nested parenthesized terms, parenthesized denominator with nested parenthesized terms, division terms raised to an exponent, numerators raised to an exponent, and denominators raised to an exponent
- equations with nested exponent terms, parenthesized exponent base with nested parenthesized terms, parenthesized exponent with nested parenthesized terms, division elements in exponent base, and division elements in exponent
- equations with transcendental functions of the dependent variable, of the independent variable, of a mix of state variables and constants, and of nested transcendental functions

- equations with transcendental functions raised to an exponent, and transcendental functions with arguments raised to an exponent

Each valid testcase is verified to have its variables exchanged for the internal database equivalents and then converted to the canonical form defined by the sort. Each of these testcases are then re-entered with some modifications in the sequence of the elements and the variables used while maintaining the semantics of the original form and checked again to ensure that the same canonical form and variable representation is generated. The correct result is achieved in all cases.

In validation of the complete system, the use of ODEASY to add equations and solutions to its database exercises the software components for the interface, the ODE Entry Form, the Parser, the Sort, the PostSort, and the Add Modules. Eighty-three ODEs and their solutions, as listed in Appendix II, have been added in this way.

Ten of the ODEs have multiple solutions and are used to verify the capability of the Add function to accept multiple solutions for the same ODE. These solutions are entered with their associated conditions. To generate the error case, some of the solutions are entered with the condition set of an existing solution for the ODE in the database to ensure that the system rejects such a solution and notifies the user of the error. In adding each solution, the source reference for the solution is entered via the Additional information dialogbox (Figure 6.11). The solutions in the database sheets are compared for accuracy with the source references. Also, the database sheets are checked to verify that the variables in the solution, the conditions, and the integration constants have been exchanged for their database representation consistent with the mapping of variables performed on the input equation and as reflected in the Variable List.

After the equations and solutions have been added, ODEASY is queried for solutions to exercise the interface, the Parser, the Sort, the PostSort, the Search, and the Display Modules. This includes querying for ODEs not in the database to generate the notification to the user that the ODE cannot be found. Each ODE in the database is queried to verify that the Parser, Sort, and PostSort converts the input ODE into the canonical form stored in the database. While maintaining the semantics of the database

equation, the input ODE is entered using a mix of different formats and sequences in terms, additional parentheses, and different characters to represent the state variables and constants. For half of the equations, the input ODE is entered in 2 different forms to test the robustness of the conversion process. A sample of the database ODEs and the equations used to query them is listed below:

Database ODE : y' – a * t^(b) * y

Query Equations: -t * (p * y^(q)) + t'

f' = (g^(t)) * f * a

Database ODE : y' – t * cos(t)/(a * y^(b) + c)

Query Equations: x' – cos(t)/(d + x^(f) * y) * t

Dy/Dt= g * cos(g)/(y^(m) * n + p)

Database ODE : y''' + (a – b) * y'' + c * y' – (a * b + c) * b * y

Query Equations: Dy/Dt * c – y * b * (c + a * b) + (a – b) * D^2y/Dt^2 + D^3y/Dt^3

((z – u) * g'') + g''' – u * (h + (z * u)) * g + (h * g')

Database ODE : y'' + (a * t + b) * (cos(c * t))^(d) * y' - a * y * (cos(c * t))^(d)

Query Equations: u' * (cos(j * v))^(k) * (h * v + i) + u'' – (cos(j * v))^(k) * h * u

-x * a * (cos(y * c))^(m) + (b + a * y) * (cos(c * y))^(m) * x' + x''

Database ODE : y''' + a * (b * t^(2) + c * t + d) * y'' – 2 * a * b * y

Query Equations: - t * f * m * 2 + ((y^(2) * m) + r + (y * n)) * t'' * f + t'''

(h'' * f) * (r + (n * t) + m * t^(2)) + h''' – f * h * 2 * m

Database ODE : y' + a * t^(b) – c * y^(2) * sin(c * t) – a * t^(b) * y * cos(c * t)

Query Equations: y * x^(t) + b' – sin(s * x) * s * b^(2) – b * cos(x * s) * (y * x^(t))

- j * cos(y * v) * v^(x) * u + v^(x) * j + u' – y * sin(y * v) * u^(2)

For ODEs with multiple solutions, the "Next Sol'n" and "Prev Sol'n" buttons are verified to be enabled and capable of refreshing the screen with the next and previous solutions to the ODE. The Display Module is verified to be behaving correctly by the display of the solutions, integration constants, and conditions in the variables that have been used in the input equation, i.e. the database variables have been re-mapped to their semantic equivalents in the input equation.

6.6 ODEASY As A Supporting Tool

As a supporting tool to other systems, ODEASY has the potential of increasing their problem domain as well as their ability to actually solve a problem if a reduction in problem size permits a solution that is otherwise not easy to obtain. This is especially relevant with complex problem classes that involve higher order nonlinear equations that are difficult for traditional methods to manage. In this way, ODEASY can extend both the scope and power of other systems.

Systems that interact with ODEASY will require access to the PC Windows-based platform. A version of ODEASY, say ODE_XCHG, is needed which bypasses the user interface in favour of a data exchange mechanism with other systems. In order not to invoke the auto-start user interface of ODEASY, ODE_XCHG is an Excel workbook with dynamic spreadsheet links to the database of ODEASY. It is intended strictly as a query tool for other systems and cannot be used to perform database updates. It employs the same sorting algorithm as ODEASY to perform the search on its linked database.

The mechanism of data exchange should be as generic as possible to improve the accessibility of ODEASY. Options for interactions between Windows applications include using the Sendkeys command to send keystrokes directly to another application, the dynamic data exchange (DDE), and object linking and embedding (OLE). OLE is considered the next advancement for Windows applications to communicate but currently very few applications support the OLE standard. DDE, the predecessor of OLE, has wider support but tends to take up a considerable amount of memory. Lastly, Sendkeys is insufficient for actually performing the data passing between applications. Given that the

amount of data to be passed, which could include parameter conditions in the future, makes it inconvenient to use a DDE channel, a storage file serves as a better medium of exchange. A text file format is generic enough for all applications. The system performing the query, the client, must follow a specified file format in terms of the location and sequence of the query equation and any associated conditions. Once invoked, ODE_XCHG extracts the query from the file, performs the search, and rewrites the file according to the specified format. The invocation by the client can either be by the shell command or through the DDE.

An example of a possible file format is provided in Appendix IV.

7. CONCLUSION

Methodologies available to solve ODEs have deficiencies that access to a database of analytical solutions such as ODEASY can either eliminate or alleviate if the analytical solution exists. Obtaining an analytical solution avoids the time and skill required in algebraic derivation and in numerical solution, it can be used to remove equations from the computation that would otherwise incur truncation error, round-off error, or exhibit instability. This is particularly beneficial for difficult to solve higher order equations as well as nonlinear equations which require variable step methods that have a greater tendency for instability for chosen step sizes. Stiffness also, is avoided if the system can be answered analytically. ODEASY can reduce the problem size so that parallel methods are expedited. In simulation studies, the exact solution presented by ODEASY for a submodel will not contribute inaccuracy or compound inaccuracies that result from the numerical processing of the remainder of the model. Mathematical software such as Maple and Mathematica are weak in dealing with higher order ODEs. ODEASY, which can handle higher order ODEs in the same manner as lower order, can be a supporting tool to these and other similar products. Likewise, for Eve and ODEXPERT, the experimental knowledge-bases with limitations in their ability to classify a system, ODEASY can be used to extend their repertoire. By increasing the scope of other systems, ODEASY can be an aid to their development as complete problem solving environments.

Several possibilities for matching a semantically identical ODE that is presented in dissimilar forms were explored in order to determine a method to access the ODEASY database. A two-phase sorting algorithm has been developed to place the elements of an ODE into a canonical form recognized by the database. The first phase employs priorities customized to sorting multiplication terms while the second phase employs priorities for addition and subtraction terms. The customization permits the sort to sequence the equation according to the conventional representation of each operator type. The algorithm, which sorts according to the laws of commutativity and associativity for

addition and multiplication, also simplifies by removing extraneous parentheses. The sorted equation becomes the index into the database.

The interface of ODEASY is a menu-driven point-and click system that permits the use of any variable to represent the state variables and constants. It offers input in Dy/Dt format as well as y' format. The system is able to accept an ODE in any format and sequence for query. Where multiple solutions exist depending upon the values of parameters and initial conditions, ODEASY accepts each unique solution along with the condition(s) for which it is valid. A database query finds all available solutions for an ODE if the ODE exists in the database.

The sorting algorithm has been implemented. Also implemented is the user interface for equation entry, the addition of an ODE and solution(s), and the presentation of search results. The database is populated with 83 ODEs, some with multiple solutions. Further investigation into how ODEASY may be interconnected with other systems is required. The processing for the distribution law that would permit greater simplification of terms is another arena to delve into. The system will also benefit from the addition of an integrator tool to solve solution subparts that are given in integral form. Currently, the development of the interface and the sort has laid the critical foundation for the viability of the tool and future works to complete and enhance the system will increase the ability of ODEASY to contribute in various problem-solving domains.

References

Abell, M.L. and Braselton, J.P. (1994). Differential Equations with Maple V. Academic Press, Inc., San Diego, CA.

Abell, M.L. and Braselton, J.P. (1997). Mathematica by Example, 2nd ed. Academic Press, Inc., San Diego, CA.

Abou-Rabia, O., Birta, L.G., and Chen, M. (1989). A Comparative Evaluation of the BPC and PPC Methods for the Parallel Solution of ODEs. Transactions of the Society for Computer Simulation, Vol.6, No. 4, pp. 265-290.

Adamy, M., Lalis, I. and Liska, M. (1996). Binding the Simulator with the Spreadsheet. 8th European Simulation Symposium, ESS '96, Vol. 2, pp.334-338.

Baras, P., Blum, J., Paumier, J., Witomski, P. and Rechenmann, F. (1992). EVE: An Object-Centered Knowledge-Based PDE Solver. In Houstis et al., editors (1992), Expert Systems for Scientific Computing, pp. 1-18, North-Holland, Amsterdam.

Birta, L.G., and Abou-Rabia, O. (1987). Parallel Block Predictor-Corrector Methods for ODEs. IEEE Transactions on Computers, Vol. C-36, No. 3, pp. 299-311.

Birta, L.G., and Cheng, P. (1995). Parallel Runge-Kutta and Predictor-Corrector Methods for Continuous System Simulation. Proceedings of the 1995 Summer Computer Simulation Conference, pp. 137-142.

Brauer, F. and Nohel, J.A. (1986). Ordinary Equations with Applications. Harper and Row, New York, N.Y.

Buehrer, R.E. et al. (1982). The ETH-Multiprocessor EMPRESS: A Dynamically Configurable MIMD System. IEEE Transactions on Computers, Vol. C-31, pp. 1035-1044, Nov.

Burkhardt, W. (1994). First Steps in Maple. Springer-Verlag, Berlin Heidelberg.

Burrage, K. (1995). Parallel and Sequential Methods for Ordinary Differential Equations. Oxford University Press Inc., New York, N.Y.

Cash, J.R. (1979). Stable Recursions With Applications to the Numerical Solution of Stiff Systems. Academic Press, Inc., London, UK.

Cellier, F.E. (1991). Continuous System Modeling. Springer-Verlag, New York.

Cellier, F.E. and Hilding, E. (1993). Automated Formula Manipulation Supports Object-Oriented Continuous-System Modeling. IEEE Control Systems Magazine, Vol. 13, Issue 2, April '93, pp. 28-38.

Cornea-Hasegan, M., Costian, C., Marinescu, D., Martin, I. and Rice, J. (1994). Towards Problem Solving Environments for High Performance Computing. High Performance Computing '94, National Supercomputer Research Center, Singapore, 1994, pp. 354-366.

Durbin, J.R. (1985). Modern Algebra, 2nd ed. John Wiley and Sons Inc., New York, NY.

Ellis, W. and Lodi, S. (1989). Maple for the Calculus Student: A Tutorial. Brooks/Cole, Pacific Grove, CA.

Fogiel, M. (1986). The Differential Equations Problem Solver. Research and Education Association, New York, NY.

Gallopoulas, S., Houstis, E. and Rice, J. (1992). Future Research Directions in Problem Solving Environments for Computational Science. Center for Supercomputing Research and Development Report No. 1259, University of Illinois at Urbana-Champaign, October 1992.

Gallopoulas, S., Houstis, E. and Rice, J. (1994). Computers as Thinker/Doer: Problem-Solving Environments for Computational Science. IEEE Computational Science and Engineering, Summer 1994, pp. 11-21.

Gear, C.W. (1971). Numerical Initial Value Problems in Ordinary Differential Equations. Prentice Hall, Englewood Cliffs, NJ.

Gear, C.W. (1984). Efficient Step Size Control for Output and Discontinuities. Transactions of the Society for Computer Simulation, 1, pp. 27-31.

Ghasem-Aghaee, N. and Oren, T.I. (1995). KbDif: A Knowledge-Based Approach for Continuous Simulation. Transactions of The Society for Computer Simulation, Vol. 12, No. 4, pp. 287-301.

Ghasem-Aghaee, N., Birta, L.G., and King, D. (1994). Practical Performance of Some Parallel Variable Stepsize ODE Solvers. Proceedings of the 1994 Summer Computer Simulation Conference, pp. 3-8.

Gibbs, N.E., Poole, W.G. and Stockmeyer, P.K. (1976). An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. SIAM Journal of Numerical Analysis, Vol. 13, pp. 236-250.

Gray, J.W. (1994). Mastering Mathematica: Programming Methods and Applications. Academic Press, Inc., San Diego, CA.

Heck, Andre (1993). Introduction to Maple. Springer-Verlag, New York.

Kamel, M., Ma, K.S. and Enright, W. (1992). ODEXPERT: A Knowledge-Based System for Automatic Selection of Initial Value ODE System Solvers. In Houstis et al., editors (1992), Expert Systems for Scientific Computing, pp. 33-54, North-Holland, Amsterdam.

Kahn-Jetter, Z.L. and Sasser, P.A. (1997). Using Spreadsheets for Studying Machine Design Problems Involving Optimization. Computer Applications in Engineering Education, Vol. 5, Iss 3, pp. 199-211.

Kharab, A. (1997). Use of Multiple Sheets for the Solution of a Three-Dimensional Transient Heat Conduction Problem. Computers and Mathematics with Applications, Vol. 34, Iss 1, pp. 71-79.

Kreyszig, E. (1988). Advanced Engineering Mathematics, 6th ed. John Wiley & Sons Inc., New York, N.Y.

Lam, C. Y. (1996). Simulation of the Three-Dimensional Laplace Equation on an Interactive Three-Dimensional Spreadsheet. International Journal of Computer Applications in Technology, Vol. 9, Iss. 5-6, pp.259-271.

Lambert, J. (1991). Numerical Methods for Ordinary Differential Equations. Wiley, New York, N.Y.

Larson, R.E. and Hostetler, R.P. (1986). Calculus with Analytic Geometry, third ed. D.C. Heath and Co., Lexington, MA.

Lee, T. (1993). Mathematical Computations with Maple V: Ideas and Applications: Proceedings of the Maple Summer Workshop and Symposium, University of Michigan, Ann Arbor, MI, June 28-30, Birkhauser, Boston.

Louden, K.C. (1993). Programming Languages: Principles and Practice. PWS Publishing Co., Boston, MA.

Lopez, R.J. (1994). Maple V: Mathematics and its Applications: Proceedings of the Maple Summer Workshop and Symposium, Rensselaer Polytechnic Institute, Troy, NY, Aug. 9-13, Birkhauser, Boston.

Murray-Smith, D.J. (1995). Continuous System Simulation. Chapman and Hall, London, UK.

Ng, H.K. and Crockett, J.R. (1996). Field Monitoring and Reporting With a Spreadsheet: A Low Cost Alternative to the Standard GUI. Proceedings of the 1996 ISA Division Symposia, ISA, Triangle Park, NC, June 3-5, Cleveland, OH, pp. 1-10.

O'Grady, E.P. and Wong, C.H. (1987). Performance Limitations in Parallel Processor Simulations. Transaction of the Society for Computer Simulation, Vol. 4, No. 4, pp. 311-330.

Petzold, L. (1983). Automatic Selection of Methods for Solving Stiff and Non-stiff Systems of Ordinary Differential Equations. SIAM Journal of Scientific and Statistical Computations, Vol. 4, pp. 136-149.

Polyanin, A.D. and Zaitsev, V.F. (1995). Handbook of Exact Solutions for Ordinary Differential Equations. CRC Press, Inc., Boca Raton, FL.

Russell, I.E. and Fatmi, H.A. (1995). Eliciting Long-Term Memory for Neural Network Simulation. Proceedings of the 14th International Congress on Cybernetics, pp. 679-684.

Sedgewick, R. (1989). Algorithms, 2nd ed. Addison-Wesley Publishing Company, Inc, Reading, MA.

Smith, J.M. (1987). Mathematical Modeling and Digital Simulation for Engineers and Scientists. John Wiley and Sons, Inc., New York.

Appendix I

Example of ODE Conversion to Canonical Form

In the example, the MultLvl and Aplvl flags are shown above each column level and the values True and False are represented as T and F, respectively. The sort priority field, with default of zero, is provided on the left of each term. The applicable rule numbers are indicated by "R #" in the comments at the base of each level. The rules are provided in §6.2.1.1 and §6.2.1.2. A new empty node that is created and pointed to is indicated by "<=".

ODE (2):

(y' + y * (t^(1 + a) + b) + ((y * t^(a)) + (t * b))/((t + b) – a)^(2) + 1/(1 – sin(y)) * exp(y))

MultLvl(1)=F, ApLvl(1)=F	MultLvl(2)=T, ApLvl(2)=F	MultLvl(3)=F, ApLvl(3)=T
0 (y' + 0 y	0 y 0 <=	1 (t^
The "+" added to node 1 creates node 2 (R 10). The next "*" promotes "y" to a new mult. level 2 and creates node 2 in level 2 (R 2).	The "(" of the right operand starts level 3 (R 3).	The "(" of the exponent argument starts level 4. Since it is an exponent argument, ApLvl(3) is set T (R 3).
MultLvl(4)=F, ApLvl(4)=F	**MultLvl(3)=F, ApLvl(3)=F**	**MultLvl(2)=T, ApLvl(2)=F**
0 (1 + 0 a)	0 (t^(1 + a) + 0 b)	0 y 0 (t^(1 + a) + b)

The "+" added to node 1 creates node 2 (R 10). The ")" in node 2 creates a complete level 4 subterm which triggers a combine (R 4) and collapse. Since ApLvl(3)=T, collapse is to the last non-empty node and ApLvl(3) is reset (R 7).	The "+" added to node 1 creates node 2 (R 10). The ")" in node 2 triggers a combine (R 4). The balanced subterm result placed in node 1 causes a collapse to level 2 node 2 (R 7).	The node 2 entry of a complete subterm does not induce a sort (R 8). The next "+" triggers a sort on this mult. level and the result, placed in node 1, is appended with the "+" (R 9).
MultLvl(2)=F, ApLvl(2)=F	MultLvl(3)=F, ApLvl(3)=T	MultLvl(4)=F, ApLvl(4)=F
11 (t^(1 + a) + b) * y + 0 <=	0 (	0 (y 0 t^
A new node 2 is created (R 9). Node 1 is updated with highest priority of the sorted elements and MultLvl(2) is reset (R 6). The next "(" starts a new level 3 (R 3).	The next "(" starts a new level 4 and updates ApLvl(3) to T since it is a nested bracket of level 3 (R 3).	The "*" sets MultLvl(4) to T changing level 4 to a mult. level and appends a blank node 2 (R 11). The "(" of the exponent argument begins a new level 5 and sets ApLvl(4) to T (R 3).
MultLvl(5)=F, ApLvl(5)=F	MultLvl(4)=T, ApLvl(4)=F	MultLvl(4)=T, ApLvl(4)=F
0 (a)	0 (y 0 t^(a))	0 y 0 t^(a)
The ")" creates a complete subterm in level 5 which causes a collapse to level 4 (R 7). The collapse of the exponent argument to level 4 node 2 resets ApLvl(4) to F (R 7).	The addition of the 2nd ")" in node 2 creates a complete subterm for the level and causes a bracket reduction prior to a possible sort (R 5). Although ApLvl(3)=T, this is the case of a nested term with excess brackets to be removed.	Since level 3 is not a mult. level and the next operator is not "*", level 4 is sorted and the result placed in node 1 (R 5, 6). The result, a complete subterm, triggers a collapse to level 3 (R 7).
MultLvl(3)=F, ApLvl(3)=F	MultLvl(4)=T, ApLvl(4)=F	MultLvl(4)=T, ApLvl(4)=F
11 (t^(a) * y + 0 <=	0 (t 0 b)	0 t 0 b

The Level 4 subterm is added to node 1 and ApLvl(3) is reset. The node 1 priority number is updated with the priority of the collapsed subterm (R 7). The next "+" is appended to node 1 and creates a new node 2 (R 10).	The parenthesized right operand of addition triggers a new level 4 (R 3). The "*" sets MultLvl(4) to T and appends a new node 2 (R 11). Entry of ")" in node 2 causes a complete subterm for level 4 leading to bracket reduction (R 5).	Following bracket reduction, the list is sorted since level 3 is not a mult. level and the next operator is not "*" (R 5). The sort places the result in node 1 and resets MultLvl(4) to F (R 6).
MultLvl(4)=F, ApLvl(4)=F	MultLvl(3)=F, ApLvl(3)=F	MultLvl(3)=F, ApLvl(3)=F
9 b * t)	11 (t^(a) * y + 9 b * t)	11 (t^(a) * y + b * t)
The sorted expression is initially not collapsed since it is neither a complete nor closing subterm (R 7). The addition of ")" creates a closing subterm causing a collapse to level 3 (R 7).	The collapse of the closing subterm to node 2 triggers a combine of level 3 (R 8).	The combine produces a complete subterm in node 1. The associated priority is updated with the largest of the priorities of the combined terms (R 4). The complete subterm causes a collapse of the term and priority to level 2 (R 7).
MultLvl(2)=F, ApLvl(2)=T	MultLvl(3)=F, ApLvl(3)=T	MultLvl(4)=F, ApLvl(4)=F
11 (t^(1 + a) + b) * y + 11 (t^(a) * y + b * t)/	0 (	0 (t + 0 b)
No further collapse occurs since the level 3 expression is a complete subterm entered in node 2 (R 8). The "(" of the denominator begins a new level 3 and sets ApLvl(2) to T (R 3).	The next "(" begins a new level 4 and sets ApLvl(3) to T (R 3).	Since it is a non-mult. level, the "+" is added to node 1 and a new node 2 is appended (R 10). The ")" in node 2 triggers the combine (R 4) and collapse of the complete subterm to level 3 (R 7).
MultLvl(3)=F, ApLvl(3)=F	MultLvl(3)=F, ApLvl(3)=F	MultLvl(2)=F, ApLvl(2)=T

0 ((t + b)	0 ((t + b) – 0 a)	11 (t^(1 + a) + b) * y + 11 (t^(a) * y + b * t)/((t + b) – a)^
The collapse is to node 1 and ApLvl(3) is reset (R 7). Note that bracket reduction is not performed on non-mult. terms in this phase. Since node 1 is not a complete or closing subterm, no further collapse occur (R 8).	The "-" is added to node 1 and a new node 2 is appended (R 10). The ")" in node 2 triggers a combine (R 4) and collapse of the complete subterm to level 2 (R 7).	The collapse is the concatenation of the denominator expression to node 2 and ApLvl(2) is reset (R 7). No further collapse since it is a complete subterm in node 2 (R 8). The "(" of the exponent argument begins a new level 3 and sets ApLvl(2) to T (R 3).
MultLvl(3)=F, ApLvl(3)=F	MultLvl(2)=F, ApLvl(2)=T	MultLvl(3)=F, ApLvl(3)=T
0 (2)	11 (t^(1 + a) + b) * y + 11 (t^(a) * y + b * t)/((t + b) – a)^(a) + 0 1/	0 (1 – 0 sin
The addition of ")" forms a complete subterm that is collapsed to node 2 of level 2 (R 7). The collapse resets ApLvl(2) to F.	Again, the collapse does not trigger a combine since it maintains a complete subterm in node 2 (R 8). A "+" is added to node 2 and creates node 3 (R 10). The next "(" for the denominator term starts level 3 and sets Aplvl(2) (R 3).	The "-" added to node 1 creates node 2 (R 10). The "(" of the "sin" argument starts a new level 4 and sets ApLvl(3) to T (R 3).
MultLvl(4)=F, ApLvl(4)=F	MultLvl(3)=F, ApLvl(3)=F	MultLvl(3)=F, ApLvl(3)=F
0 (y)	0 (1 – 0 sin(y))	0 (1 – sin(y))
The complete subterm in node 1 is collapsed to level 3 node 2 and ApLvl(3) is reset (R 7).	The addition of ")" in node 2 creates a closing subterm which triggers a combine (R 4).	The combine forms a complete subterm in node 1 which is collapsed to level 2 node 3 (R 7). ApLvl(2) is reset.
MultLvl(2)=F, ApLvl(2)=F	MultLvl(3)=T, ApLvl(3)=F	MultLvl(4)=F, ApLvl(4)=F

11 (t^(1 + a) + b) * y + 11 (t^(a) * y + b * t)/((t + b) – a)^(a) + 0 1/(1 – sin(y))	0 1/(1 – sin(y)) 0 exp	0 (y)
The collapse from level 3 leaves a complete subterm that does not trigger a combine (R 8). The next "*" promotes its left operand, the node 3 term, to node 1 of a new mult. level 3 (R 11).	A new node 2 is appended (R 11). The "(" of the "exp" argument begins level 4 and ApLvl(3) is set T (R 3).	The complete subterm is collapsed to level 3 node 2 and resets ApLvl(3) (R 7).
MultLvl(3)=T, ApLvl(3)=F	MultLvl(3)=F, ApLvl(3)=F	MultLvl(2)=F, ApLvl(2)=F
0 1/(1 – sin(y)) 0 exp(y))	28.8 exp(y) * 1/(1 – sin(y)))	11 (t^(1 + a) + b) * y + 11 (t^(a) * y + b * t)/((t + b) – a)^(a) + 28.8 exp(y) * 1/(1 – sin(y)))
The collapse of the complete subterm into node 2 does not trigger a sort (R 8). The 2nd ")" added to node 2 creates a closing subterm which induces a sort and resets MultLvl(3) (R 5, 6).	The sort results in a closing subterm in node 1. This triggers a collapse to level 2 which includes updating the level 2 node 3 priority with the priority of level 3 (R 7).	The collapse places a closing subterm in node 3 which causes a level 2 combine (R 8).
MultLvl(2)=F, ApLvl(2)=F	MultLvl(1)=F, ApLvl(1)=F	MultLvl(1)=F, ApLvl(1)=F
28.8 (t^(1 + a) + b) * y + (t^(a) * y + b * t)/((t + b) – a)^(a) + exp(y) * 1/(1 – sin(y)))	0 (y' + 28.8 (t^(1 + a) + b) * y + (t^(a) * y + b * t)/((t + b) – a)^(a) + exp(y) * 1/(1 – sin(y)))	(y' + (t^(1 + a) + b) * y + (t^(a) * y + b * t)/((t + b) – a)^(a) + exp(y) * 1/(1 – sin(y)))
The combine places a closing subterm in node 1 that is followed by a further collapse to level 1 (R 7).	Again, the collapse results in a closing subterm in level 1 node 2 which triggers a combine (R 8).	The last combine completes the multiplication phase on this equation.

The multiplication phase changes the equation

ODE (2):

(y' + y * (t^(1 + a) + b) + ((y * t^(a)) + (t * b))/((t + b) – a)^(2) + 1/(1 – sin(y)) * exp(y))

into the following by re-ordering the multiplication terms according to the Table 6.1 priorities and removing superfluous parentheses enclosing multiplication expressions:

ODE (2'):

(y' + (t^(1 + a) + b) * y + (t^(a) * y + b * t)/((t + b) – a)^(a) + exp(y) * 1/(1 – sin(y)))

The example ODE is continued to illustrate the application of the a/s rules. In the example, the ApLvl flag is shown above each column level, and in each row are displayed the 3 fields for the sign, priority and term. As before, the priority is given a default of zero; a new empty node is indicated by "<="; and the applicable rule numbers are indicated by "R #' in the comments at the base of each level. The rules are in §6.2.1.2.

ODE (2'):

(y' + (t^(1 + a) + b) * y + (t^(a) * y + b * t)/((t + b) – a)^(a) + exp(y) * 1/(1 – sin(y)))

ApLvl(1)=F	ApLvl(2)=T	ApLvl(3)=F
0 + (y' 0 + <=	0 + (t^	0 + (1 0 + a)
The start of every parenthesized expression begins with an addition operator in the sign field (R 2). The 2nd "+" appends a new node 2 (R 1). The next "(" starts level 2 (R 2).	The next "(" starts level 3 (R 2) and sets ApLvl(2) to T for the exponent base in node 1.	The 2nd "+" appends a new node 2 (R 1). The closing ")" in node 2 triggers a sort (R 3). Bracket reduction cannot occur for an exponent (R 4).
ApLvl(3)=F	ApLvl(2)=F	ApLvl(2)=F
34 + (a + 1)	0 + (t^(a + 1) 0 + b)	33 + (t^(a + 1) + b)

The complete subterm in node 1 is collapsed to level 2 node 1. Since the subterm contains only constants, the receiving node's priority is not updated. ApLvl(2) is reset (R 5).	The 2nd "+" appends a new node 2 (R 1). Closing ")" in node 2 triggers bracket reduction and sort (R 3). Peek ahead shows this term is a mult. factor so brackets are not removed (R 4).	The sort results in a complete subterm in node 1 that is collapsed to level 1. The empty receiving node is assigned the highest priority of the non-constant terms in the collapsed expression (R 5).
ApLvl(1)=F	ApLvl(1)=F	ApLvl(2)=T
0 + (y' 6 + (t^(a + 1) + b)	0 + (y' 9 + (t^(a + 1) + b) * y 0 <=	0 + (t^
Bracket reduction process finds no excess brackets for exponentiation (R 4) in the collapsed complete subterm (R 5).	The "*" and its operand is appended in node 2. The mult. operand's priority exceeds current node 2 priority of 6 and hence, replaces it (R 1). The next "+" appends node 3 (R 1).	Level 2 is started by the "(" (R 2). The next "(" begins level 3 and sets ApLvl(2) to T for the exponent base in level 2 (R 2).
ApLvl(3)=F	ApLvl(2)=F	ApLvl(2)=F
33 + (a)	9 + (t^(a) * y 7 + b * t)	9 + (b * t + t^(a) * y)
Enclosing brackets cannot be removed since the term is an exponent (R 4). The complete subterm in node 1 is collapsed and Aplvl(2) is reset. The receiving node's priority is not updated by the constant (R 5).	The "*" and its operand is appended in node 1. The higher priority of the operand replaces the zero priority of node 1 (R 1). The 2nd "+" appends node 2 (R 1). Closing ")" triggers a bracket reduction attempt and sort (R 3).	Peek ahead indicates term is a numerator which precludes bracket removal (R 4). Proceeds with sort. Sorted result in node 1 is a complete subterm that is collapsed. The empty receiving node gets the highest priority of the terms in the collapsed result (R 5).
ApLvl(1)=T	ApLvl(2)=T	ApLvl(3)=F

0 + (y' 9 + (t^(a + 1) + b) * y 9 + (b * t + t^(a) * y)/	0 + (	0 + (t 0 + b)
The "^" in the collapsed complete subterm triggers an attempt at bracket reduction for the exponentiation operation (R 4). The "/" is appended in node 3 (R 1) and the next "(" starts level 2 and sets ApLvl(1) to T (R 2).	The next "(" starts level 3 and sets ApLvl(2) to T for the nested term (R 2).	The 2nd "+" appends new node 2 (R 1). The ")" in node 2 causes bracket reduction for the level's complete subterm (R 4).
ApLvl(3)=F	ApLvl(2)=F	ApLvl(2)=F
0 + t 0 + b	0 + (t 0 + b 0 - a)	33 + (t – a + b)
The bracket reduction is not followed by a sort since the level is no longer parenthesized. Instead, the terms are transferred to level 2 beginning in node 1 and Aplvl(2) is reset (R 4).	The "-" appends new node 3 (R 1). The ")" in node 3 triggers attempt at bracket reduction and sort (R 3). Brackets not removed since the level holds a denominator term (R 4). Proceeds with sort.	The sorted result is a complete subterm in node 1 that is collapsed to the last non-empty node of level 1. Aplvl(1) is reset (R 5).
ApLvl(1)=F	ApLvl(1)=T	ApLvl(2)=F
0 + (y' 9 + (t^(a + 1) + b) * y 9 + (b * t + t^(a) * y)/(t – a + b)	0 + (y' 9 + (t^(a + 1) + b) * y 9 + (b * t + t^(a) * y)/(t – a + b)^	34 + (2)
The collapse into node 3 creates a complete subterm in the node. It is checked for superfluous brackets on the exponentiation and division operations (R 4).	The "^" is appended in node 3 (R 1). Next "(" for the exponent sets ApLvl(1) to T and starts level 2 (R 3).	The closing ")" creates a complete subterm in node 1 that cannot be bracket reduced since the term is an exponent (R 4). The term is assigned a priority (R 3) and collapsed to level 1, resetting ApLvl(1) (R 5).

ApLvl(1)=F	ApLvl(1)=T	ApLvl(2)=F
0 + (y' 9 + (t^(a + 1) + b) * y 9 + (b * t + t^(a) * y)/(t – a + b)^(2)	0 + (y' 9 + (t^(a + 1) + b) * y 9 + (b * t + t^(a) * y)/(t – a + b)^(2) 0 + exp	9 + (y)
The collapse continues the complete subterm in node 3. The collapse causes the node 3 term to be processed again for excess brackets on the exponentiation and division operations (R 4).	The next "+" appends new node 4 (R 1). The "(" for the function argument begins a new level 2 and sets ApLvl(1) to T for the function in level 1 (R 2).	Bracket reduction cannot occur for the complete subterm in node 1 since the term is a function argument (R 4). The term is assigned a priority (R 3) and collapsed to level 1 (R 5).
ApLvl(1)=F	ApLvl(1)=T	ApLvl(2)=T
0 + (y' 9 + (t^(a + 1) + b) * y 9 + (b * t + t^(a) * y)/(t – a + b)^(2) 23.8 + exp(y)	0 + (y' 9 + (t^(a + 1) + b) * y 9 + (b * t + t^(a) * y)/(t – a + b)^(2) 23.8 + exp(y) * 1/	0 + (1 0 - sin
The collapse to node 4 updates the node's priority by assigning it the highest priority of the terms in the new expression. In this case, the collapse forms a complete transcendental function. Also resets ApLvl(1) (R 5).	The "*" is appended in node 4 (R 1). The next "(" begins a new level 2 and sets ApLvl(1) to T for the denominator in level 1 (R 2).	The '-' appends new node 2 (R 1). The "(" of the function argument starts a new level 3 and sets ApLvl(2) to T for the function in level 2 (R 2).
ApLvl(3)=F	ApLvl(2)=F	ApLvl(2)=F
9 + (y)	0 + (1 30.9 - sin(y))	34 + (1 – sin(y))

The complete subterm, a function argument, cannot be bracket reduced (R 4). It is assigned a priority (R 3) and collapsed to level 2 (R 5).	ApLvl(2) is reset. Only the term is transferred from level 2 and not the sign (R 2). Node 2 receives a priority based upon its updated expression (R 5). The closing ")" in node 2 triggers a bracket reduction attempt and a sort (R 3).	Although the numeric constant has a higher priority than sin(), the sin() is not sorted to the beginning of the expression since it is a negative term (R 3). Bracket removal not done for denominator term (R 4). Sorted result in node 1 is collapsed (R 5).
ApLvl(1)=F	ApLvl(1)=F	ApLvl(1)=F
0 + (y' 9 + (t^(a + 1) + b) * y 9 + (b * t + t^(a) * y)/(t – a + b)^(2) 30.9 + exp(y) * 1/(1 – sin(y))	5 + (y' 9 + (t^(a + 1) + b) * y 9 + (b * t + t^(a) * y)/(t – a + b)^(2) 30.9 + exp(y) * 1/(1 – sin(y)))	(y' + (t^(a + 1) + b) * y + (b * t + t^(a) * y)/(t – a + b)^(2) + exp(y) * 1/(1 – sin(y)))
The collapse in node 4 is checked for excess brackets on the division operation (R 5). No excess brackets exist. ApLvl(1) is reset.	The final ")" creates a closing subterm which triggers a sort (R 6). Any missing priorities are first filled in.	The identical priorities of nodes 2 and 3 are resolved alphabetically. The a/s phase and the sort is complete.

Appendix II

ODEASY Database Equations

Database Eqns

First Order Linear

Eqn #	Database Equation	Condition	Solution
1	y' - a * y	a <> 0	y = k * exp(a * t)
2	y' - a * y/t	t <> 0	y = 0 y = k * t^(a)
3	y' - a * t^(b) * y	b <> - 1	y = k * exp(a/(b + 1)) * t^(b + 1)
4	y' - a * t^(b)	b <> - 1	y = a/(b + 1) * t^(b + 1) + k
	As Above	b = - 1	y = a * lnlt| + k
	As Above	b = 0	y = a * t + k
5	y' - t - y		t + y + 1 = C * exp(x)

First Order Non-Linear

Eqn #	Database Equation	Condition	Solution
1	y' - a * t/y	y <> 0	y = +/ - (k + a * t^(2))^(1/2)
2	y' - a * t^(b) * y^(c)	c <> - 1	y = [a * (1 - c)/(b + 1) * t^(b + 1) + k]^(1/(1 - c))
	As Above	c = 1	y = k * exp(a/(b + 1)) * t^(b + 1)
3	y' - (a * t^(b) + c)/(d * y^(h) + k)		a/(b + 1) * t^(b + 1) + c * t - d/(h + 1) * y^(h + 1) - f * y = k
4	y' - t * cos(t)/(a * y^(b) + c)	b <> - 1	c * y + a/(b + 1) * y^(b + 1) = t * sin(t) + cos(t) + k
	As Above	b = - 1	c * y + a * log|y| = t * sin(t) + cos(t) + k
5	y' - t * sin(t)/(a * y^(b) + c)	b <> - 1	c * y + a/(b + 1) * y^(b + 1) = sin(t) - t * cos(t) + k
	As Above	b = - 1	c * y + a * log|y| = sin(t) - t * cos(t) + k
6	y' - a * t^(b)/((t^(b + 1) + c)^(d) * y^(h))	h <> - 1	y = [a * (h + 1)/((b + 1) * (1 - d)) * (c + t^(b + 1))^(1 - d) + k]^(1/(b + 1))
	As Above	h = - 1	y = k * exp(a/((b + 1) * (1 - d))) * (c + t^(b + 1))^(1 - d)
7	y' - a * (b * t^(c) + t^(c) * y^(d + 1))/(h * y^(d) + t^(c + 1) * y^(d))		y = [(k * 1/((t^(c + 1) + h)^(- a * (d + 1))))^(1/(c + 1)) - b]^(1/(d + 1))
8	(a * y - t^(2)) * y'^(2) + 2 * t * y * y' - y^(2)		(k * y + t)^(2) = 4 * a * y

9	y' + a * (ln(t))^(b) - a * t^(c + 1) * y * (ln(t))^(b) + (c + 1) * t^(c) * y^(2)	y0 = t^(- c - 1)
10	y' - a * b * t^(b - 1) + a^(2) * c * t^(2 * b) * (ln(t))^(d) - c * y^(2) * (ln(t))^(d)	y0 = a * t^(b)
11	y' - a * (ln(t)) - b * y^(2) * (ln(t))^(c) + b * a * t * y * (ln(t))^(c + 1) - a	y0 = a * t * ln(t)
12	y' - a * b * t^(b - 1) - c * (y - a * t^(b) - d)^(2) * (ln(t))^(h)	y0 = a * t^(b) + d
13	y' - a * b * (ln(t))^(c) + b^(2) * d * (ln(t))^(h) - d * y^(2) * (ln(t))^(h) - a * y * (ln(t))^(c)	y0 = - b
14	t * y' + a^(2) * t * (ln(b * t))^(2) - t * y^(2) - a	y0 = a * ln(b * t)
15	t * y' - a * b * (ln(c * t))^(b - 1) + a^(2) * t * (ln(c * t))^(2 * b) - t * y^(2)	y0 = a * (ln(c * t))^(b)
16	t * y' + a^(2) * b * t^(c) * (ln(t))^(2) - b * t^(c) * y^(2) - a	y0 = a * ln(t)
17	y' + a * b * ln(c * t) - y^(2) - a * y * ln(c * t) + b^(2)	y0 = b
18	y' - a * (ln(b * t))^(c) - y^(2) - a * t * y * (ln(b * t))^(c)	y0 = - (1/t)
19	y' - a * ln(t) - b * t^(c) * y^(2) + b * a * t^(c + 1) * y * ln(t) - a	y0 = a * t * ln(t)
20	y' - a * b * sin(c * t) - y^(2) - a * y * sin(c * t) + b^(2)	y0 = - b
21	y' - a * (sin(b * t))^(c) - y^(2) - a * t * y * (sin(b * t))^(c)	y0 = - (1/t)
22	y' - (a - b) * b * (tan(a * t))^(2) - y^(2) - a * b	y0 = b * tan(a * t)
23	y' - y^(2) - a^(2) - (a - b) * b * (tan(a * t))^(2) - 3 * b * a	y0 = b * tan(a * t) - a * (tan(a * t))^(- 1)
24	y' - a * b * sin(b * t) - a^(2) * (sin(b * t))^(2) - y^(2) + a^(2)	y0 = - a * cos(b * t)
25	y' - a * b * cos(b * t) - a^(2) * (cos(b * t))^(2) - y^(2) + a^(2)	y0 = a * sin(b * t)
26	y' - a * (sin(a * t))^(3) - a * y^(2) * sin(a * t)	y0 = - cos(a * t)
27	y' - a * (cos(a * t))^(3) - a * y^(2) * cos(a * t)	y0 = sin(a * t)
28	y' - a * (cos(b * t))^(2) - y^(2) * (a * (cos(b * t))^(2) + b) + a - b	y0 = tan(b * t)
29	y' + a * (cos(b * t))^(c - 1) - b * y^(2) * sin(b * t) - a * y * (cos(b * t))^(c)	y0 = 1/cos(b * t)
30	y' + a * tan(b * t) - b * y^(2) * sin(b * t) - a * y * sin(b * t)	y0 = 1/cos(b * t)

31	y' + a * t^(b) - c * y^(2) * sin(c * t) - a * t^(b) * y * cos(c * t)	y0 = 1/cos(c * t)
32	y' + a * (sin(t))^(b) - a * t^(c + 1) * y * (sin(t))^(b) + (c + 1) * t^(c) * y^(2)	y0 = t^(- c - 1)
33	y' + a * (tan(t))^(b) - a * t^(c + 1) * y * (tan(t))^(b) + (c + 1) * t^(c) * y^(2)	y0 = t^(- c - 1)
34	y' - a * b * t^(b - 1) - c * (y - a * t^(b) - d)^(2) * (sin(h * t + k))^(m)	y0 = a * t^(b) + d
35	y' - a * b * (tan(b * t))^(2) + a^(2) * c * (tan(b * t))^(d + 2) - c * y^(2) * (tan(b * t))^(d) - a * b	y0 = a * tan(b * t)
36	y' - a * b * t^(b - 1) - c * (y - a * t^(b) - d)^(2) * (tan(h * t + k))^(m)	y0 = a * t^(b) + d
37	y' - a * ln(t) - y^(2) * g(t) + a * t * y * g(t) * ln(t) - a	y0 = a * t * ln(t)

Second Order Linear

Eqn #	Database Equation	Condition	Solution
1	y" + (a * t^(3) + 2 * b) * y' + (a * b * t^(3) - a * t^(2) + b^(2)) * y		y0 = t * exp(- b * t)
2	t * y" + a * y' + b * t^(1 - 2 * a) * y	b > 0	y = c1 * sin((b^(1/2)/(a - 1) * t^(1 - a)) + c2 * cos((b^(1/2)/(a - 1) * t^(1 - a))
	As Above	b < 0	y = c1 * exp((- b^(1/2))/(a - 1) * t^(1 - a)) + c2 * exp((- (- b^(1/2)))/(a - 1) * t^(1 - a))
3	t * y" + (1 - 3 * a) * y' - b^(2) * a^(2) * t^(2 * a - 1) * y		y0 = (b * t^(a) + 1) * exp(- b * t^(a))
4	t * y" + a * y' + b * (-b * t^(c + 1) + a + c) * t^(c) * y		y0 = exp(-b/(c + 1) * t^(c + 1))
5	t * y" + a * t * y' + a * y		y0 = t * exp(- a * t)
6	y" + (a * t^(b) + c) * y' + d * (a * t^(b) + c - d) * y		y0 = exp(- d * t)
7	y" + (a * t^(b) + 2 * c) * y' + (a * c * t^(b) - a * t^(b - 1) + c^(2)) * y		y0 = t * exp(- c * t)
8	y" + (a * b * t^(c) + b * t^(c - 1) + 2 * a) * y' + a^(2) * (b * t^(c) + 1) * y		y0 = (a * t + 1) * exp(- a * t)
9	y" + (2 * a * t^(b - 1) + c * a * t^(b) - c^(2) * t) * y' + c * (c * a * t^(b) + a * t^(b - 1) - c^(2) * t) * y		y0 = (a * t + 2) * exp(- a * t)
10	y" + (a * t^(2) + (a * b + c) * t + c * b) * t^(d) * y' - t^(d) * (a * t + c) * y		y0 = t + b
11	y" + (a * t^(b) + c * t^(d)) * y' - (a * t^(b - 1) + c * t^(d - 1)) * y		y0 = t
12	y" + (a * t^(b) + c * t^(d)) * y' + ((b + 1) * a * t^(b - 1) + (d + 1) * c * t^(d - 1)) * y		y0 = t * exp(- a/(b + 1) * t^(b + 1) - c/(d + 1) * t^(d + 1))

13	y" + (a * t^(b) + c * t^(d)) * y' + h * (a * t^(b) + c * t^(d) - h) * y	y0 = exp(- h * t)
14	y" + (a * t^(b) + c * t^(d)) * y' + (a * c * t^(d + b) - a * t^(b - 1) + (d + 1) * c * t^(d - 1)) * y	y0 = t * exp(- c/(d + 1) * t^(d + 1))
15	y" + (a * t^(b) + c * t^(d) + h) * y' + (a * c * t^(d + b) + a * b * t^(b - 1) + c * h * t^(d)) * y	y0 = exp(- a/(b + 1) * t^(b + 1) - h * t)
16	y" + (a * t + b) * (cos(c * t))^(d) * y' - a * y * (cos(c * t))^(d)	y0 = a * t + b
17	t * y" + (a * t^(b) * ln(t) + 1) * y' - a * t^(b - 1) * y	y0 = ln(t)
18	t * y" + a * t * ln(t) * y' + a * y * (ln(t) + 1)	y0 = exp(a * t) * t^(1 - a * t)
19	y" + a * (t^(2) - b^(2)) * y' - a * (t + b) * y	y0 = t - b
20	y" + (a * t^(2) + b) * y' + c * (a * t^(2) + b - c) * y	y0 = exp(- c * t)
21	y" + (a * t^(2) + 2 * b) * y' + (a * b * t^(2) - a * t + b^(2)) * y	y0 = t * exp(- b * t)
22	y" + (a * t^(2) + b * t) * y' + (3 * a * t + 2 * b) * y	y0 = t * exp((- 1/3) * a * t^(3) - (1/2) * b * t^(2))
23	y" + (a * b * t^(3) + b * t^(2) + 2 * a) * y' + a^(2) * (b * t^(3) + 1) * y	y0 = (a * t + 1) * exp(- a * t)
24	t^(2) * y" + (a * t^(2) + b) * y' + g((a - c) * t^(2) + b) * y	y0 = exp(- g * t)
25	t^(2) * y" + (a * t^(2) + b * t) * y' - b * y	y0 = t^(- b) * exp(- a * t)

Second Order Non-Linear

Eqn #	Database Equation	Condition	Solution
1	y" - a * t^(-5/2) * y^(-1/2)		t = a * C1^(- 3) * (T^(3) - 3 * T + C2)^(- 1) y = b * C1 * (T^(2) - 1)^(2) * (T^(3) - 3 * T + C2)^(- 1), a = +/- (4/9) * a^(1/2) * b^(3/2)
2	y" - a * t * y^(-5/3)		t = +/- a * C1^(8) * (T^(4) - 6 * T^(2) + 4 * C2 * T - 3) y = b * C1^(9) * (T^(3) - 3 * T + C2)^(3/2), a = +/- (9/64) * a^(-3) * b^(8/3)
3	y" - a * t^(2/3) * y^(-5/3) - (3/4) * t^(-2) * y		t = C1 * (T^(3) +/- 3 * T + C2)^(1/2) y = b * C1 * (T^(2) +/- 1)^(3/2) * (T^(3) +/- 3 * T + C2)^(-1/4), a = +/- (4/3) * b^(8/3)
4	y" + a * y'^(2) + b * y		t = c + / - a * ∫[d * a^(2) * exp(- 2 * a * y) + b * (1/2 - a * y)]^(- 1/2)dy
5	y" - (a * y^(b) + c * y^(-1)) * y'^(d)	d <> 2 b <> - 1	t = ∫[C1 + (a * (2 - d)/(b + 1)) * y^(b + 1) + (2 - d) * c * ln(y)]^(1/(d - 2))dy + C2
6	(a^(2) - t^(2)) * (b^(2) - y^(2)) * y" + (a^(2) - t^(2)) * y * y'^(2) - t * (b^(2) - y^(2)) * y'		arcsin(y/b) = C1 + C2 * arcsin(t/b)

7	y'' - a * exp(t + y) * (y' + 1)	y = - ln(C1 * exp(- C2 * t) - (a/(1 + C2)) * exp(t))
8	y'' - a * t^(b) * exp(y) * y' - a * b * t^(b - 1) * exp(y)	y = C1 * t - ln[C2 - a * ∫t^(b) * exp(C1 * t)dt]

Third Order Linear

Eqn #	Database Equation	Condition	Solution
1	t^(3) * y''' + t^(2) * arctan^(a)(t) * y'' + t * (arctan^(a)(t) - 1) * y' + y * (arctan^(a)(t) - 3)		y1 = cos(ln(t)) y2 = sin(ln(t))
2	y''' - a * (b * t + c) * t * y'' + (c - b^(2)) * a * y' + 2 * a * b * y		y0 = t^(2) + b * t + (1/2) * (b^(2) - c)
3	y''' - (a * (t^(2) + b * t + c) + d * (2 * t + b)) * y'' + 2 * d * y' + 2 * a * y		y0 = t^(2) + b * t + c
4	t * y''' + 3 * y'' + a * (b * t^(2) + 1) * t * y' - a * (b * t^(2) - 1) * y		y0 = a * t + (1/t)
5	y''' + (a - b) * y'' + c * y' - (a * b + c) * b * y		y0 = exp(b * t)
6	y''' + (a + b) * y'' + (a * b + c) * y' + c * b * y		y0 = exp(-b * t)
7	y''' + a * (b * t + c) * y'' + a * t * y' - 2 * a * y		y0 = t^(2) + 2 * b * t + c
8	y''' + a * (b * t^(2) + c * t + d) * y'' - 2 * a * b * y		y0 = b * t^(2) + c * t + d

Appendix III

ODEASY Testcases

t1 = " g"+ sin(t) - 2*g'=-tan(f /y)-y*t+g*exp(t)-g*y *sin^(2)(g)"
t2 = "f = tan(t)-csc^2(t)-h^2(f)*f'-11 *y+k(t)*f"*y-k(t)+k(t)^m-h(f)*k(t)"
t3 = "f' - tan^3(t)+ csc(t)+f" +h(t)=k(f)-h^(1+t)(t)*k^2(f)-k(f)"
t4 = "k^(c)(y)+t'+ b * t =8*y-tan(t)-exp(t)*t" -12*exp((t+1)^b)"
t5 = "b*t+t"-1+ ln(y)+t*exp(b^(1+y))+f(y)-g(t)+f(y)*g(t)=0"
t6 = "y' + exp(t * b)/y^(a) - b"
t7 = "a' - (a + (b/a) * t^n + m * (t^p * 2 * b/a^2) - (8 * b)/a^3)"
t8 = "a^(3)+g-sec(g)*f' + 2*f" +f""*sin(a)+h(g)"
t9 = "j' + (a * ln(k) * (j/k))/(m + b) = -k"   't=k
t10 = "y' * exp(t * b)/y^c - b"
t11 = "y' + (c^(y) + g(t))/(t * exp(y) - f(y))"
t12 = "y' + (c/t) * t^(-b)/(a * sin(cos(a)) * t^(b - 1))"
t13 = "y' + (y + c/a)"
t14 = "y' + (sin(y) * cos(t))^y"
t15 = "y' - ((t/m + b)^(csc(t) + y))"
t16 = "y' - (a * ln(t) * (y^t))^(m + b)"
t17 = "y' * (exp(t * b)^y)/c - b"
t18 = "y' + (c/(y) + g(t))^(t * exp(y) - f(y))"
t19 = "y' + t/(-b)^(a * sin(cos(a)) * t/(b - 1))"
t20 = "y' + (y + c^a)"
t21 = "y' + (t/2 + b)/(-a^2)"
t22 = "y' + a * (d^(a * c)) + b"
t23 = "y' + exp(t) + ((c - b)^(b - a)/(c - 2)^(b - a)) * exp(t)"
t24 = "y' + exp(t * (b - a)) + ((c - b)^b/(c - b)^b) * exp(t)"
t25 = "y' + ((a + b)/2)^(a + b)"
t26 = "y' + (a + b)/(2^(a + b))"
t27 = "y' + (a + b)/(2)^(a + b)"
t28 = "y' + exp(t * (b - a)) + (c^(b - a)/(c - b)^b) * exp(t)"
t29 = "y" * b + (cos(y * a) * sin(t))/((f(y) * b * t) + (1 + y * 3) + sin(t))"
t30 = "y"+ tan(t) + (exp(y) - t + n) + (m - t * y')"
t31 = "y' - a * (t/y) + y^n * (t^(m + 1) + b)^!"
t32 = "y' - b * ((y + a)/y) * t^(c) + exp(c * y * t) * b * y^n * t^(n + 1)"
t33 = "y' - ((t^(m) * a) + b)/(csc(t) + (c * y^n))"
t34 = "y' - t * sin(t)/(a + t^n)"
t35 = "y' + (a * y^n) * (t^m + a) + (t^n * y^(1 + m) + b * t^n)/(c * y^m + t^(1 + n) * y^m)"
t36 = "a * y" + (t^n + a) * ((t^n * y^(1 + m)) + (b * t^n))/(c * y^m + t^(1 + n) * y^m)"
t37 = "a * y' * t^m + a * ((t^n * y^(1 + m)) + b * t^n)/(c * y^m + t^(1 + n) * y^m)"
t38 = "a * y' * t^m + a * ((y^(1 + m)) * t^n + (t^n * b))/(c * y^m + t^(1 + n) * y^m)"
t39 = "y' - (sin(m) * a)/(y^n * (b + t^(cos^(2)(y) + m)))"
t40 = "y' - sin(m) * a/(y^n * (b + t^(cos^(2)(y) + m)))"
t41 = "y' - (a * ln(t) * (y/t))/(m + (y + n) - (t * b))"
t42 = "y' + (a * c^(y * c) + g(t))/(t * (exp(y * c) * b) - f(y))"
t43 = "y' - h * sin(y * a) * cos(y) + (t^n + b) + y^(1 + m) * exp(t * p)/(cos(y * a) * t^(1 + n) * j + exp(p * t) * (y^m * c) + f(y))"
t44 = "y' + (a * y^(n + 1) * t^n + b * y * exp(y * t * a))/(y^m * sin(t * a) + f(y) + exp(y * p) * t^(1 + n) * b)"
t45 = "y' + (sin(y * a) * t * a + cos(y) * a * t)/(t^(1 + l) * y^n * w)"
t46 = "y' - ((y * a) + (b * t))/(y^2 + (t^2 * b))"
t47 = "y' - (y * a + t * b)/(y^2 + t^2 * b)"
t48 = "y' - (c/t) * y + t^(-b) * a/(a + c * sin(cos(y + a)) * t^(b - 1))"
t49 = "y' + (c/t) * y + a * t^(-b)/(a + c * sin(cos(y + a)) * t^(b - 1))"
t50 = "(c/(b - a)) * exp(b * t) + (c/((b - a)^2 * t)) * exp(b * t) + d/t * exp(a * t) + y'"

t51 = "y' + c/(b - a) * exp(t * b) + c/((b - a)^2 * t) * exp(b * t) + (d/t * exp(t * a))"
t52 = "y' + y'' * (a/t) + t^(-b) * a + (a * tan(t * b) * y) + sin(b * t) * c"
t53 = "m * cos(t * b)^(b/a) + y' - (y + c/b) + (b - a) + cos(b * t) * ln(sin^2(b * t))"
t54 = "y' + (y * cos(t * b)^(b/a)) - (a * c/(b * 2)) * arctan(b * t) * t"
t55 = "tan(y) + c' * t^(1 + a) + (d + t^(a) + exp(a * t)) * d + 1/(1 - sin(y)) * exp(t)"
t56 = "y'' + exp(t * a) + (d - t - (b/a)) - b/a^2"
t57 = "y' + (y + c/a) * (t^2 + b)^(-a/2)"
t58 = "y' - d * exp(t^(m + 1) * a/(1 + m)) - b/a"
t59 = "a' - (a + (b/a) * t^n + m * (t^p * 2 * (b/a)^2) + (8 * b)/a^3)"
t60 = "a' - (a + (b/a) * t^n + m * (t^p * 2 * b/a^2) - (8 * b)/a^3)"
t61 = "y' + (t * exp(t * a) * y - c/(b - a)) * t * exp(t * (b - a)) + (c/(b - a)^2) * exp(t + (b - a))"
t63 = "t' + (t * g + f) - y"
t62 = "y' * y^(1 + n) + y'' - (((y - b) + a)/((t + b) * (a - b))) + (y/n) * (b + (y - n)^(t))"
t64 = "g' + sin((cos(t + b) * a) + h) + (a/(b + t^(csc(g) + h))) * exp(t + (b - g))"
t65 = "m * ln(cos^(2)(a * t) * n * f) + f'' + sin(t) - exp(((f + a)^(-a/2)) * (m * (f - n * t)))"
t66 = "y' + y * (a + (cos(y) * sin(t))/b) - (t^(m) * y/(1 + m)) * (tan(t) + m) * a - (ln(y) * y + t - b)"
t67 = "y' + y * (t^(1 + a) + b) + ((y^(b) * t^(a)) + (t * b))/((t + b) - a)^(2) + 1/(1 - sin(y)) * exp(y)"
t68 = "t' + (d * (b * t - a) * c)"
t69 = "y' - (sin(y) - cos(t)) - t - (exp(t) + y + ln(y) - a)"
t70 = "f' + c + (sin(f) * g(y)) * 3 + y"
t71 = "y' + (cos(exp(y * a)) * sin(t)) * b + a"
t72 = "y' + t * (sin(y) - a) - (cos(y) + b)/(sin(cos(t)) + c)"
t73 = "h * sin(y * a) * cos(y) * t^n + b'' * y^(1 + m) * exp(b * p)/(cos(y * a) * t^(1 + y) * j + exp(p * t) * (y^m * c) + f(y))"
t74 = "y' + sin(y + (a * t) + b)^(m - y - t)"
t75 = " g''+ sin(t) - 2*g'=-tan(f /y)-y*t+g*exp(t)-g*y *sin^(2)(g)"
t76 = "f = tan(t)-csc^2(t)-h^2(f)*f'-11 *y+k(t)*f''*y-k(t)+k(t)^m-h(f)*k(t)"
t77 = "f' - tan^3(t)+ csc(t)+f''' +h(t)=k(f)-h^(1+t)(t)*k^2(f)-k(f)"
t78 = "k^(c)(y)+t'+ b * t =8*y-tan(t)-exp(t)*t'' -12*exp((t+1)^b)"
t79 = "b*t+t''-1+ ln(y)+t*exp(b^(1+y))+f(y)-g(t)+f(y)*g(t)=0"
t80 = "a^(3)+g-sec(g)*f' + 2*f'' +f'''*sin(a)+h(g)"
t81 = "y'''*sin(t)+cos(g)*f =f*(y*tan(t)) -2*(t)+h(y) -k^123(f)-k(f)*23 + k^(a)(f)"
t82 = "t*3+arccos(f)*y'' -y''' +f*y' +m(y)-3*n(t)+n(t)*m^(x+1)(sin(y))"
t83 = "k + g'' + g' + k^3 + g = g'''-1"
t84 = "sin(t) + a*y'' + k*(2*(j*(t))) - a*(b(t))+3*e(y) - cos(8*y) - (3*t)"
t85 = " y'''+ t^2 -3*m + sin(y-t)+m*t"
t86 = "y''' + g(x)^z - y''*f^2(y)+ y''^23 - x = 0"
t87 = "y'+ g(x)- y'''-f(y)*y''^2 + 1 -e*tan^23(x)+ (((x -1)*3) + t)"
t88 = "y'' + g(x) - y' - f(y)* y'''^2 + y*(x- 1)^x + sin^a(y)"
t89 = "g(t) + t* y'' + y' + cos(y + t)^23 * sin^2(t) * y - exp(tan^2(t)) + arctan^(y)(3 * t) * cos(t)"
t90 = "tan(y * t) * sin^(a(x) * y' + tan(t * y) * sin(x) * y - sin^(x + 1)(y) * csc^2(t) * y - y * f(y)"
t91 = "sin(x) * cos(t * y) * y' + sin(y) * cos(t * y) * y - f^(2)(y) * y + g(t) * y + g^3(t)"
t92 = "sec((2 * y) + y) * tan(t) * y' + sec(y + (3 * y)) * tan(t * y) * y' - sin(y + (3 * y)) * tan(t * y) * y + y'''"
t93 = "m + arctan^(y)(3 * t) * cos(x) * y''+ cos(y + t) * sin(x) * y'"
t94 = "m + y''^2+ cos(y + t) * sin(x) * y'"
t95 = "m + (y'')^2+ cos(y + t) * sin(x) * y'"
t96 = "y''+ 1/(1+y)"
t97 = "m + (3 * t) * cos(t) * y''+ cos(y) * sin(x)"
t98 = "m + (3 * t) * cos(t) * y''+ cos^2(y) * sin(x)"
t99 = "m + (3 * t) * cos(t) * y''+ cos^a(y) * sin(x)"
t100 = "y'' - f(y)* y'''^2 + y*(x- 1)^x + sin^a(y)"

t101 = "f'+cos((t+1)^b) + exp(f+cos(f))+ cos(g(t)+t)"
t102 = "ln^3(y)"
t103 = "sin^(a * b)(t * y)"
t104 = "ln^(a * b)(c * y)"
t105 = "arcsin(t)"
t106 = "ln(y * t)"
t107 = "(sin(t) + m - y - t)"
t108 = "(sin(i) - y - t + m)"
t109 = "(m - y + sin(t) - t)"
t110 = "(b * (z - a - (y - x)))"
t111 = "b * (z - a - (y - x))"
t112 = "(t - (exp(t) + y + ln(y) - a))"
t113 = "(a * sin^(cos(b + m))) + h"
t114 = "(sin(cos^(y + m)) * a) + h"
t115 = "(sin(cos(b + t) * a) + h)"
t116 = "((z - b) - a)"
t117= "((z - a) + b)"
t118 = "(z - y + m + (n - x))"

Appendix IV

Example of File Format for Data Exchange

Format

File Format

Standard, space delimited text file.

Format of Input Equation

Accepts the following as input:

- Standard math notation (+ - / * ^)
- Transcendental functions (sin, cos, tan, csc, sec, arcsin, arccos, arctan)
  - -arguments are parenthesized
- Transcendental functions (ln, exp)
  - -arguments are parenthesized
- Sequencing and spacing of terms is not important.
- Each order of the dependent variable is indicated by a '
- All variables are 1 character In length.

Initial Conditions

Optional.
One for each derivative.
Each derivative is on a separate line with the sequence in descending order starting with the first derivative.

Parameter Conditions

Optional.
Each parameter condition is on a separate line.

Input Requirements

Line 1	Input equation				
Line 2	Independent variable				
Line 3	1st deriv.	(space)	time	(space)	value
Line 4	2nd deriv.	(space)	time	(space)	value
Line 5	3rd deriv.	(space)	time	(space)	value
Line 6	4th deriv.	(space)	time	(space)	value
Line 7	5th deriv.	(space)	time	(space)	value
Line 8	const1	(space)	value		
Line 9	const2	(space)	value		
Line 10	const3	(space)	value		
Line 11	const4	(space)	value		
Line 12	const5	(space)	value		
	:				
	:				

Output

Echo of the input requirements.

Beginning of solution indicator: @
Result:
Highest order on LHS.
The operators +, -, and * have one leading blank and one trailing blank
May have more than 1 result if there are conditions that have not been specified, e.g. b=0 or b<>0.
Each result occurs 1 blank line following the integration constant of the last result.
Blank line

Auxialiary Variables:
In alphabetical order.
Blank line

Auxialiary Parameters:
In alphabetical order.
Blank line

Integration constant:
Blank line
@ - if there is another solution.

Ex. 1 Input

```
y' = a * t/y
t
@
```

Ex. 1 Output

```
Equation:   y' = a * t/y
Indep Var:  t
@
Solution:   y = + (k + a * (t^2))^(1/2), y = -(k + a * (t^2))^(1/2)
Integ. Constk = yo^2 - a * to^2
Conditions: y <> 0
```

Ex. 2 Input

```
y' = b * t^a
t
@
y' = a * y * t^b
t
@
x' = a * s^m * x^n
s
@
y' = (a * t^m)/((x^n) * (b + t^(m + 1))^L)
t
@
y' = (a + y^2)/(b + t^2)
t
@
x' = - (a * exp(c * x) + g(t))/(b * t * exp(c * x) + f(x))
t
@
y' = -(a * t^n * y + g(t))/(b * t^(n + 1) + f(y))
t
@
x' = -(b * x^(n + 1) * t^n * exp(c * x * t))/(b * x^n * t^(n + 1) * exp(c * x * t))
t
@
y' = (a + b * y^(m + 1))/(y^(m)) * t^n
t



m >= 1
@
x' = a * (b - c * (x/t)^2)^(1/2) + x/t
t
@
y' = (a * s + b * y)/(s + c * y)
s
@
x' = a * tan(b * t) * x + c * sin(b * t)
t
@
x' = a/t * x + c * t^b
t
@
```

Ex. 2 Output

```
Equation:   y' = b * t^a
Indep Var:  t
@
Solution:   y = (b/(a + 1)) * t^(a+1) + k
Integ. Constk = yo - (b/(a+1)) * to^(a + 1)
Conditions: a <> -1
@
Solution:   y = b * t + k
Integ. Constk = yo - b * to
Conditions: a = 0
@
Solution:   y = b * ln|t| + k
Integ. Constk = yo - b * ln|to|
Conditions: a = -1

@
Equation:   y' = a * y * t^b
Indep Var:  t
@
Solution:   y = k * exp(a/(b + 1) * t^(b + 1))
Integ. Constk = yo/exp(a/(b + 1) * to^(b + 1))
Conditions: b <> 1

@
Equation:   x' = a * s^m * x^n
Indep Var:  s
@
Solution:   x = (a * (1 - n)/(m + 1) * s^(m + 1) + k)^(1/(1 - n))
Integ. Constk = xo^(1 - n) - a * (1 - n)/(m + 1) * so^(m + 1)
Conditions: n <> -1
@
Solution:   x = k * exp(a/(m + 1) * x^(m + 1))
Integ. Constk = log|xo| - a/(m + 1) * xo^(m + 1)
Conditions: n = 1

@
Equation:   y' = (a * t^m)/((x^n) * (b + t^(m + 1))^L)
Indep Var:  t
@
Solution:   y = (a * (n + 1)/((m + 1) * (1 - L)) * (b + t^(m + 1))^(1 - L) + k)^(1/(n + 1))
Integ. Constk = yo^(m + 1) - a * (n + 1)/((m + 1) * (1 - L)) * (b + to^(m + 1))^(1 - L)
Conditions: n <> -1
@
Solution:   y = k * exp(a/((m + 1) * (1 - L))) * (b + t^(m + 1))^(1 - L)
Integ. Constk = log|yo| - a/((m + 1) * (1 - L)) * (b + to^(m + 1))^(1 - L)
Conditions: n = -1

@
Equation:   y' = (a + y^2)/(b + t^2)
Indep Var:  t
@
Solution:   y = a^(1/2) * tan(1/(2 * (a * (-b))^(1/2)) * ln((t - (-b)^(1/2))/(t + (-b)^(1/2))) + k)
Integ. Constk = 1/(a^(1/2)) * arctan(yo/(a^(1/2)) - 1/(2 * (-b)^(1/2)) * ln((to - (-b)^(1/2))/(to +
(-b)^(1/2)))
Conditions: a > 0
            b < 0

@
Equation:   x' = - (a * exp(c * x) + g(t))/(b * t * exp(c * x) + f(x))
Indep Var:  t
@
Solution:   a * t * exp(c * x) - G(t) + F(x) = k
Integ. Constk = a * to * exp(c * xo) + G(to) + F(xo)
Conditions: b = a * c

@
Equation:   y' = -(a * t^n * y + g(t))/(b * t^(n + 1) + f(y))
Indep Var:  t
@
Solution:   y = b * y * t^(n + 1) + G(t) + F(y) = k
Integ. Constk = b * yo * to^(n + 1) + G(to) + F(yo)
```

Ex. 2 Output

```
Conditions: a = b * (n + 1)
            n <> -1

@
Equation:   x' = -(b * x^(n + 1) * t^n * exp(c * x * t))/(b * x^n * t^(n + 1) * exp(c * x * t))
Indep Var:  t
@
Solution:   b * t^(n + 1) * ((x^n)/(c * t) * exp(c * t * x) - n/(c * t) * Ix^(n - 1) * exp(c * t *
x) dx) = k
Integ. Constk = b * to^(n + 1) * (xo^n/(c * to) * exp(c * to * xo) - n/(co * to) * F(xo,to))
Conditions: F(xo,to) = Ix^(n - 1) * exp(c * t * x) dx - g
            g: arbitrary
            n: integer
            n > 0

@
Equation:   y' = (a + b * y^(m + 1))/(y^(m)) * t^n
Indep Var:  t
@
Solution:   y = (1/b * ((k * t)^(b * (m + 1)) - a))^(1/(m + 1))
Integ. Constk = 1/to * (a + b * yo^(m + 1))^(1/(b* (m + 1)))
Conditions: m >= 1
            n = -1
@
Solution:   1/(b * (m + 1)) * ln(a + b * x^(m + 1)) - 1/(n + 1) * t^(n + 1) = k
Integ. Constk = 1/(b * (m + 1)) * ln(a + b * yo^(m + 1)) - 1/(n + 1) * to^(n + 1)
Conditions: m >= 1
            n <> -1

@
Equation:   x' = a * (b - c * (x/t)^2)^(1/2) + x/t
Indep Var:  t
@
Solution:   x = ((b/c)^(1/2) * t)/sin(ln|x^(a * b^(1/2))| + k)
Integ. Constk = arcsin((b/c)^(1/2)) * to/xo - ln|xo^(a * b^(1/2))|
Conditions: c > 0
            x > 0

@
Equation:   y' = (a * s + b * y)/(s + c * y)
Indep Var:  s
@
Solution:   1/s * ((y/s) - P1)^A1 * ((y/s) - P2)^A2 = k
Integ. Constk = 1/so * ((yo/so) - P1)^A1 * ((yo/so) - P2)^A2
Conditions: A1 = (1 + c * P1)/(P1 - P2)
            A2 = (1 + c * P2)/(P2 - P1)
            P1 = (1 - b - ((b - 1)^2 - 4 * a * c)^(1/2))/(-2 * c)
            P2 = (1 - b - ((b - 1)^2 - 4 * a * c)^(1/2))/(-2 * c)
            a > 0
            b > 0
            b <> 1
            c > 0
@
Solution:   s^(2 * (a * c)^(1/2)) * (a - c * (y/s)^2)^((a * c)^(1/2)) * ((c^(1/2) * y/s +
a^(1/2))/(c^(1/2) * y/s - a^(1/2))) = k
Integ. Constk = so^(2 * (a * c)^(1/2)) * (a - c * (yo/so))^((a * c)^(1/2)) * ((c^(1/2) * yo/so +
a^(1/2))/(c^(1/2) * yo/so - a^(1/2)))
Conditions: a > 0
            b = 1
            c > 0

@
Equation:   x' = a * tan(b * t) * x + c * sin(b * t)
Indep Var:  t
@
Solution:   x = k * cos^(-b/a)(b * t) - (a * c)/(b * (b + a)) * cos(b * t)
Integ. Constk = xo * cos^(b/a)(b * to) + (a * c)/(b * (b + a)) * cos^(b/a)(b * to) + 1
Conditions: a <> b
            a <> -b
@
Solution:   x = k * cos^(-1)(b * t) - c/(2 * b) * sin^2(b * t)
```

Ex. 2 Output

```
Integ. Constk = xo * cos(b * to) + c/(2 * b) * sin^2(b * to)
Conditions: a = b
@
Solution:   x = k * cos(b * t) + c/b * cos(b * t) * ln(cos(b * t))
Integ. Constk = xo * cos^(-1)(b * to) - c/b * ln(cos(b * to))
Conditions: a = -b

@
Equation:   x' = a/t * x + c * t^b
Indep Var:  t
@
Solution:   x = c/(b - a + 1) * t^(b + 1) + k * t^a
Integ. Constk = 1/to^a * (xo - c/(b - a + 1) * to^(b+1))
Conditions: a <> b
            b - a <> -1
@
Solution:   x = t^a * (ln|t^c| + k)
Integ. Constk = to^(-a) * x - ln|to^c|
Conditions: a <> b
@           b - a = -1
Solution:   x = c * t^(a + 1) + k * t^a
Integ. Constk = x * t^(-a) - c * t
Conditions: a = b
```

Appendix V

Source Code

```
=============================================================================
Attribute VB_Name = "Interface"
------------------------------
Dim command As Integer
Dim toolbar_array(15) As Boolean


Sub Auto_Open()
Attribute Auto_Open.VB_ProcData.VB_Invoke_Func = " \n14"
'Begins upon program start-up to bring up the logo.

  Application.ScreenUpdating = False
  Sheets("Logo").Visible = True
  Sheets("Logo").Select
  [A1].Show

  If (Toolbars(1).Visible) Then toolbar_array(1) = True
  If (Toolbars(2).Visible) Then toolbar_array(2) = True
  If (Toolbars(3).Visible) Then toolbar_array(3) = True
  If (Toolbars(4).Visible) Then toolbar_array(4) = True
  If (Toolbars(5).Visible) Then toolbar_array(5) = True
  If (Toolbars(6).Visible) Then toolbar_array(6) = True
  If (Toolbars(7).Visible) Then toolbar_array(7) = True
  If (Toolbars(8).Visible) Then toolbar_array(8) = True
  If (Toolbars(9).Visible) Then toolbar_array(9) = True
  If (Toolbars(10).Visible) Then toolbar_array(10) = True
  If (Toolbars(11).Visible) Then toolbar_array(11) = True
  If (Toolbars(12).Visible) Then toolbar_array(12) = True
  If (Toolbars(13).Visible) Then toolbar_array(13) = True

  Toolbars(1).Visible = False
  Toolbars(2).Visible = False
  Toolbars(3).Visible = False
  Toolbars(4).Visible = False
  Toolbars(5).Visible = False
  Toolbars(6).Visible = False
  Toolbars(7).Visible = False
  Toolbars(8).Visible = False
  Toolbars(9).Visible = False
  Toolbars(10).Visible = False
  Toolbars(11).Visible = False
  Toolbars(12).Visible = False
  Toolbars(13).Visible = False


  With Application
    .DisplayStatusBar = False
    .DisplayNoteIndicator = False
  End With

  With ActiveWindow
    .DisplayGridlines = False
    .DisplayHeadings = False
    .DisplayOutline = False
    .DisplayHorizontalScrollBar = False
    .DisplayVerticalScrollBar = False
    .DisplayWorkbookTabs = False
  End With

  Application.OnKey "{return}", "showMain"

End Sub


Sub showMain()

  If ActiveWindow.DisplayHorizontalScrollBar Then
    ActiveWindow.DisplayHorizontalScrollBar = False
  End If
  If Sheets("Logo").Visible Then
    Sheets("Logo").Visible = False
```

```
    Application.OnKey "{return}"
  End If
  Sheets("Main").Select
  Range("A1").Select
  Range("M20").Select

End Sub


Sub showEntryForm()
Attribute showEntryForm.VB_ProcData.VB_Invoke_Func = " \n14"

  Sheets("EntryForm").Select
  If Not ActiveWindow.DisplayHorizontalScrollBar Then
    ActiveWindow.DisplayHorizontalScrollBar = True
  End If
  Range("A50").Value = 0
  equation = ""
  disableDeriv
  Range("A16").Select
  Selection.ClearContents

End Sub


Sub getOrder()
Attribute getOrder.VB_ProcData.VB_Invoke_Func = " \n14"
'This procedure is called once a derivative order has been selected in
'the ODE entry form.  It enables both the dy/dt and y derivative buttons
'to allow the user a choice of equation forms for entry and display if the
'point-and-click interface is used.

  ActiveSheet.DrawingObjects(Array("dyDrv", "yDrv")).Select
  Selection.Interior.ColorIndex = 40
  Selection.Font.ColorIndex = 5
  ActiveSheet.DrawingObjects(Array("dyDrv", "yDrv")).Enabled = True
  Range("A16").Select

End Sub


Sub disableDeriv()
Attribute disableDeriv.VB_ProcData.VB_Invoke_Func = " \n14"
'Disables both the dy/dt and y derivative buttons on the ODE entry form.
'This procedure is called after a derivative has been selected and cannot
'be selected again until another order has been chosen.

  ActiveSheet.DrawingObjects(Array("dyDrv", "yDrv")).Select
  Selection.Interior.ColorIndex = 19
  Selection.Font.ColorIndex = 16
  ActiveSheet.DrawingObjects(Array("dyDrv", "yDrv")).Enabled = False

End Sub


Sub dyDeriv()
Attribute dyDeriv.VB_ProcData.VB_Invoke_Func = " \n14"
'Called when the dy/dt button in the ODE entry form is pressed, this
'procedure disables the derivative buttons (since a derivative has now
'been chosen), and assigns the selected order to the derivative.  The
'highest order for the system is currently set at 5 but this can be
'increased by adding higher order cases here and increasing the knowledge
'base accordingly.
Dim cur_deriv As String, equation As String

  disableDeriv
  Select Case Range("D12").Value    'Set the derivative to the chosen order.
    Case 1
      cur_deriv = "Dy/Dt"
    Case 2
      cur_deriv = "D2y/Dt2"
```

```
    Case 3
      cur_deriv = "D3y/Dt3"
    Case 4
      cur_deriv = "D4y/Dt4"
    Case 5
      cur_deriv = "D5y/Dt5"
  End Select
  Range("A50").Value = 0
  Range("A16").Select
  equation = ActiveCell.Formula + cur_deriv
  displayEqn equation, "A16"

End Sub


Sub yDeriv()
Attribute yDeriv.VB_ProcData.VB_Invoke_Func = " \n14"
'Called when the y derivative button in the ODE entry form is pressed, this
'procedure disables the derivative buttons (since a derivative has now
'been chosen), and assigns the selected order to the derivative.  The
'highest order for the system is currently set at 5 but this can be
'increased by adding higher order cases here and increasing the knowledge
'base accordingly.
Dim cur_deriv As String, equation As String

  disableDeriv
  Select Case Range("D12").Value    'Set the derivative to the chosen order.
    Case 1
      cur_deriv = "y'"
    Case 2
      cur_deriv = "y''"
    Case 3
      cur_deriv = "y'''"
    Case 4
      cur_deriv = "y''''"
    Case 5
      cur_deriv = "y'''''"
  End Select
  Range("A50").Value = 0
  Range("A16").Select
  equation = ActiveCell.Formula + cur_deriv
  displayEqn equation, "A16"

End Sub


Sub displayEqn(equation, addr)
Attribute displayEqn.VB_ProcData.VB_Invoke_Func = " \n14"
'This procedure is called whenever new elements are added to an ODE.
'It properly formats superscripts associated derivatives.
Dim pos As Integer, cur_len As Integer

  If InStr(equation, "D") > 0 Then
    Application.ScreenUpdating = False
  End If
  Range(addr).Value = equation
  pos = InStr(equation, "D")
  If pos > 0 Then
    equation = Mid(equation, pos)
    cur_len = pos                      'Pointer to entire equation.
    Do
      'Check for a derivative of order greater than 1.
      If (Left(equation, 1) = "D") And (Left(equation, 2) <> "Dy") Then
        Range(addr).Characters(start:=cur_len + 1, length:=1).Font.Superscript = True
        Range(addr).Characters(start:=cur_len + 6, length:=1).Font.Superscript = True
        equation = Mid(equation, 8)    'Skip to end of higher order deriv.
        cur_len = cur_len + 7
      Else
        equation = Mid(equation, 6)    'Skip to end of 1st order deriv.
        cur_len = cur_len + 5
      End If
```

```
      pos = InStr(equation, "D")       'Look for next derivative.
      If pos > 0 Then
        equation = Mid(equation, pos)
        cur_len = cur_len + pos - 1
      Else
        Exit Do
      End If
    Loop
    Application.ScreenUpdating = True
  End If

End Sub


Sub showFn1()
Attribute showFn1.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + "g(t)"
  displayEqn eqn, "A16"

End Sub


Sub showFn2()
Attribute showFn2.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + "f(y)"
  displayEqn eqn, "A16"

End Sub


Sub showT()
Attribute showT.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + "t"
  displayEqn eqn, "A16"

End Sub


Sub showY()
Attribute showY.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + "y"
  displayEqn eqn, "A16"

End Sub


Sub showLParen()
Attribute showLParen.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + "("
  displayEqn eqn, "A16"

End Sub


Sub showRParen()
Attribute showRParen.VB_ProcData.VB_Invoke_Func = " \n14"
```

```
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + ")"
  displayEqn eqn, "A16"

End Sub


Sub showPlus()
Attribute showPlus.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + " + "
  displayEqn eqn, "A16"

End Sub


Sub showMinus()
Attribute showMinus.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + " - "
  displayEqn eqn, "A16"

End Sub


Sub showMult()
Attribute showMult.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + " * "
  displayEqn eqn, "A16"

End Sub


Sub showDiv()
Attribute showDiv.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + "/"
  displayEqn eqn, "A16"

End Sub


Sub showExpon()
Attribute showExpon.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + "^("
  displayEqn eqn, "A16"

End Sub


Sub showInteg()
Attribute showInteg.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + "I("
  displayEqn eqn, "A16"
```

```
End Sub


Sub showEqual()
Attribute showEqual.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + " = "
  displayEqn eqn, "A16"

End Sub


Sub showE()
Attribute showE.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + "e"
  displayEqn eqn, "A16"

End Sub


Sub showPi()
Attribute showPi.VB_ProcData.VB_Invoke_Func = " \n14"
Dim eqn As String

  Range("A16").Select
  eqn = ActiveCell.Formula + "pi"
  displayEqn eqn, "A16"

End Sub


Sub showTranscendental()
Attribute showTranscendental.VB_ProcData.VB_Invoke_Func = " \n14"
'Called when the Transcendentals button is pressed in EntryForm.
'Activates the transcendentals dialog box for the user to select
'a transcendental.
Dim eqn As String, trans As String

  If DialogSheets("transDbx").Show Then
    With DialogSheets("transDbx").ListBoxes("transList")
      trans = .List(.ListIndex)
    End With
    Range("A16").Select
    eqn = ActiveCell.Formula + trans + "("
    displayEqn eqn, "A16"
  End If

End Sub


Sub showConstant()
Attribute showConstant.VB_ProcData.VB_Invoke_Func = " \n14"
'Called when the Constants button is pressed in EntryForm.
'Activates the constants dialog box for the user to select
'either an alpha or numeric constant.
Dim eqn As String, constant As String

  If DialogSheets("constDbx").Show Then
    With DialogSheets("constDbx")
      If .OptionButtons("optAlpha").Value = xlOn Then
        With .ListBoxes("alphaList")
          constant = .List(.ListIndex)
        End With
      Else
        constant = .TextBoxes("Number").Text
```

```
      End If
    End With
    Range("A16").Select
    eqn = ActiveCell.Formula + constant
    displayEqn eqn, "A16"
  End If

End Sub


Sub pickAlpha()
Attribute pickAlpha.VB_ProcData.VB_Invoke_Func = " \n14"
'Activated by selecting from the alpha listbox in the Constants dialog
'box.  It sets the corresponding Alpha option button to true.
  DialogSheets("constDbx").OptionButtons("optAlpha").Value = xlOn
End Sub


Sub pickNumeric()
Attribute pickNumeric.VB_ProcData.VB_Invoke_Func = " \n14"
'Activated by scrolling the numeric spinner in the Constants dialog
'box.  It sets the corresponding Numeric option button to true.
  DialogSheets("constDbx").OptionButtons("optNumeric").Value = xlOn
End Sub


Sub noEntry()
Attribute noEntry.VB_ProcData.VB_Invoke_Func = " \n14"
'Called when the Cancel button is pressed in the ODE Entry Form.
'It activates the No Entry dialog box to exit the program,
'return to the main menu, or return for a new entry in Entry Form.

  If DialogSheets("noEntryDbx").Show Then
    With DialogSheets("noEntryDbx")
      If .OptionButtons("optExit").Value = xlOn Then
        closeODEASY
      Else
        If .OptionButtons("optMain").Value = xlOn Then
          Sheets("Main").Select
          Range("M20").Select
          With ActiveWindow
            .DisplayHorizontalScrollBar = False
            .DisplayVerticalScrollBar = False
          End With
          ActiveSheet.OptionButtons("search_opt").Value = xlOn
        Else
          Range("A16").Value = ""
        End If
      End If
    End With
  End If

End Sub


Sub addDone()
Attribute addDone.VB_ProcData.VB_Invoke_Func = " \n14"
'Connected to the OK button on the Add equation form.  Depending upon
'the option button selected on the form, the equation entry is either
'added to the ODEASY knowledge base or it is erased, or the program
'returns to the Main menu, or the application closes.

'Check for empty equation.  Also save if new eqn added.
  If ActiveSheet.DrawingObjects("Add").Value = xlOff Then
    Sheets("EntryForm").Activate
    Range("A16").ClearContents
  Else
  End If
End Sub
```

```
Sub get_entry(entryType)
'Called by procedures activated by the option buttons in the Main
'Menu and by the add, and edit procedures to get an ODE from
'the user.  Brings up the ODE Entry Form.  The entryType parameter
'indicates its add (0), edit (2), search (3), delete (1), or to
'show a database (4).

  If entryType = 4 Then
    show_db
  Else
    command = entryType
    showEntryForm
  End If

End Sub


Sub Return_Settings()
Attribute Return_Settings.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns the Excel environment back to the way it was before the toolbars,
'scrollbars, etc. had to be removed to show the logo and the ODEASY working
'environment.

  Application.ScreenUpdating = False
  With ActiveWindow
    .DisplayHeadings = True
    .DisplayOutline = True
    .DisplayHorizontalScrollBar = True
    .DisplayVerticalScrollBar = True
    .DisplayWorkbookTabs = True
  End With
  Application.OnKey "{return}"
  If (toolbar_array(1) = True) Then Toolbars(1).Visible = True
  If (toolbar_array(2) = True) Then Toolbars(2).Visible = True
  If (toolbar_array(3) = True) Then Toolbars(3).Visible = True
  If (toolbar_array(4) = True) Then Toolbars(4).Visible = True
  If (toolbar_array(5) = True) Then Toolbars(5).Visible = True
  If (toolbar_array(6) = True) Then Toolbars(6).Visible = True
  If (toolbar_array(7) = True) Then Toolbars(7).Visible = True
  If (toolbar_array(8) = True) Then Toolbars(8).Visible = True
  If (toolbar_array(9) = True) Then Toolbars(9).Visible = True
  If (toolbar_array(10) = True) Then Toolbars(10).Visible = True
  If (toolbar_array(11) = True) Then Toolbars(11).Visible = True
  If (toolbar_array(12) = True) Then Toolbars(12).Visible = True
  If (toolbar_array(13) = True) Then Toolbars(13).Visible = True

  With Application
    .DisplayStatusBar = True
    .DisplayNoteIndicator = True
    .ShowToolTips = True
    .ColorButtons = True
  End With
  Sheets("Logo").Visible = False
  Application.ScreenUpdating = True

End Sub


Sub closeODEASY()
Attribute closeODEASY.VB_ProcData.VB_Invoke_Func = " \n14"
'Close application without querying for save.  This does not save
'the application but only masks the query.  The Save option is available
'when a new equation & solution have been added to the knowledge base.

  Return_Settings
  ActiveWorkbook.Saved = True
  Application.Quit

End Sub
```

```
Sub EntryDone()
Attribute EntryDone.VB_ProcData.VB_Invoke_Func = " \n14"
'Activated when the "Done" button is clicked in the ODE Entry Form.
'Calls procedures to ensure all elements are on the LHS of the "="
'sign, changes equations entered in superscript into standard form,
'is sorted, changes display variables into internal variables, and
'performs an add, edit, or search function (the type of function
'depends upon the module command variable that has been set from
'the main menu or in one of the 3 functions already executed).
Dim equation As String, entryEqn As String

  entryEqn = Range("A16").Value
  'First, ensure that it is a differential equation.
  If (InStr(entryEqn, "D") = 0) And (InStr(entryEqn, "'") = 0) Then
    i = MsgBox("Entry is not a differential equation.", _
        vbOKOnly + vbExclamation, "Please Re-enter")
    Range("A16") = ""
  Else
    Application.ScreenUpdating = False
    equation = entryEqn
    preProcess equation
    sortEqn equation
    swapConst equation  'Post sort phase replaces display constants.

    Select Case command
      Case 0: add entryEqn, equation
      Case 1: 'delete
      Case 2: 'edit
      Case 3: searchEqn entryEqn, equation
    End Select
  End If

End Sub


Sub opt_chosen()
'Activated by clicking the OK button on the Main Menu.

  With ActiveSheet
    If .OptionButtons("search_opt").Value = xlOn Then
      get_entry (3)
    Else
      If .OptionButtons("add_opt").Value = xlOn Then
        get_entry (0)
      Else
        If .OptionButtons("edit_opt").Value = xlOn Then
       '   get_entry (2)  - not implemented
        Else
          If .OptionButtons("delete_opt").Value = xlOn Then
          '    get_entry (1) - not implemented
          Else
            If .OptionButtons("showdb_opt").Value = xlOn Then
              get_entry (4)
            Else
              closeODEASY
            End If
          End If
        End If
      End If
    End If
  End With

End Sub


Sub show_db()
'This routine is activated by an option button selected from the
'Main menu.  It will bring up a dialog box for the user to choose
'the database to show.

  If DialogSheets("Db_Dbx").Show Then
```

```
  'The result of the listbox in this dialogsheet is entered into
  'B20 of defaults sheet.

    Select Case Sheets("defaults").Range("B20").Value
      Case 1: Sheets("1st Ordr Lnr").Select
      Case 2: Sheets("1st Ordr NonLnr").Select
      Case 3: Sheets("2nd Ordr Lnr").Select
      Case 4: Sheets("2nd Ordr NonLnr").Select
      Case 5: Sheets("3rd Ordr Lnr").Select
      Case 6: Sheets("3rd Ordr NonLnr").Select
      Case 7: Sheets("4th Ordr Lnr").Select
      Case 8: Sheets("4th Ordr NonLnr").Select
    End Select

   'Turn on scrollbars.
    With ActiveWindow
      .DisplayHorizontalScrollBar = True
      .DisplayVerticalScrollBar = True
    End With
    Range("A5").Select

   'Add menu to top menu bar to allow return to ODEASY.
    MenuBars(xlWorksheet).Menus.add "&ODEASY"
    MenuBars(xlWorksheet).Menus("ODEASY").MenuItems.add _
            Caption:="&Main Menu", OnAction:="del_menu"
  End If

End Sub


Sub del_menu()
'Removes menu added to top menu bar used in the database
'sheets to return to ODEASY.

'Turn off scrollbars.
  With ActiveWindow
    .DisplayHorizontalScrollBar = False
    .DisplayVerticalScrollBar = False
  End With
  MenuBars(xlWorksheet).Menus("ODEASY").Delete
  showMain

End Sub


================================================================================
Attribute VB_Name = "Parser"
------------------------------
Dim eqnDisplay As Integer          'Indicates whether equation is to be
                                   'displayed as y' (0) or Dy/Dt (1).

Sub preProcess(equation)
Attribute preProcess.VB_ProcData.VB_Invoke_Func = " \n14"
'Variables in the equation are swapped for standard ones for internal
'processing.  The display will continue to show the original variables.
'Determines whether an equation is in y' form or Dy form.  If it is
'in the latter form or a mix of the two forms, it will be converted
'to the y' form for internal processing.  All elements will be moved
'to the LHS of the "=" sign to prepare the equation for sorting.
Dim index As Integer, term As String
Dim leftSide As String, rightSide As String
Dim tempStr As String

  Sheets("defaults").Range("I2:I14").Clear
  Sheets("defaults").Range("F22:I30").Clear
  equation = Trim(equation)
  swapIndVar equation           'Substitute independent and dependent
  swapDepVar equation           'variables, as well as function names
  swapFns equation              'with those predefined for internal
                                'processing.
  spaceSigns equation
```

```
  index = InStr(equation, "=")
  If index <> 0 Then

      leftSide = Trim(Mid(equation, 1, (index - 1)))
      rightSide = Trim(Mid(equation, (index + 1)))

      Do While (Len(rightSide) > 0) And (rightSide <> "0")
        term = get_term(rightSide)
        If (Left(term, 1) <> "-") Then
          If (Left(term, 1) = "+") Then           'Term begins with a "-"
            term = Mid(term, 2)                   'operator so assign "+"
          End If                                  'to LHS before adding term.
          If Left(term, 1) = "*" Then             'No adjustment required if
            leftSide = leftSide + "*" + Trim(term)     'its a multiplicative
          Else                                        'term.
            leftSide = leftSide + " - " + Trim(term)
          End If
        Else                                      'Term begins with a "+"
          term = Mid(term, 2)                     'so assign "-" to LHS.
          leftSide = leftSide + " + " + Trim(term)
        End If
      Loop
      equation = leftSide
  End If

  pos = InStr(equation, "^")
  If pos > 0 Then
    Do While pos > 0
      If Mid(equation, pos + 1, 1) <> "(" Then    'Bracket the single exponent.
        tempStr = Mid(equation, 1, pos) + "("
        Do While (Mid(equation, pos + 1, 1) <> ")") And _
                 (Mid(equation, pos + 1, 1) <> " ") And _
                  Mid(equation, pos + 1) <> Empty
        'Read entire exponent.  An exponent of two or
        'more characters is numerical.
          tempStr = tempStr + Mid(equation, pos + 1, 1)
          pos = pos + 1
        Loop
        equation = tempStr + ")" + Mid(equation, pos + 1)
      End If
      pos = pos + 2
      pos = InStr(pos, equation, "^")
    Loop
  End If
  equation = "(" + equation + ")"                 'Parenthesize for Sorter.

End Sub


Function preDefined(eqn, pos) As Boolean
Attribute preDefined.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by swapIndVar, swapDepVar, and swapConst to determine
'whether a display variable, indicated by pos in equation eqn,
'is part of a transcendental or pi name.  Returns True if it is,
'False otherwise.
Dim subterm1 As String, subterm2 As String
Dim subterm3 As String, reserved As Boolean

  reserved = False
  subterm1 = ""
  subterm2 = ""
  subterm3 = ""
  subterm1 = Mid(eqn, pos, 3)                 'Isolate all 3 character
  If pos > 1 Then                             'substrings surrounding
    subterm2 = Mid(eqn, (pos - 1), 3)         'the character sought in
    If (pos - 1) > 1 Then                     'order to ensure that it
      subterm3 = Mid(eqn, (pos - 2), 3)       'is not part of a trans-
    End If                                    'cendental.
  End If

  Select Case Mid(eqn, pos, 1)
```

```
  Case "a"
    If (InStr(subterm1, "arc") > 0) Or (InStr(subterm2, "tan") > 0) Then
      reserved = True
    End If
  Case "c"
    If (InStr(subterm1, "cos") > 0) Or (InStr(subterm1, "csc") > 0) Or _
       (InStr(subterm3, "arc") > 0) Or (InStr(subterm3, "csc") > 0) Or _
       (InStr(subterm3, "sec") > 0) Then
      reserved = True
    End If
  Case "e"
    If (InStr(subterm1, "exp") > 0) Or (InStr(subterm2, "sec") > 0) Then
      reserved = True
    End If
  Case "i"
    If (InStr(subterm2, "sin") > 0) Or (InStr(subterm2, "pi") > 0) Then
      reserved = True
    End If
  Case "l"
    If (InStr(subterm1, "ln") > 0) Then
      reserved = True
    End If
  Case "n"
    If (InStr(subterm2, "ln") > 0) Or (InStr(subterm3, "sin") > 0) Or _
       (InStr(subterm3, "tan") > 0) Then
      reserved = True
    End If
  Case "o"
    If (InStr(subterm2, "cos") > 0) Then
      reserved = True
    End If
  Case "p"
    If (InStr(subterm1, "pi") > 0) Or (InStr(subterm3, "exp") > 0) Then
      reserved = True
    End If
  Case "r"
    If (InStr(subterm2, "arc") > 0) Then
      reserved = True
    End If
  Case "s"
    If (InStr(subterm1, "sin") > 0) Or (InStr(subterm1, "sec") > 0) Or _
       (InStr(subterm2, "csc") > 0) Or (InStr(subterm3, "cos") > 0) Then
      reserved = True
    End If
  Case "t"
    If (InStr(subterm1, "tan") > 0) Then
      reserved = True
    End If
  Case "x"
    If (InStr(subterm2, "exp") > 0) Then
      reserved = True
    End If
  Case "I"      'Symbol for integral.
    reserved = True
  End Select
  preDefined = reserved

End Function


Sub swapIndVar(equation)
Attribute swapIndVar.VB_ProcData.VB_Invoke_Func = " \n14"
'Where the entry is in y' form only, determines the independent variable
'from the user and, provided the independent variable occurs in the
'equation, substitutes it with the independent variable recognized for
'internal processing.  Where the user performs no entry, the independent
'variable used for internal processing is assumed.
Dim pos As Integer, dmyConst As String
Dim indepVar As String, depVar As String
Dim msg As String, internalVar As String
```

```
  internalVar = Sheets("defaults").Range("G3").Value
  If InStr(equation, "D") = 0 Then  'Verify with user the indep. var.
    Do
      DialogSheets("indpVarDbx").EditBoxes("iVar").Text = internalVar
      DialogSheets("indpVarDbx").Show
      indepVar = DialogSheets("indpVarDbx").EditBoxes("iVar").Text
      If Len(indepVar) <> 1 Then    'Length of indep. var. exceeds 1.
        msg = "Please limit the independent variable to 1 character in length."
        i = MsgBox(msg, vbOKOnly + vbExclamation, _
                   "Independent Variable Length Error")
      Else
        pos = InStr(equation, "'")
        depVar = Mid(equation, (pos - 1), 1)
        If indepVar = depVar Then   'Indep. and dep. var. cannot be the same.
          msg = "'" + depVar + "' " + _
                "is the dependent variable.  Please use another character."
          i = MsgBox(msg, vbOKOnly + vbExclamation, _
                     "Independent Variable Naming Error")
        Else
          Exit Do
        End If
      End If
    Loop
  Else  'Take the indep. variable from the Dy/Dx form that has been entered.
    pos = InStr(equation, "/")
    indepVar = Mid(equation, (pos + 2), 1)
  End If

  Sheets("defaults").Select
  Range("I3").Value = indepVar
  If (indepVar <> internalVar) Then         'Internal independent variable
                                            'differs from display variable.

  'Must first ensure that the internal indep var is not used as a constant
  'or the dependent variable in the display equation.  Must also ensure
  'that any such constant or dependent variable found is not part of a
  'transcendental word (i.e. "tan").  If it is not, then replace it with
  'dummy constant.
    If InStr(equation, internalVar) > 0 Then
      dmyConst = Range("G16").Value
      index = 1
      Do
        pos = InStr(index, equation, internalVar)
        If pos > 0 Then
          If Mid(equation, pos, 3) <> "tan" Then
            Mid(equation, pos) = dmyConst
          End If
          index = pos + 1
        End If
      Loop Until pos = 0
    End If

  'Next, substitute processing variable for display variable if required.
    If (InStr(equation, indepVar) > 0) Then
      pos = InStr(equation, indepVar)
      Do  'Need to check for special cases here.
        If (Not preDefined(equation, pos)) Then
          If (pos = 1) Then
            Mid(equation, pos) = internalVar
          ElseIf (Mid(equation, (pos - 1), 1) <> "D") Then  'Only replace
            Mid(equation, pos) = internalVar                'if independent
          End If                                            'variable is not
        End If                                              'part of Dy/Dt form.
        pos = InStr((pos + 1), equation, indepVar)      'Seek next substitution.
      Loop Until pos = 0
    End If
  End If
  Sheets("EntryForm").Select

End Sub
```

```
Sub swapDepVar(equation)
Attribute swapDepVar.VB_ProcData.VB_Invoke_Func = " \n14"
'The dependent variable is changed to the one required for internal
'processing if necessary.  Those equations that are in Dy form will
'continue to be displayed in that form (eqnDisplay set to 1) while
'those in y' form or a mix of the two will be displayed in y' form
'(eqnDisplay set to 0).
Dim dmyConst As String, leftStr As String
Dim depVar As String, procVar As String
Dim pos As Integer, constPos As Integer

  If InStr(equation, "'") <> 0 Then
    eqnDisplay = 0              'Display equation in y' form.
    pos = InStr(equation, "'")
  Else
    eqnDisplay = 1              'Display equation in Dy form.
    pos = InStr(equation, "/")
  End If
  depVar = Mid(equation, (pos - 1), 1)
  Sheets("defaults").Select
  Range("I2").Value = depVar
  procVar = Range("G2").Value

 'Must first ensure that the internal dependent var is not used as a constant
 'in the display equation.  If it is, then replace it with dummy constant.
  If (depVar <> procVar) And (InStr(equation, procVar) > 0) Then
    constPos = InStr(equation, procVar)
    dmyConst = Range("G15").Value
    Do
      Mid(equation, constPos) = dmyConst
      constPos = InStr((constPos + 1), equation, procVar)
    Loop Until constPos = 0
  End If

  pos = InStr(equation, depVar)  'Next, perform substitution where required.
  Do
    If Mid(equation, (pos + 1), 2) = "/D" Then      'A derivative in Dy form
      If Mid(equation, (pos - 1), 1) <> "D" Then    'of order greater than 1.
        order = Val(Mid(equation, pos - 1, 1))
        leftStr = Left(equation, pos - 3) + procVar
        Do
          leftStr = leftStr + "'"
          order = order - 1
        Loop Until order = 0
        leftStr = leftStr + Mid(equation, pos + 5)
      Else                                  'A derivative in Dy form of order 1.
        leftStr = Left(equation, pos - 2) + procVar + "'"
        leftStr = leftStr + Mid(equation, pos + 4)
      End If
      equation = leftStr
    ElseIf (Not preDefined(equation, pos)) Then     'A derivative in y' form
      Mid(equation, pos) = procVar                  'which only requires
    End If                                          'direct substitution.
    pos = InStr((pos + 1), equation, depVar)
  Loop Until pos = 0

  Sheets("EntryForm").Select

End Sub


Sub swapFns(equation)
Attribute swapFns.VB_ProcData.VB_Invoke_Func = " \n14"
'Changes the names of functions for the dependent and independent variable
'to those recognized for internal processing.  It is assumed that the names
'of functions are one character long and that there is at most one function
'name associated with the dependent variable and one function name associated
'with the independent variable.
Dim depVar As String, indVar As String
Dim lbrac As Integer, rbrac As Integer
```

```
Dim curFn As String, varFound As Boolean
Dim isIVar As Boolean, isDVar As Boolean
Dim pos As Integer, fnPos As Integer
Dim dMatch As Boolean, iMatch As Boolean

  Sheets("defaults").Select
  pos = InStr(equation, "(")
  If pos > 0 Then
    depVar = Range("G2").Value
    indVar = Range("G3").Value
    Range("I4").Value = Range("G4").Value        'Initialize function names
    Range("I5").Value = Range("G5").Value        'to those used for internal
    isDVar = False                               'processing.
    isIVar = False
    dMatch = False
    iMatch = False

    If (InStr(equation, Range("G4").Value) > 0) Or _
       (InStr(equation, Range("G5").Value) > 0) Then    'Replace any consts
      substConst equation                               'that are fn names.
    End If
    If InStr(equation, "f(y)") > 0 Then
      dMatch = True
    End If
    If InStr(equation, "g(t)") > 0 Then
      iMatch = True
    End If
    If Not (dMatch And iMatch) Then
    Do
    'Search for names of the respective functions.
      curFn = get_fn(equation, pos, fnPos)
      If curFn <> " " Then

        If (curFn <> depVar) And (curFn <> indVar) And _
           (curFn <> "/") And (Not (is_num(curFn))) Then
        'The function name is not a number, the dependent or independent
        'variable, so determine whether it is a function of the dependent
        'or independent variable or neither.

          lbrac = 1
          rbrac = 0
          varFound = False
          Do                'Loop til var. is found or there is no variable.
            pos = pos + 1
            Select Case Mid(equation, pos, 1)
            Case depVar
            If Not dMatch Then
              Mid(equation, fnPos) = Range("G5").Value   'Replace display
              If Not isDVar Then                         'name with internal.
                isDVar = True
                varFound = True
                Range("I5").Value = curFn              'User defined fn name.
              End If
            End If
            Case indVar
            If Not iMatch Then
              Mid(equation, fnPos) = Range("G4").Value   'Replace display
              If Not isIVar Then                         'name with internal.
                isIVar = True
                varFound = True
                Range("I4").Value = curFn              'User defined fn name.
              End If
            End If
            Case "("
              lbrac = lbrac + 1
            Case ")"
              rbrac = rbrac + 1
            End Select
          Loop Until (lbrac = rbrac) Or (varFound)
        End If
      End If
```

```
      pos = InStr((pos + 1), equation, "(")      'Look for next function name.
    Loop Until pos = 0
    End If
  End If     'To pos>0.
  Sheets("EntryForm").Select

End Sub


Sub substConst(equation)
Attribute substConst.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by swapFns to replace any constant variables which are used as
'the internal function variables.
Dim fn As String, dmyConst As String

  fn = Range("G4").Value        'Function of indep. var, g(t).
  pos = InStr(equation, fn)
  dmyConst = Range("G17").Value
  Do While pos > 0
    If (pos > 0) And (Mid(equation, pos + 1, 1) <> "(") _
        And (Mid(equation, pos + 1, 1) <> "^") Then      'Ensuring that
      Mid(equation, pos) = dmyConst                      'it is a constant.
    End If
    pos = InStr(pos + 1, equation, fn)
  Loop
  fn = Range("G5").Value        'Function of dependent var, f(y).
  pos = InStr(equation, fn)
  dmyConst = Range("G18").Value
  Do While pos > 0
    If (pos > 0) And (Mid(equation, pos + 1, 1) <> "(") _
        And (Mid(equation, pos + 1, 1) <> "^") Then
      Mid(equation, pos) = dmyConst
    End If
    pos = InStr(pos + 1, equation, fn)
  Loop

End Sub


Function get_fn(equation, bracPos, fnPos) As String
Attribute get_fn.VB_ProcData.VB_Invoke_Func = " \n14"
'Determines whether the string preceding a left parenthesis (whose
'position is indicated by bracPos) is a function and not a transcendental.
'If so, the function name is returned, else a blank is returned.  If a
'valid function name is found, its index in the equation is returned in
'fnPos.
Dim lbrac As Integer, rbrac As Integer
Dim index As Integer, tmpFn As String

  If bracPos > 1 Then                    'In order to be a fn, brackets
                                         'cannot begin in the 1st position.
    If (Mid(equation, bracPos - 1, 1) = "^") Then
      If bracPos > 3 Then
        tmpFn = Mid(equation, bracPos - 3, 2)  'Possibly a transcendental.
      Else
        tmpFn = Mid(equation, bracPos - 2, 1)  'A one character fn name.
      End If
      lbrac = 1
      rbrac = 0
      fnPos = bracPos - 2
      index = bracPos
      Do                                       'Skip bracketted exponent.
        index = index + 1
        Select Case Mid(equation, index, 1)
        Case "("
          lbrac = lbrac + 1
        Case ")"
          rbrac = rbrac + 1
        End Select
      Loop Until (lbrac = rbrac)
      If Mid(equation, index + 1, 1) <> "(" Then    'It is not a function.
```

```
        tmpFn = " "
      Else
        bracPos = index + 1
      End If

    ElseIf is_num(Mid(equation, bracPos - 1, 1)) Then  'Go back before all
      index = bracPos                                  'numbers to look for
      Do While index > 1                               'exponent sign.
        index = index - 1
        If Not (is_num(Mid(equation, index, 1))) Then
          Exit Do
        End If
      Loop
      If Mid(equation, index, 1) = "^" Then
        If index > 2 Then
          tmpFn = Mid(equation, index - 2, 2)        'Possibly a trans. fn.
        Else
          tmpFn = Mid(equation, index - 1, 1)        'A one character fn.
        End If
        fnPos = index - 1
      Else
        tmpFn = " "                                  'Not a function.
      End If
    Else
      If bracPos > 2 Then
        tmpFn = Mid(equation, bracPos - 2, 2)        'Possibly a trans. fn.
      Else
        tmpFn = Mid(equation, bracPos - 1, 1)        'A one character fn.
      End If
      fnPos = bracPos - 1
    End If

    If Len(tmpFn) = 2 Then                   'Determine whether the function
      Select Case tmpFn                      'could be a transcendental or mult op.
      Case "in", "os", "an", "ec", "sc", "xp", "ln", "pi", "* ", " I"
        tmpFn = " "
      Case Else
        Select Case Right(tmpFn, 1)
        Case "*", "+", "-", "/", "^", "(", "=", "I"
        'Ensure that it is not a mult operation, nested brackets,
        'or an equal sign before a parenthesized term.
          tmpFn = " "
        Case Else
          tmpFn = Right(tmpFn, 1)
        End Select
      End Select
    End If
    get_fn = tmpFn

  Else
    get_fn = " "         'Equation begins with a left bracket --> not a fn.
  End If

End Function


Function is_num(data) As Boolean
Attribute is_num.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns true if the text argument is a number.

  If data <> "0" Then
    Select Case Val(data)
      Case 1 To 9
        is_num = True
      Case Else
        is_num = False
    End Select
  Else
    is_num = True
  End If
```

```
End Function


Sub replaceFn(equation, oldVar, newVar)
Attribute replaceFn.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by procedure swapFns to replace all occurrences of a user defined
'function name with that of an equivalent one recognized by the system for
'internal processing.  It is assumed that there is at least one occurrence
'of oldVar to be replaced with newVar in the input equation.
Dim curPos As Integer, nextPos As Integer

  curPos = InStr(equation, oldVar)
  Do
    nextPos = curPos + 1
    If (Mid(equation, nextPos, 1) = "(") Or _
       (Mid(equation, nextPos, 1) = "^") Then 'Ensure that it is a function.
      Mid(equation, curPos) = newVar
    End If
    curPos = InStr(nextPos, equation, oldVar)
  Loop Until curPos = 0

End Sub


Sub spaceSigns(equation)
Attribute spaceSigns.VB_ProcData.VB_Invoke_Func = " \n14"
'As part of the equation preprocessing, this routine removes extra spaces
'between elements while ensuring that operators are one-spaced apart from
'elements.  Called by the preProcess routine.
Dim spacedEqn As String, char As String
Dim pos As Integer, eqnLen As Integer

  pos = 1
  eqnLen = Len(equation)
  Do                                  'Remove any leading blanks.
    char = Mid(equation, pos, 1)
    If char <> " " Then
      spacedEqn = char
      Exit Do
    Else
      pos = pos + 1
    End If
  Loop

  Do                                  'Space out rest of equation.
    pos = pos + 1
    char = Mid(equation, pos, 1)
    If (char <> " ") Then
      If (char = "+") Or (char = "-") Or (char = "*") Then
        spacedEqn = spacedEqn + " " + char + " "
      Else
        spacedEqn = spacedEqn + char
      End If
    End If
  Loop Until pos = eqnLen
  equation = spacedEqn

End Sub


Function get_term(equation) As String
Attribute get_term.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns the first complete term in the equation.
Dim leadOp As Boolean, termFound As Boolean
Dim leftBrac As Integer, rightBrac As Integer
Dim term As String, peek As String

  termFound = False
  leadOp = True                 'A leading operator indicates the beginning
  leftBrac = 0                  'of a term.  LeftBrac & rightBrac keep track
  rightBrac = 0                 'of when a bracketted term ends.  Extracted
```

```
  term = ""                        'elements are accumulated in "term".
  equation = Trim(equation)
  Do While Not termFound
    char = Left(equation, 1)
    Select Case char               'Determine whether this is a bracketed term.
      Case "("
        leftBrac = leftBrac + 1
      Case ")"
        rightBrac = rightBrac + 1
    End Select

    If (char = "+") Or (char = "-") Then
      If term = "" Then            'Element is a leading operator so initialize
        term = char + " "          'term accordingly.
        leadOp = True
        equation = LTrim(Mid(equation, 2))
      Else
        leadOp = False
        If (leftBrac <> rightBrac) Then     'Current char is within the term.
          term = term + char + " "
          equation = LTrim(Mid(equation, 2))
        End If
      End If
    Else                                    'Char is not an operator.
      term = term + char
      equation = Mid(equation, 2)
    End If
    peek = Left(equation, 2)
    If ((Not leadOp) And (leftBrac = rightBrac) And _
       (peek = "+ " Or peek = "- ") And _
       (term <> "-") And (term <> "+")) Or (Len(equation) = 0) Then
      'End of one term has been reached.
       termFound = True
    End If
  Loop
  get_term = term

End Function


'============================================================================
Attribute VB_Name = "Sort"
'--------------------------
Dim demo As Boolean

Sub sortEqn(equation)
Attribute sortEqn.VB_ProcData.VB_Invoke_Func = " \n14"
'Accepts a parsed equation for sorting.

  demo = False   'Turn display of sort on.

  sortMult equation
  sortRest equation

End Sub


Sub do_wait(sec)
'Used for demo to slow down execution of the sort.
Dim newHr As Integer, newMin As Integer
Dim newSec As Integer, waitTime As Variant

  newHr = Hour(Now())
  newMin = Minute(Now())
  newSec = Second(Now()) + sec
  waitTime = TimeSerial(newHr, newMin, newSec)
  Application.Wait waitTime

End Sub
```

```
Function findVars(ByVal term) As Integer
Attribute findVars.VB_ProcData.VB_Invoke_Func = " \n14"
'Determines the type of argument to a transcendental.  The type of
'argument differs depending on whether it contains the dependent
'variable, independent variable, neither, or both.  It is assumed
'that the "term" parameter has had the transcendental name and any
'exponents stripped off and is at the start of the parenthesized
'or unparenthesized argument.  The return codes are as follows:
'                       Code   v1  v2
'                       ----   --  --
'                        1     ""  ""
'                        2     y   ""
'                        3     t   ""
'                        4     y   y
'                        5     y   t
'                        6     t   y
'                        7     t   t

Dim lbrac As Integer, rbrac As Integer
Dim ind_var As String, dep_var As String
Dim var1 As String, var2 As String
Dim arg As String

  lbrac = 0
  rbrac = 0
  arg = Left(term, 1)
  term = Mid(term, 2)
  If arg <> "(" Then                                'Not a bracketted argument.
    Do While Left(term, 1) <> "("
      arg = arg & Left(term, 1)
      term = Mid(term, 2)
    Loop
  Else
    lbrac = 1
    Do While (term <> "") And (lbrac <> rbrac)   'Extract the argument.
      Select Case Left(term, 1)
      Case "("
        lbrac = lbrac + 1
      Case ")"
        rbrac = rbrac + 1
      End Select
      arg = arg & Left(term, 1)
      term = Mid(term, 2)
    Loop
  End If

  If lbrac <> rbrac Then                'Improperly formed argument.
    findVars = 1
  Else
    dep_var = Range("G2").Value
    ind_var = Range("G3").Value
    If InStr(arg, dep_var) = 0 And InStr(arg, ind_var) = 0 Then
      findVars = 1                'Case of no state vars in argument.
    Else
      var1 = ""
      var2 = ""
      Do While arg <> ""
        Select Case Left(arg, 1)
        Case dep_var
          If var1 = "" Then
            var1 = dep_var
          ElseIf var2 = "" Then
            var2 = dep_var
          End If
        Case ind_var
          If var1 = "" Then
            var1 = ind_var
          ElseIf var2 = "" Then
            var2 = ind_var
          End If
        End Select
```

```
        If (var1 <> "" And var2 <> "") Then
          Exit Do
        Else
          arg = Mid(arg, 2)
        End If
      Loop

      var1 = var1 + var2
      Select Case var1
      Case dep_var
        findVars = 2
      Case ind_var
        findVars = 3
      Case dep_var + dep_var
        findVars = 4
      Case dep_var + ind_var
        findVars = 5
      Case ind_var + dep_var
        findVars = 6
      Case ind_var + ind_var
        findVars = 7
      End Select
    End If
  End If

End Function


Sub chkDivExp(term, symbol, lbraclvl, rbraclvl, nestlevel)
Attribute chkDivExp.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sortRest to determine whether there are excess brackets
'enclosing division or exponent terms.  It is assumed that the input
'term contains either "/" or "^" depending upon the symbol parameter.
'Unnecessary brackets are removed in order to comply with the format
'of the equations in the database.  This will increase the chance of
'a successful search.
Dim leftPart As String, rightPart As String
Dim lbrac As Integer, rbrac As Integer
Dim xtraLbrc As Integer, xtraRbrc As Integer
Dim pos As Integer, extra As Integer
Dim leftTerm As String, rightTerm As String
Dim leftHead As String, rightTail As String
Dim sLbrac As Integer, sRbrac As Integer
Dim altSymbol As String

  lbrac = 0
  rbrac = 0
  If (InStr(term, "(") > 0) Or (InStr(term, ")") > 0) Then
    countBracs term, sLbrac, sRbrac
  End If
  leftTerm = ""
  rightTerm = ""
  If symbol = "/" Then
    altSymbol = "^"
  Else
    altSymbol = "/"
  End If
  pos = InStr(term, symbol)
  leftPart = Mid(term, 1, pos - 1)

  If Right(leftPart, 1) = ")" Then  'Move past bracketted subterm.
    Do
      leftTerm = Right(leftPart, 1) + leftTerm
      leftPart = Left(leftPart, Len(leftPart) - 1)
      Select Case Left(leftTerm, 1)
        Case altSymbol
          skipLeftSym leftPart, leftTerm
        Case ")"
          rbrac = rbrac + 1
        Case "("
          lbrac = lbrac + 1
```

```
    End Select
  Loop Until (Right(leftPart, 1) <> altSymbol) And _
             ((leftPart = "") Or (lbrac = rbrac))
End If

If lbrac = rbrac Then              'Look for extra left brackets.
  lbrac = 0
  include = False
  Do While (leftPart <> "")
  'Move backwards from the head of the numerator or exponent base
  'to match brackets.
    Select Case Right(leftPart, 1)
      Case altSymbol
        If Right(leftPart, 2) = (")" + altSymbol) Then
          skipLeftBase leftPart
        End If
      Case ")", " ", symbol
          Exit Do
      Case "("
        If (Right(leftPart, 2) = " (") Or _
           (Right(leftPart, 2) = "((") Or (leftPart = "(") Then
        'precludes cases of transcendental arg or "altsymbol+(".
          lbrac = lbrac + 1
        End If
        If Right(leftPart, 2) = (altSymbol + "(") Then
        'Deduct in case there is an opening bracket to a division
        'term that could be erroneously matched with a parenthesized
        'denominator.
          lbrac = lbrac - 1
        End If

      Case "-"                              'Correct for counting
        If Right(leftPart, 2) = "(-" Then 'bracket required for
          lbrac = lbrac - 1                 'enclosing negative value.
        End If
    End Select
    leftPart = Left(leftPart, Len(leftPart) - 1)
  Loop
End If

If lbrac > 0 Then          'Excess brackets exist in the leftpart.
  xtraLbrc = lbrac
  rightPart = Mid(term, pos + 1)
  lbrac = 0
  rbrac = 0
  If Left(rightPart, 1) = "(" Then
    Do
      rightTerm = rightTerm + Left(rightPart, 1)
      rightPart = Mid(rightPart, 2)
      Select Case Right(rightTerm, 1)
        Case ")"
          rbrac = rbrac + 1
        Case "("
          lbrac = lbrac + 1
      End Select
    Loop Until (rightPart = "") Or (lbrac = rbrac)
    If (Left(rightPart, 1) = altSymbol) Then
      skipRightSym rightPart, rightTerm
    End If
  End If

  If lbrac = rbrac Then   'Look for extra right brackets.
    rbrac = 0
    Do While (rightPart <> "")
    'Move forward from the head of the denominator or exponent
    'to match brackets.
      Select Case Left(rightPart, 1)
        Case altSymbol
          If Left(rightPart, 2) = (altSymbol + "(") Then
            skipRightBase rightPart
          End If
```

```
          Case "(", " ", symbol
            Exit Do
          Case ")"
            If Left(rightPart, 2) <> (")" + altSymbol) Then
              rbrac = rbrac + 1
            End If
            If Left(rightPart, 2) = (")" + altSymbol) And _
               (xtraLbrc = 1) Then
            'Case of a completed exponent base.
              Exit Do
            End If
        End Select
        rightPart = Mid(rightPart, 2)
      Loop
      xtraRbrc = rbrac
    End If

    If xtraLbrc > 0 And xtraRbrc > 0 Then
      If xtraLbrc > xtraRbrc Then
        extra = xtraLbrc - xtraRbrc
      ElseIf xtraLbrc < xtraRbrc Then
        extra = xtraRbrc - xtraLbrc
      Else
        extra = xtraLbrc
      End If

      leftHead = leftPart
      rightTail = rightPart
      leftPart = Mid(term, 1, pos - 1)
      rightPart = Mid(term, pos + 1)
      If leftTerm <> "" Then
        pos = InStr(leftPart, leftTerm)
        leftPart = Mid(leftPart, 1, pos - (extra + 1))
        leftPart = leftPart + leftTerm
      Else
        leftPart = leftHead + Right(leftPart, _
                   Len(leftPart) - Len(leftHead) - extra)
      End If

      If rightTerm <> "" Then
        If rightTail = "" Then
          pos = Len(rightTerm)
          rightPart = Mid(rightPart, pos + 1, _
                          Len(rightPart) - (pos + extra))
          rightPart = rightTerm + rightPart
        Else
          pos = Len(rightTerm)
          rightPart = Mid(rightPart, pos + 1, _
                      Len(rightPart) - Len(rightTail) - (pos + extra))
          rightPart = rightTerm + rightPart + rightTail
        End If
      Else
        rightPart = Mid(rightPart, 1, _
                    (Len(rightPart) - Len(rightTail) - extra))
        rightPart = rightPart + rightTail
      End If

      ActiveCell.Value = leftPart + symbol + rightPart
      countBracs ActiveCell.Value, lbrac, rbrac
      lbraclvl(nestlevel) = lbraclvl(nestlevel) - sLbrac + lbrac
      rbraclvl(nestlevel) = rbraclvl(nestlevel) - sRbrac + rbrac
    End If
  End If

End Sub


Sub skipRightSym(fromTerm, toTerm)
Attribute skipRightSym.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by chkDivExp to move the exponent or denominator element from
'the term being parsed, "fromTerm", to the parenthesized subterm,
```

```
'"toTerm".
Dim lbrac As Integer, rbrac As Integer

  toTerm = toTerm + Left(fromTerm, 1)        'Transfer symbol.
  fromTerm = Mid(fromTerm, 2)
  If Left(fromTerm, 1) <> "(" Then           'Transfer nonbracketted
    toTerm = toTerm + Left(fromTerm, 1)      'subterm.
    fromTerm = Mid(fromTerm, 2)
  Else
    lbrac = 0
    rbrac = 0
    Do                                       'Transfer bracketted
      toTerm = toTerm + Left(fromTerm, 1)    'subterm.
      fromTerm = Mid(fromTerm, 2)
      Select Case Right(toTerm, 1)
        Case "("
          lbrac = lbrac + 1
        Case ")"
          rbrac = rbrac + 1
      End Select
    Loop Until lbrac = rbrac
  End If

End Sub


Sub skipLeftSym(fromTerm, toTerm)
Attribute skipLeftSym.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by chkDivExp to move the exponent base or the numerator
'element from the term being parsed, "fromTerm", to the parenthesized
'subterm, "toTerm".
Dim lbrac As Integer, rbrac As Integer

  If Right(fromTerm, 1) = ")" Then           'Transfer bracketted
    lbrac = 0                                'subterm.
    rbrac = 0
    Do
      toTerm = Right(fromTerm, 1) + toTerm
      fromTerm = Left(fromTerm, Len(fromTerm) - 1)
      Select Case Left(toTerm, 1)
        Case "("
          lbrac = lbrac + 1
        Case ")"
          rbrac = rbrac + 1
      End Select
    Loop Until lbrac = rbrac
  Else                                       'Transfer nonbracketted
    Do                                       'subterm.
      toTerm = Right(fromTerm, 1) + toTerm
      fromTerm = Left(fromTerm, Len(fromTerm) - 1)
    Loop Until (Right(fromTerm, 1) = " ") Or _
               (fromTerm = "") Or (Right(fromTerm, 1) = "(")
  End If

End Sub


Sub skipLeftBase(leftPart)
Attribute skipLeftBase.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by chkDivExp to bypass either the parenthesized exponent base
'or parenthesized numerator in the left side of the term being parsed.
'It is assumed that the parameter, "leftPart", ends either in ")/" or
'")^".
Dim lbrac As Integer, rbrac As Integer

  lbrac = 0
  rbrac = 1
  leftPart = Mid(leftPart, 1, Len(leftPart) - 1)
  Do While (lbrac <> rbrac) And (leftPart <> "")
    leftPart = Left(leftPart, Len(leftPart) - 1)
    Select Case Right(leftPart, 1)
```

```
      Case "("
        lbrac = lbrac + 1
      Case ")"
        rbrac = rbrac + 1
    End Select
  Loop

End Sub


Sub skipRightBase(rightPart)
Attribute skipRightBase.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by chkDivExp to bypass either the parenthesized exponent or
'parenthesized denominator in the left side of the term being parsed.
'It is assumed that the parameter, "rightPart", begins either in "^("
'or "/(".
Dim lbrac As Integer, rbrac As Integer

  lbrac = 1
  rbrac = 0
  rightPart = Mid(rightPart, 2)
  Do While (lbrac <> rbrac) And (rightPart <> "")
    rightPart = Mid(rightPart, 2)
    Select Case Left(rightPart, 1)
      Case "("
        lbrac = lbrac + 1
      Case ")"
        rbrac = rbrac + 1
    End Select
  Loop

End Sub


Sub stripExp(term)
Attribute stripExp.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by getPriority when it encounters an exponent on a constant
'to strip the exponent out of the way in order to determine whether
'it is a numeric constant.  The calling routine has already performed
'preprocessing so that the exponent will be at the head of the term
'that is passed in.
Dim lbrac As Integer, rbrac As Integer

  If Left(term, 1) = "(" Then    'Exponent is bracketted.
    lbrac = 0
    rbrac = 0
    Do
      Select Case Left(term, 1)
      Case "("
        lbrac = lbrac + 1
      Case ")"
        rbrac = rbrac + 1
      End Select
      term = Mid(term, 2)
    Loop Until (lbrac = rbrac)
  Else
    Do                          'Move past the non-bracketted
      term = Mid(term, 2)       'exponent.
    Loop Until (Left(term, 1) = "(") Or (Left(term, 1) = " ") Or (Left(term, 1) = "")
  End If

End Sub


Sub enterList(constChar)
Attribute enterList.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by the constants function to put constants into the constList in the
'defaults sheet for later substitution into the equation following sorting.
'Substitution is required in order to replace constants entered by the user
'with those recognized by the knowledge base.
Dim found As Boolean, fromCell As Object
```

```
  found = False
  Set fromCell = ActiveCell
  Range("constList").Select
  Do While (ActiveCell.Value <> Empty) And (Not found)
    If ActiveCell.Value = constChar Then
      found = True
    Else
      ActiveCell.Offset(1, 0).Select
    End If
  Loop
  If ActiveCell.Value = Empty Then
    ActiveCell.Value = constChar
  End If
  Range(fromCell.Address).Select

End Sub


Function getPriority(curTerm) As Single
Attribute getPriority.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sortSubTerm to determine the priority that should be
'assigned to curTerm.
Dim var1 As String, var2 As String
Dim char As String, match As Boolean
Dim i As Integer, curntPri As Single
Dim pos As Integer, transPri As Single
Dim index As Integer, transIdx As Single

  curntPri = 0
  For i = 1 To 34
    Select Case i
    Case 1 To 5, 20 To 31    't^, t, y^, y, g^, g(t), f^, f(y),
     'pi^, pi, e^, e, and derivatives up to order 5.
      If InStr(curTerm, ActiveCell.Value) > 0 Then
        curntPri = ActiveCell.Offset(0, -1).Value
        Exit For
      End If

    Case 6 To 19           'trans^(v1,v2), trans(v1,v2).
      If i < 13 Then       'trans(v1,v2).
        transArray = Array("arccos", "arcsin", "arctan", "cos", _
                           "csc", "ln", "sec", "sin", "tan")
      Else                 'trans^(v1,v2).
        transArray = Array("arccos^", "arcsin^", "arctan^", "cos^", _
                           "csc^", "exp", "ln^", "sec^", "sin^", "tan^")
      End If
      match = False
      char = curTerm
      transPri = 0
      Do While curTerm <> ""
        transIdx = 0.1
        For Each trans In transArray
          If InStr(curTerm, trans) > 0 Then
            pos = InStr(curTerm, trans)
            If Not (Mid(curTerm, pos + Len(trans), 1) = "^" And _
                    (i < 13)) Then
            'Ensure its not a case of a transcendental with exponent.
              var1 = Mid(curTerm, pos + Len(trans))
              var2 = Mid(curTerm, 1, pos - 1)
              curTerm = var2 + Mid(curTerm, pos + Len(trans))
              match = True
              Exit For
            End If
          End If
          transIdx = transIdx + 0.1
        Next trans

        If match Then
          If InStr(trans, "^") > 0 Then   'Strip non-exp exponents.
            stripExp var1
```

```
        End If

       'Case of an incomplete transcendental is assigned the priority
       'of the transc. with "t&t" args so that it will be updated later.
        If var1 <> "" Then
          index = findVars(var1)
          transPri = ActiveCell.Offset(index - 1, -1).Value + transIdx
        Else
          transPri = ActiveCell.Offset(0, -1).Value - 6
        End If
        If transPri > curntPri Then
          curntPri = transPri
        End If
        match = False                    'Reset to look for next trans.
      Else
        Exit Do
      End If
    Loop
    curTerm = char
    If i < 13 Then     'Jump to next section of table to look for
      i = 12           'transcendentals raised to an exponent.
    Else               'Jump past section for trans with exponent.
      i = 19
    End If
    ActiveCell.Offset(6, 0).Select

    If curntPri > 0 Then       'A priority has been found.
      Exit For
    End If

Case 32              'numeric constant.
    pos = 1
    match = False
    var1 = curTerm
    char = Left(var1, pos)
    Do While char <> ""
      If IsNumeric(char) Then
        curntPri = ActiveCell.Offset(0, -1).Value
        match = True
        Exit Do
      End If
      pos = pos + 1
      char = Mid(var1, pos, 1)
      If char = "^" Then
        index = InStr(var1, "^")
        var1 = Mid(var1, index + 1)
        stripExp var1
        char = Mid(var1, pos, 1)
      End If
    Loop
    If match Then
      Exit For
    End If

Case 33               'alpha^.
    pos = InStr(curTerm, "^")
    If pos > 0 Then
      var1 = Left(curTerm, pos - 1)
      Do While Right(var1, 1) = ")"
        var1 = Left(var1, Len(var1) - 1)
      Loop
      char = Right(var1, 1)              'Internal consts have been
      Select Case char                   'exchanged by Parser.
      Case "a" To "z", "@", "~", "!", "#", "A" To "H", "J" To "Z"
        curntPri = ActiveCell.Offset(0, -1).Value
        enterList char
        Exit For
      End Select
    End If

Case 34               'alpha.
```

```
      pos = 1
      match = False
      char = Left(curTerm, pos)
      Do While char <> ""
        Select Case char
        Case "a" To "z", "@", "~", "!", "#", "A" To "H", "J" To "Z"
          curntPri = ActiveCell.Offset(0, -1).Value
          enterList char
          match = True
          Exit Do
        End Select
        pos = pos + 1
        char = Mid(curTerm, pos, 1)
      Loop
      If match Then
        Exit For
      End If
    End Select
    ActiveCell.Offset(1, 0).Select
  Next i
  getPriority = curntPri

End Function


Function sortSubTerm(priority, sortType) As String
Attribute sortSubTerm.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sortMult and sortRest and backSort to sort the subterm
'elements in the current level of the default sheet.  Returns the
'sorted subterm and the highest priority of the subterm.
Dim lBracCnt As Integer, rBracCnt As Integer
Dim curntPri As Single, highPri As Single
Dim recPri As Single, curTerm As String
Dim head As String, tail As String
Dim col As Integer, termCell As Object

 'Go to last element in column.
  Do While ActiveCell.Value <> ""
    ActiveCell.Offset(1, 0).Select
  Loop
  If (ActiveCell.Row > 1) Then
    ActiveCell.Offset(-1, 0).Select
  End If
  If Right(ActiveCell.Value, 1) = ")" Then  'May need to remove
    curTerm = ActiveCell.Value              'trailing bracket(s)
    lBracCnt = 0                            'on the last term.
    rBracCnt = 0
    tail = ""
    Do While curTerm <> ""
      Select Case Left(curTerm, 1)
        Case "("
          lBracCnt = lBracCnt + 1
        Case ")"
          rBracCnt = rBracCnt + 1
          tail = tail + ")"
      End Select
      curTerm = Mid(curTerm, 2)
    Loop
    If rBracCnt > lBracCnt Then
      With ActiveCell
        .Value = Mid(.Value, 1, (Len(.Value) - (rBracCnt - lBracCnt)))
      End With
      If Len(tail) > (rBracCnt - lBracCnt) Then
        tail = Mid(tail, 1, (rBracCnt - lBracCnt))
      End If
    Else
      tail = ""
    End If
  End If

  With ActiveSheet
```

```
  col = ActiveCell.Column
  .Cells(1, col).Select
  If Left(ActiveCell.Value, 1) = "(" Then 'May need to remove
    curTerm = ActiveCell.Value            'leading bracket(s)
    lBracCnt = 0                          'on the first term.
    rBracCnt = 0
    head = ""
    Do While curTerm <> ""
      Select Case Left(curTerm, 1)
        Case "("
          lBracCnt = lBracCnt + 1
          head = head + "("
        Case ")"
          rBracCnt = rBracCnt + 1
      End Select
      curTerm = Mid(curTerm, 2)
    Loop
    If lBracCnt > rBracCnt Then
      ActiveCell.Value = Mid(ActiveCell.Value, (lBracCnt - rBracCnt + 1))
      If Len(head) > (lBracCnt - rBracCnt) Then
        head = Mid(head, 1, (lBracCnt - rBracCnt))
      End If
    Else
      head = ""
    End If
  End If
End With

If sortType = 1 Then
  recPri = ActiveCell.Offset(0, -1).Value
Else
  recPri = ActiveCell.Offset(0, -2).Value
End If
highPri = recPri
curTerm = ActiveCell.Value
Do While curTerm <> ""

  If demo Then
    Application.ScreenUpdating = False
  End If

  Set termCell = ActiveCell
  If sortType = 1 Then                'A sortType of 1 indicates
    ActiveSheet.Cells(4, 30).Select   'the mult. index is to be used,
  Else                                'else the add/minus/deriv index
    ActiveSheet.Cells(48, 30).Select  'table is to be referenced.
  End If
  curntPri = getPriority(curTerm)     'Get associated priority for
                                      'the current subterm.

  Range(termCell.Address).Select
  If Not (sortType = 2 And InStr(curTerm, "y'") > 0) Then
    If (curntPri > recPri) Then
      If sortType = 1 Then
        ActiveCell.Offset(0, -1).Value = curntPri
      Else
      'Ensure that for consecutive sequences of numeric or alpha
      'constants, those that involve multiplication are placed
      'ahead of addition/subtraction.  This enforces a structure
      'in a series of constants.
        If curntPri > 31 Then
        'Alpha and numeric constants in the addition/subtraction
        'sort phase have priorities 32, 33, 34.
          If InStr(curTerm, "*") > 0 Then
          'Reduce priority of multiplicative constant sequences.
            curntPri = curntPri - 0.5
          End If
        End If
        ActiveCell.Offset(0, -2).Value = curntPri
      End If
    End If
```

```
    If (recPri > highPri) Then
      highPri = recPri
    End If
  Else    'A derivative for a "+"/"-" sort will have a lower index
          'than other elements.  Overwrite any record priority.
    ActiveCell.Offset(0, -2).Value = curntPri
    recPri = curntPri
  End If
  If (curntPri > highPri) Then
    highPri = curntPri
  End If

  ActiveCell.Offset(1, 0).Select      'Advance to next cell.
  If sortType = 1 Then
    recPri = ActiveCell.Offset(0, -1).Value
  Else
    recPri = ActiveCell.Offset(0, -2).Value
  End If
  curTerm = ActiveCell.Value

  If demo Then
    Application.ScreenUpdating = True
    do_wait 2
  End If

Loop

If sortType = 1 Then      'Sort and collect terms for multiplication.
  col = col - 1
  Columns(col).Select
  If ActiveCell.Offset(1, 0).Value <> "" Then
  'Only sort if it is there is more than one element.

    ActiveCell.Offset(0, 0).Columns("A:B").EntireColumn.Select

    If demo Then
      do_wait 1
    End If

    'Next, sort by index, then by alpha.
    Selection.Sort Key1:=ActiveCell, Order1:=xlAscending, Key2:=ActiveCell. _
      Offset(0, 1).Range("A1"), Order2:=xlAscending, Header:=xlNo, OrderCustom _
      :=1, MatchCase:=False, Orientation:=xlTopToBottom
  End If

  If demo Then
    do_wait 2
  End If

  Sheets("defaults").Cells(1, (col + 1)).Select
  curTerm = ActiveCell.Value
  ActiveCell.Offset(1, 0).Select
  Do While ActiveCell.Value <> ""       'Collect subterms.

    If demo Then
      do_wait 1
    End If

    curTerm = curTerm + " * " & ActiveCell.Value
    ActiveCell.Offset(1, 0).Select
  Loop

Else          'Sort and collect terms for add/minus/derivatives.
  col = col - 2
  Columns(col).Select
  If ActiveCell.Offset(1, 0).Value <> "" Then
  'No need to sort for one node subterm.

    ActiveCell.Offset(0, 0).Columns("A:C").EntireColumn.Select

    If demo Then
```

```
        do_wait 1
      End If

      'Next, sort by index, then by alpha.
      Selection.Sort Key1:=ActiveCell, Order1:=xlAscending, Key2:=ActiveCell. _
        Offset(0, 2).Range("A1"), Order2:=xlAscending, Header:=xlNo, OrderCustom _
        :=1, MatchCase:=False, Orientation:=xlTopToBottom
    End If

    If demo Then
      do_wait 2
    End If

    'If starting element is negative, then move behind a positive element.
    moveMinus col
    Sheets("defaults").Cells(1, (col + 2)).Select
    curTerm = ActiveCell.Value

    'Initialize first subterm with its operator and advance to next node.
    curTerm = ActiveCell.Offset(0, -1).Value + " " + curTerm
    ActiveCell.Offset(1, 0).Select
    Do While ActiveCell.Value <> ""       'Collect subterms.

      If demo Then
        do_wait 1
      End If

      curTerm = curTerm + " " + ActiveCell.Offset(0, -1).Value + _
                " " & ActiveCell.Value
      ActiveCell.Offset(1, 0).Select
    Loop
  End If

 'Ensure any leading sign appears ahead of outside left bracket.
  curTerm = LTrim(curTerm)
  If head <> "" Then
    If Left(curTerm, 1) = "+" Then
      curTerm = LTrim(Mid(curTerm, 2))
      curTerm = "+ " + head + curTerm
    Else
      curTerm = head + curTerm
    End If
  End If
  If tail <> "" Then
    curTerm = curTerm + tail
  End If

  Columns(col).Select                    'Erase work area.
  If sortType = 1 Then
    ActiveCell.Offset(0, 0).Columns("A:B").EntireColumn.Select
    Selection.Clear
    ActiveCell.Select
  Else
    ActiveCell.Offset(0, 0).Columns("A:C").EntireColumn.Select
    Selection.Clear
    ActiveCell.Offset(0, 2).Select
  End If
  priority = highPri
  sortSubTerm = curTerm

End Function


Sub countBracs(ByVal subterm, lbrac, rbrac)
Attribute countBracs.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by adjustBracs, SortTerm, backSort, and backConnect to find
'the number of left and right brackets in an individual row element.
'Also called by chkDivExp to count the number of left and right
'parentheses remaining following the removal of superfluous brackets.

  lbrac = 0
```

```
  rbrac = 0
  Do While Len(subterm) > 0
    If Left(subterm, 1) = "(" Then
      lbrac = lbrac + 1
    ElseIf Left(subterm, 1) = ")" Then
      rbrac = rbrac + 1
    End If
    subterm = Mid(subterm, 2)
  Loop

End Sub


Sub stepDown(lbraclvl, rbraclvl, nestlevel)
Attribute stepDown.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sort during the initial equation parsing and sorting
'stage to go back down one level due to the completion of a subterm.
'This routine updates the bracket counts for levels n and n+1, clears
'the contents of the higher level, and brings the cursor to the first
'first empty cell in level n.

  lbraclvl(nestlevel - 1) = lbraclvl(nestlevel - 1) + lbraclvl(nestlevel)
  rbraclvl(nestlevel - 1) = rbraclvl(nestlevel - 1) + rbraclvl(nestlevel)
  lbraclvl(nestlevel) = 0
  rbraclvl(nestlevel) = 0
  Range("lvl1Elt").Select
  ActiveCell.Offset(0, (2 * nestlevel) - 2).Select
  ActiveCell.ClearContents                          'Remove current
  ActiveCell.Offset(0, -1).ClearContents            'subterm.
  nestlevel = nestlevel - 1
  Range("lvl1Elt").Select
  ActiveCell.Offset(0, (2 * nestlevel) - 2).Select
  Do While ActiveCell.Value <> ""                   'Get to first
    ActiveCell.Offset(1, 0).Select                  'blank in previous
  Loop                                              'column.

End Sub


Sub stepUp(lbraclvl, rbraclvl, nestlevel)
Attribute stepUp.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sortMult to go up a level in order to accomodate the
'change in operator to "*".  This also entails the moving up of
'the last element in the previous level.  This new level for mult.
'is required if the current level contains any operands that are
'followed by either a "+" or "-" operator.
Dim tmpPri As Single, subterm As String

  tmpPri = ActiveCell.Offset(0, -1).Value    'Requires moving
  subterm = RTrim(ActiveCell.Value)          'current cell value
  ActiveCell.Offset(0, -1).ClearContents     'to next level.
  ActiveCell.ClearContents
  adjustBracs subterm, lbraclvl, rbraclvl, nestlevel
  nestlevel = nestlevel + 1
  Range("lvl1Elt").Select
  ActiveCell.Offset(0, (2 * nestlevel) - 2).Select
  ActiveCell.Offset(0, -1).Value = tmpPri
  ActiveCell.Value = subterm

End Sub


Sub adjustBracs(ByVal subterm, lbraclvl, rbraclvl, nestlevel)
Attribute adjustBracs.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by stepUp to keep track of the parentheses when terms are
'moved up one level by either the occurrence of "*" or a "+"
'following a "-" with a "(".
Dim lbrac As Integer, rbrac As Integer

  countBracs subterm, lbrac, rbrac
  lbraclvl(nestlevel + 1) = lbraclvl(nestlevel + 1) + lbrac
```

```
  rbraclvl(nestlevel + 1) = rbraclvl(nestlevel + 1) + rbrac
  lbraclvl(nestlevel) = lbraclvl(nestlevel) - lbrac
  rbraclvl(nestlevel) = rbraclvl(nestlevel) - rbrac

End Sub


Function append(ByVal term) As Boolean
Attribute append.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sortTerm to determine whether the preceding element in a
'cell is a transcendental, a function of a state variable, an exponent
'sign, or a division operator and hence, any open parenthesis should
'begin on the same line.
Dim lbrac As Integer, rbrac As Integer
Dim app As Boolean

  app = False
  If ActiveCell.Value <> "" Then
   'First, check on the case of trans^()(....
    If Right(term, 1) = ")" Then
      lbrac = 0
      rbrac = 1
      term = Mid(term, 1, Len(term) - 1)

     'Move past the bracketted exponent.
      Do While Len(term) > 0
        If Right(term, 1) = "(" Then
          lbrac = lbrac + 1
        ElseIf Right(term, 1) = ")" Then
          rbrac = rbrac + 1
        End If
        term = Mid(term, 1, Len(term) - 1)
        If lbrac = rbrac Then
          Exit Do
        End If
      Loop
    End If

   'Move past any nonbracketted exponent.
    If (InStr(term, "^") > 0) And (Right(term, 1) <> "^") Then
      Do While Right(term, 1) <> "^"
        term = Mid(term, 1, Len(term) - 1)
      Loop
    End If

   'Determine whether it is an exponent.
    If Right(term, 1) = "^" Then
      app = True
    Else
   'Check for case of transcendental without exponent.
      term = Right(term, 3)
      Select Case term
        Case "exp", "sin", "cos", "tan", "sec", "csc"
          app = True
        Case Else
          If (InStr(term, "ln") > 0) Or (Right(term, 1) = "f") Or _
             (Right(term, 1) = "g") Or (Right(term, 1) = "/") Or _
             (Right(term, 1) = "I") Or (Right(term, 1) = "(") Then
            app = True
          End If
      End Select
    End If
  End If
  append = app

End Function


Sub simplify(apLvl, lbraclvl, rbraclvl, nestlevel, sType)
Attribute simplify.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sortMult (sType=1) or sortRest (sType=2) when the number
```

```
'of left parentheses equal the number of right parentheses in a
'multiplication level.  This routine will determine whether the
'conditions for removing leading and trailing brackets are satisfied
'and if so, to remove them accordingly.
Dim lbrac As Integer, rbrac As Integer
Dim firstCell As Object, lastCell As Object
Dim lastLvlOk As Boolean

  If nestlevel > 1 Then
    If apLvl(nestlevel - 1) Then          'May still simplify if last
      Range("lvl1Elt").Select             'level ends in "(".
      If sType = 1 Then
        ActiveCell.Offset(0, (2 * (nestlevel - 1)) - 2).Select
      Else
        ActiveCell.Offset(0, (3 * (nestlevel - 1)) - 2).Select
      End If
      lastElement
      If Right(ActiveCell.Value, 1) = "(" Then
        lastLvlOk = True
      Else
        lastLvlOk = False
      End If
      Range("lvl1Elt").Select             'Return to current level.
      If sType = 1 Then
        ActiveCell.Offset(0, (2 * nestlevel) - 2).Select
      Else
        ActiveCell.Offset(0, (3 * nestlevel) - 2).Select
      End If
      lastElement
    Else
      lastLvlOk = True
    End If
  Else
    lastLvlOk = False
  End If

  If lastLvlOk Then
    Do While ActiveCell.Row <> 1        'Go to first node.
      ActiveCell.Offset(-1, 0).Select
    Loop
    countBracs ActiveCell.Value, lbrac, rbrac
    If (Left(ActiveCell.Value, 1) = "(" And lbrac > rbrac) Then
      Set firstCell = ActiveCell
      lastElement
      countBracs ActiveCell.Value, lbrac, rbrac

      If (Right(ActiveCell.Value, 1) = ")" And rbrac > lbrac) And _
         (InStr(Left(term, 2), "^") = 0 And InStr(Left(term, 2), "/") = 0) Then
      'Current level is enclosed in parentheses and is not an
      'exponent base or numerator so beginning and ending brackets
      'may be removed.

        Set lastCell = ActiveCell
        Do While (Left(firstCell.Value, 1) = "(") And _
                 (Right(lastCell.Value, 1) = ")")
          firstCell.Value = Mid(firstCell.Value, 2)
          lbraclvl(nestlevel) = lbraclvl(nestlevel) - 1
          lastCell.Value = Left(lastCell.Value, Len(lastCell.Value) - 1)
          rbraclvl(nestlevel) = rbraclvl(nestlevel) - 1
        Loop
      End If
    ElseIf (ActiveCell.Offset(1, 0).Value = Empty) And (lbrac > 0 And _
            lbrac = rbrac) And (Not apLvl(nestlevel - 1)) Then
    'Consider the case of a bracketted expression contained in
    'the first (and only) node of the level.

      Do While (Left(ActiveCell.Value, 1) = "(") And _
               (Right(ActiveCell.Value, 1) = ")")
        ActiveCell.Value = Mid(ActiveCell.Value, 2)
        ActiveCell.Value = Left(ActiveCell.Value, _
                                Len(ActiveCell.Value) - 1)
```

```
        lbraclvl(nestlevel) = lbraclvl(nestlevel) - 1
        rbraclvl(nestlevel) = rbraclvl(nestlevel) - 1
      Loop

    End If
  End If

End Sub


Sub sortMult(term)
Attribute sortMult.VB_ProcData.VB_Invoke_Func = " \n14"
'A term here is any sequence of multiplicative elements or elements
'enclosed in parenthesis and it may contain subterms within brackets.
'Processing of a term is from left to right; however, subterms will
'be sorted first before the enclosing parent term is.
Dim char As String, startOp As String
Dim tmpPri As Single, subterm As String
Dim lbrac As Integer              'Separate bracket counters required
Dim rbrac As Integer              'to track individual row elements
                                  'rather than at the nestlevel.
Dim idx As Integer
Dim elements(30) As String
Dim maxlevel As Integer
Dim nestlevel As Integer          'Indicate level of nesting.
Dim rbraclvl(1 To 20) As Integer  'Keep track of parentheses count at each
Dim lbraclvl(1 To 20) As Integer  'level.  Indicator of completed term.
Dim multLvl(1 To 20) As Boolean   'Indicates whether a level is a mult. level.
Dim apLvl(1 To 20) As Boolean     'Indicates if elt at a level is a trans,
                                  'fn of state var, a denominator, or a
                                  'variable raised to an exponent arg.
Dim lastMult As Boolean
Dim hasDeriv As Boolean

  term = Trim(term)
  If Len(Trim(term)) > 1 Then   'Bypass trivial case of one element term.
    Sheets("defaults").Select

    If demo Then
      Application.ScreenUpdating = True
    End If

    Range("lvl1Elt").Select

    nestlevel = 1               'Start at the first level and check for
                                'beginning operator or left bracket.
    If Left(term, 1) = "+" Or Left(term, 1) = "-" Or _
       Left(term, 1) = "(" Then
      If Left(term, 1) = "+" Or Left(term, 1) = "-" Then
        startOp = Left(term, 1)
      Else
        lbraclvl(1) = 1
        ActiveCell.Value = Left(term, 1)
      End If
      term = Mid(LTrim(term), 2)
    End If

    If demo Then
     do_wait 1
    End If

    char = Left(term, 1)
    Do While Len(term) > 0          'Break down term to be sorted.
      Select Case char
      Case "("                          'Need to start next nest level.
        If Not apLvl(nestlevel) Then
          apLvl(nestlevel) = append(ActiveCell.Value)
        End If
        If (apLvl(nestlevel)) Or (ActiveCell.Value <> "") Or _
           (ActiveCell.Row <> 1) Then   'Not currently at the start
                                        'of a new level.
```

```
    nestlevel = nestlevel + 1      'Begin next nest level.
    multLvl(nestlevel) = False
    Range("lvl1Elt").Select
    ActiveCell.Offset(0, (2 * nestlevel) - 2).Select
  End If
  ActiveCell.Value = ActiveCell.Value & char
  lbraclvl(nestlevel) = lbraclvl(nestlevel) + 1

Case ")"                           'A subterm is completed.
  If ActiveCell.Value = "" Then
    ActiveCell.Offset(-1, 0).Select
  End If
  rbraclvl(nestlevel) = rbraclvl(nestlevel) + 1
  Selection.NumberFormat = "@"
  ActiveCell.Value = ActiveCell.Value & char

  If (multLvl(nestlevel)) Then
    If (InStr(Left(term, 2), "/") = 0 And _
        InStr(Left(term, 2), "^") = 0) And _
       (lbraclvl(nestlevel) = rbraclvl(nestlevel)) Then
    'Could separate multiplicative numerator elts, but this
    'is not done in order to follow the usual method of
    'representation commonly practiced.

    'Remove unnecessary leading and trailing parentheses.
      simplify apLvl, lbraclvl, rbraclvl, nestlevel, 1
      lastElement
    End If

    countBracs ActiveCell.Value, lbrac, rbrac
    If nestlevel > 1 Then
      If multLvl(nestlevel - 1) Then
        lastMult = True
      Else
        lastMult = False
      End If
    Else
      lastMult = False
    End If
    If (InStr(Left(term, 5), "*") = 0 And _
        lastMult = False) Or (rbrac > lbrac) Then
   'Sort to one subterm only if the next and previous operator
   'is not multiplication or if there is a trailing right bracket.

      subterm = sortSubTerm(tmpPri, 1)     'Assign priority
                                           'and sort.
     'Return term and priority to current level.
      ActiveCell.Value = tmpPri
      ActiveCell.Offset(0, 1).Select
      Selection.NumberFormat = "@"
      ActiveCell.Value = subterm
      multLvl(nestlevel) = False

      If demo Then
        do_wait 2
      End If

    ElseIf multLvl(nestlevel - 1) Then
   'Need to transfer current mult elements down one level.
      Range("lvl1Elt").Select
      ActiveCell.Offset(0, (2 * nestlevel) - 2).Select
      idx = 0
      Do While ActiveCell.Value <> ""
        elements(idx) = ActiveCell.Value
        ActiveCell.ClearContents
        ActiveCell.Offset(0, -1).ClearContents
        ActiveCell.Offset(1, 0).Select
        idx = idx + 1
      Loop

     'Go down 1 level without the need to adjust brackets (there
```

```
    'are none at the current level) or apLvl.
     multLvl(nestlevel) = False
     nestlevel = nestlevel - 1
     Range("lvl1Elt").Select
     ActiveCell.Offset(0, (2 * nestlevel) - 2).Select
     Do While ActiveCell.Value <> ""
       ActiveCell.Offset(1, 0).Select    'Go to first empty cell.
     Loop
     If apLvl(nestlevel) Then            'This case occurs for
       ActiveCell.Offset(-1, 0).Select   'level n-1 with "(" as
     End If                              'its last element.
     For i = 0 To (idx - 1)              'Put elts in level n-1.

       If demo Then
         do_wait 2
       End If

       If i = 0 Then
         If apLvl(nestlevel) Then
           ActiveCell.Value = ActiveCell.Value & elements(i)
           apLvl(nestlevel) = False
         Else
           ActiveCell.Value = elements(i)
         End If
       Else
         ActiveCell.Value = elements(i)
       End If
       ActiveCell.Offset(1, 0).Select
     Next i
   End If

ElseIf ActiveCell.Row > 1 Then          'Combine a non-mult. level.
   combine subterm

   If demo Then
     do_wait 2
   End If

End If

'Next, may need to collapse successive columns with completed
'subterms.
Do While (nestlevel > 1) And ((lbraclvl(nestlevel) > 0 And _
          lbraclvl(nestlevel) = rbraclvl(nestlevel)) Or _
          (rbraclvl(nestlevel) - lbraclvl(nestlevel) = 1)) And _
          (ActiveCell.Row = 1)

'Either a bracketted subterm or a "closing" subterm, i.e.
'one with an extra right bracket, is complete.

   If (Not multLvl(nestlevel - 1)) Or (multLvl(nestlevel - 1) And _
       lbraclvl(nestlevel) = rbraclvl(nestlevel)) Then

     'Collapse down one level only if last level is not a mult
     'level or if last level is a mult level and current level
     'has balanced parentheses.

     If (Len(ActiveCell.Value) < 7) And (Left(ActiveCell.Value, 1) = "(") _
        And (Right(ActiveCell.Value, 1) = ")") Then
       simplify apLvl, lbraclvl, rbraclvl, nestlevel, 1
     End If
     subterm = ActiveCell.Value
     tmpPri = ActiveCell.Offset(0, -1).Value

     'Collapse completed subterm down one level.
     stepDown lbraclvl, rbraclvl, nestlevel

     'Check for division operation or opening left bracket or the
     'return of a transcendental or function of state var argument
     'or a variable with an exponent argument.
     If ActiveCell.Row <> 1 Then
```

```
        ActiveCell.Offset(-1, 0).Select
      End If

      If (apLvl(nestlevel)) Then
        ActiveCell.Value = ActiveCell.Value & subterm
        If tmpPri > ActiveCell.Offset(0, -1).Value Then
          ActiveCell.Offset(0, -1).Value = tmpPri
        End If
        apLvl(nestlevel) = False
      Else

     'Enter into the next empty cell.
        If ActiveCell.Value <> "" Then
          ActiveCell.Offset(1, 0).Select
        End If
        ActiveCell.Value = subterm
        ActiveCell.Offset(0, -1).Value = tmpPri
      End If

      If demo Then
        do_wait 2
      End If

     'Re-sort or combine the level if this entry causes
     'a closing term.
      countBracs subterm, lbrac, rbrac
      If (ActiveCell.Row <> 1) And (rbrac - lbrac = 1) Then
        If multLvl(nestlevel) Then

          If lbraclvl(nestlevel) = rbraclvl(nestlevel) Then
          'Remove unnecessary leading and trailing parentheses.
            simplify apLvl, lbraclvl, rbraclvl, nestlevel, 1
            lastElement
            countBracs ActiveCell.Value, lbrac, rbrac
          End If
          If (InStr(Left(term, 3), "*") = 0 And _
              multLvl(nestlevel - 1) = False) Or (rbrac > lbrac) Then
         'Sort to one subterm only if the next operator is not
         'multiplication or there is a trailing bracket.

            subterm = sortSubTerm(tmpPri, 1)
            Range("lvllElt").Select
            ActiveCell.Offset(0, (2 * nestlevel) - 2).Select
            ActiveCell.Value = subterm
            ActiveCell.Offset(0, -1).Value = tmpPri
            multLvl(nestlevel) = False
          End If
        Else
          combine subterm
        End If

        If demo Then
          do_wait 2
        End If

      End If
    Else
      Exit Do
    End If
  Loop

Case "-", "+"
  If (Left(term, 2) = "- ") Or (char = "+") Then
  'A subtraction or addition operation rather than the
  'beginning of a negative element.

    If multLvl(nestlevel) Then
    'Currently at a multiplication level.  This change of sign
    'forces a sort for the mult expression and the current
    'level is reset to a non-mult level.
```

```
      subterm = sortSubTerm(tmpPri, 1)
      ActiveCell.Value = tmpPri
      ActiveCell.Offset(0, 1).Select
      Selection.NumberFormat = "@"
      ActiveCell.Value = subterm + " "
      multLvl(nestlevel) = False

      If demo Then
        do_wait 2
      End If

    End If
    If Right(ActiveCell.Value, 1) <> " " Then
      ActiveCell.Value = ActiveCell.Value & " " & char & " "
    Else
      ActiveCell.Value = ActiveCell.Value & char & " "
    End If
    ActiveCell.Offset(1, 0).Select    'Shouldn't need this.

  ElseIf (Left(term, 2) <> "- ") And (char <> "+") Then
  'Case of a negative value.
    ActiveCell.Value = ActiveCell.Value + char
  End If

Case "/"
  If multLvl(nestlevel) Then
  'Should sort the numerator prior to appending the operator
  'if it is a multlevel.  The level that is appended with "/"
  'retains its multlvl status after sorting.

    hasDeriv = False
    If (ActiveCell.Row > 1) Then
    'First determine whether a derivative is part of the
    'numerator.  If it is, the level is not sorted.
      i = ActiveCell.Row + 1
      Do While i > 1
        If InStr(ActiveCell.Offset((i - 1) - ActiveCell.Row, _
                 0).Value, "'") > 0 Then
          hasDeriv = True
          Exit Do
        End If
        i = i - 1
      Loop
    End If

    If Not hasDeriv Then
    'If the previous level is a mult level, then transfer
    'current level elements down.  The numerator will
    'include the elements of the previous level.  Collapse
    'continues until level n-1 is not a mult level.

      Do While multLvl(nestlevel - 1) And _
         Not (apLvl(nestlevel - 1))
      'Go to node 1 of current level to determine whether
      'an opening left bracket needs to be transferred to
      'node 1 of level n-1.
        Range("lvl1Elt").Select
        ActiveCell.Offset(0, (2 * nestlevel) - 2).Select

        If Left(ActiveCell.Value, 1) = "(" Then
          ActiveCell.Value = Mid(ActiveCell.Value, 2)
          lbraclvl(nestlevel - 1) = _
            lbraclvl(nestlevel - 1) + lbraclvl(nestlevel)
          rbraclvl(nestlevel - 1) = _
            rbraclvl(nestlevel - 1) + rbraclvl(nestlevel)
          lbraclvl(nestlevel) = 0
          rbraclvl(nestlevel) = 0

          Range("lvl1Elt").Select
          ActiveCell.Offset(0, (2 * (nestlevel - 1)) - 2).Select
          ActiveCell.Value = "(" + ActiveCell.Value
```

```
          Range("lvl1Elt").Select
          ActiveCell.Offset(0, (2 * nestlevel) - 2).Select
        End If

        idx = 0
        Do While ActiveCell.Value <> ""
          elements(idx) = ActiveCell.Value
          ActiveCell.ClearContents
          ActiveCell.Offset(0, -1).ClearContents
          ActiveCell.Offset(1, 0).Select
          idx = idx + 1
        Loop
        multLvl(nestlevel) = False
        nestlevel = nestlevel - 1
        Range("lvl1Elt").Select
        ActiveCell.Offset(0, (2 * nestlevel) - 2).Select
        Do While ActiveCell.Value <> ""
          ActiveCell.Offset(1, 0).Select    'Go to first empty cell.
        Loop

        For i = 0 To (idx - 1)              'Put elts in level n-1.

          If demo Then
            do_wait 1
          End If

          ActiveCell.Value = elements(i)
          ActiveCell.Offset(1, 0).Select
        Next i
      Loop

      subterm = sortSubTerm(tmpPri, 1)
      ActiveCell.Value = tmpPri
      ActiveCell.Offset(0, 1).Select
      Selection.NumberFormat = "@"
      ActiveCell.Value = subterm

      If demo Then
        do_wait 1
      End If

    End If
  End If
  ActiveCell.Value = ActiveCell.Value & char

Case "*"
  If (ActiveCell.Row = 1) And (Right(ActiveCell.Value, 1) <> "+") _
     And (Right(ActiveCell.Value, 1) <> "-") And _
     (Left(ActiveCell.Value, 2) <> " +") And _
     (Left(ActiveCell.Value, 2) <> " -") And _
     (Not multLvl(nestlevel)) Then
   'This is a level with one entry not followed or preceded by "+"
   'or "-" operator and hence, may be designated as a mult. level.
    multLvl(nestlevel) = True
    ActiveCell.Offset(1, 0).Select

  Else
    If Not (multLvl(nestlevel)) Then
   'Current level is not suitable to be a mult. level.
      If (Left(ActiveCell.Value, 2) = " +") Or _
         (Left(ActiveCell.Value, 2) = " -") Then
        nestlevel = nestlevel + 1 'Will this case occur?
        multLvl(nestlevel) = True
        Range("lvl1Elt").Select
        ActiveCell.Offset(0, (2 * nestlevel) - 2).Select

      Else
      'Only transfer element up one level if it does not
      'begin with a +/- operator.
        stepUp lbraclvl, rbraclvl, nestlevel
        ActiveCell.Offset(1, 0).Select
```

```
            multLvl(nestlevel) = True
          End If
        ElseIf ActiveCell.Value <> "" Then
          ActiveCell.Offset(1, 0).Select
        End If
      End If

    Case " "
      If (ActiveCell.Value <> "") And (Not multLvl(nestlevel) And _
          InStr(Left(term, 3), "*") = 0) Then
        ActiveCell.Value = ActiveCell.Value & " "
      End If

    Case Else                          'Build up current element.
      ActiveCell.Value = ActiveCell.Value & char
    End Select

    term = Mid(term, 2)
    char = Left(term, 1)

    If demo Then
      Application.ScreenUpdating = True
      do_wait 1
    End If
  Loop

  If (nestlevel > 1) Or _
     (Range("lvl1Elt").Offset(1, 0).Value <> "") Then
    backSort apLvl, multLvl, lbraclvl, rbraclvl, nestlevel, maxlevel
    backConnect apLvl, multLvl, lbraclvl, rbraclvl, maxlevel
  End If

  If demo Then
    do_wait 5
  End If

  term = ActiveCell.Value
  ActiveCell.ClearContents
  ActiveCell.Offset(0, -1).ClearContents
  If startOp <> "" Then
    term = startOp & term
  End If
 End If

 If demo Then
   Beep
   Application.ScreenUpdating = False
 End If

End Sub


Sub lastElement()
Attribute lastElement.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by combine, simplify, sortRest, and sortMult (also
'backSort and backConnect from alternative sort method) to reach
'the last element in the current level.

 Do While ActiveCell.Value <> ""
   ActiveCell.Offset(1, 0).Select
 Loop
 If ActiveCell.Row > 1 Then
   ActiveCell.Offset(-1, 0).Select
 End If

End Sub


Sub combine(subterm)
Attribute combine.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sortMult to wind up the elements in a non-multiplication
```

```
'level.

  lastElement
  subterm = ""
  Do While ActiveCell.Row <> 1
    subterm = ActiveCell.Value & subterm
    ActiveCell.ClearContents
    ActiveCell.Offset(0, -1).ClearContents
    ActiveCell.Offset(-1, 0).Select
  Loop
  subterm = ActiveCell.Value & subterm
  ActiveCell.Value = subterm

End Sub


Function hasFn(subterm) As Boolean
Attribute hasFn.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sortRest to determine whether the argument to a state
'or transcendental function also contains a state or transcendental
'function.
Dim match As Boolean

  fnArray = Array("arccos", "arcsin", "arctan", "cos", "csc", _
                  "ln", "sec", "sin", "tan", "exp", "f", "g", "I")
  match = False
  For Each fn In fnArray
    If InStr(subterm, fn) > 0 Then
      match = True
      Exit For
    End If
  Next fn
 hasFn = match

End Function


Sub sortRest(term)
Attribute sortRest.VB_ProcData.VB_Invoke_Func = " \n14"
'Called as the final step in sortTerm to re-order terms that have
'subtraction elements.  Only addition and subtraction elements are
'considered and only a left parenthesis can begin a new level.  A
'level is always collapsed if it is a complete subterm.  Closing
'subterms cannot occur in this sorting.  Multiplication and division
'elements are ignored.  Processing of a term is from left to right;
'however, subterms will be sorted first before the enclosing parent
'term is.
Dim char As String, sign As String
Dim tmpPri As Single, subterm As String
Dim lbrac As Integer              'Separate bracket counters required
Dim rbrac As Integer              'to track individual row elements
                                  'rather than at the nestlevel.
Dim apLvl(1 To 20) As Boolean     'Indicates if elt at current nestlevel is
                                  'a trans, fn of state var, a denominator,
                                  'or a variable raised to an exponent arg.
Dim nestlevel As Integer          'Indicate level of nesting.
Dim rbraclvl(1 To 20) As Integer  'Keep track of parentheses count at each
Dim lbraclvl(1 To 20) As Integer  'level.  Indicator of completed term.
Dim lvlCell As Object, neg_mult As Boolean
Dim elements(20) As String, idx As Integer
Dim signs(20) As String, topSign As String
Dim endLbrac As Integer, endRbrac As Integer
Dim isLbrac As Boolean, aConst As String
Dim update As Boolean

  Sheets("defaults").Select

  If demo Then
    Application.ScreenUpdating = True
  End If
```

```
Range("lvl1Elt").Select
nestlevel = 1

'Set the first element as positive or negative.
If Left(term, 1) = "-" Then
  ActiveCell.Value = "-"
  term = Mid(LTrim(term), 2)
Else
  ActiveCell.Value = "+"
End If
ActiveCell.Offset(0, 1).Select

'Strip first left parenthesis.
If Left(term, 1) = "(" Then
  lbraclvl(1) = 1
  ActiveCell.Value = Left(term, 1)
  term = Mid(LTrim(term), 2)
End If

char = Left(term, 1)
Do While Len(term) > 0          'Break down term to be sorted.
  Select Case char
  Case "("                              'Need to start next nest level.
    If Not apLvl(nestlevel) Then
      apLvl(nestlevel) = append(ActiveCell.Value)
    End If
    If InStr(Right(ActiveCell.Value, 2), "**") > 0 Then
      apLvl(nestlevel) = True
    End If
    If (apLvl(nestlevel)) Or (ActiveCell.Value <> "") Or _
       (ActiveCell.Row <> 1) Then
   'Begin next level if not currently at the start of a new level.
      nestlevel = nestlevel + 1
      Range("lvl1Elt").Select
      ActiveCell.Offset(0, (3 * nestlevel) - 2).Select
    End If
    ActiveCell.Value = ActiveCell.Value & char
    If ActiveCell.Offset(0, -1).Value = "" Then
      ActiveCell.Offset(o, -1).Value = "+"
    End If
    lbraclvl(nestlevel) = lbraclvl(nestlevel) + 1

  Case ")"                              'A subterm is completed.
    If ActiveCell.Value = "" Then
      ActiveCell.Offset(-1, 0).Select
    End If
    rbraclvl(nestlevel) = rbraclvl(nestlevel) + 1
    Selection.NumberFormat = "@"
    ActiveCell.Value = ActiveCell.Value & char
    If (Right(ActiveCell.Value, 2) <> "))") Or _
       (ActiveCell.Row <> 1) Then
     'Ensures that it is not a second close to a completed subterm.

      If nestlevel > 1 Then
        Set lvlCell = ActiveCell          'Verify that it is not a
        Range("lvl1Elt").Select           'negative or mult term.
        ActiveCell.Offset(0, (3 * (nestlevel - 1)) - 2).Select
        ActiveCell.Offset(0, -1).Select   'Must look for last sign.
        lastElement
        If (ActiveCell.Value = "-") Or _
           (InStr(Right(ActiveCell.Offset(0, 1).Value, 2), "**") > 0) Then
          neg_mult = True
        Else
          neg_mult = False
        End If

        If ActiveCell.Offset(0, 1).Value = "(" Then
          isLbrac = True
        Else
          isLbrac = False
        End If
```

```
  Range(lvlCell.Address).Select
Else
  neg_mult = True
End If

If (neg_mult = False) And (InStr(Left(term, 2), "/") = 0) And _
   (InStr(Left(term, 2), "^") = 0) And _
   (InStr(Left(term, 3), "*") = 0) And _
   (lbraclvl(nestlevel) = rbraclvl(nestlevel)) Then
'May only remove parentheses if it is not a negative term, a
'numerator, an exponent, or a multiplicative factor.

  simplify apLvl, lbraclvl, rbraclvl, nestlevel, 2
End If

If nestlevel > 1 Then            'Extract apLvl value without
  If apLvl(nestlevel - 1) Then   'overstepping lower boundary.
    neg_mult = True              'Also applies to denom and
  Else                           'exponent subterms.
    neg_mult = False
  End If
Else
  neg_mult = False
End If

Do While ActiveCell.Row <> 1     'Go to first element.
  ActiveCell.Offset(-1, 0).Select
Loop
countBracs ActiveCell.Value, lbrac, rbrac
If (nestlevel = 1) And (lbrac > 0) And _
   (ActiveCell.Offset(0, -1).Value = "-") Then
  topSign = "-"
  ActiveCell.Offset(0, -1).Value = "+"
Else
  topSign = ""
End If
lastElement
countBracs ActiveCell.Value, endLbrac, endRbrac
If (lbrac > 0 And endRbrac > 0 And InStr(Left(term, 5), "+") = 0 And _
    InStr(Left(term, 5), "-") = 0) Or (endRbrac > endLbrac) Or _
    (neg_mult And lbrac > 0 And _
     rbraclvl(nestlevel) = lbraclvl(nestlevel)) Then
'Sort if the next operator is not +/- or if there is a
'trailing right bracket.  If rbrac=0, then should copy all
'elts to lower level for a more complete sort.

  subterm = sortSubTerm(tmpPri, 2)   'Assign priority and sort.

 'Return term and priority to current level.
  ActiveCell.Offset(0, -2).Value = tmpPri
  If (Left(subterm, 1) = "-" Or topSign = "-") Then
    If Left(subterm, 1) = "-" And isLbrac Then
    'Case of a negative element beginning a bracketted term.
    'The space between the negative sign and the elt is
    'removed here.
      subterm = "-" + Mid(subterm, 3)
    Else
      ActiveCell.Offset(0, -1).Value = "-"
    End If
  Else
    ActiveCell.Offset(0, -1).Value = "+"
  End If
  If ((Left(subterm, 1) = "-") And _
     (Not isLbrac)) Or (Left(subterm, 1) = "+") Then
    'Remove sign only if it is not a negative elt.
    subterm = LTrim(Mid(subterm, 2))
  End If
  Selection.NumberFormat = "@"
  ActiveCell.Value = subterm

  If demo Then
```

```
    do_wait 2
  End If

 'Remove excess brackets enclosing division and exponent terms.
  If InStr(ActiveCell.Value, "/") And _
     (Left(term, 2) <> ")^") Then

    If nestlevel > 1 Then
      If Not apLvl(nestlevel - 1) Then
      'This condition ensures transcendental arguments do
      'not have their parentheses removed.
        chkDivExp ActiveCell.Value, "/", lbraclvl, rbraclvl, nestlevel
      End If
    Else  'At level 1 which cannot be collapsed.
      chkDivExp ActiveCell.Value, "/", lbraclvl, rbraclvl, nestlevel
    End If
  End If
  If InStr(ActiveCell.Value, "^") And _
     (Left(term, 2) <> ")/") Then
    If nestlevel > 1 Then
      If Not apLvl(nestlevel - 1) Then
        chkDivExp ActiveCell.Value, "^", lbraclvl, rbraclvl, nestlevel
      End If
    Else
      chkDivExp ActiveCell.Value, "^", lbraclvl, rbraclvl, nestlevel
    End If
  End If

ElseIf (lbrac = 0 Or lbrac = rbrac) Then     'Transfer elts back to level n-1.
  If ActiveCell.Row = 1 Then
    If InStr(ActiveCell.Value, "/") And _
       (Left(term, 2) <> ")^") Then

      If nestlevel > 1 Then
        If Not apLvl(nestlevel - 1) Then
        'This condition ensures transcendental arguments do
        'not have their parentheses removed.
          chkDivExp ActiveCell.Value, "/", lbraclvl, rbraclvl, nestlevel
        End If
      Else  'At level 1 which cannot be collapsed.
        chkDivExp ActiveCell.Value, "/", lbraclvl, rbraclvl, nestlevel
      End If
    End If

    If InStr(ActiveCell.Value, "^") And _
       (Left(term, 2) <> ")/") Then
      If nestlevel > 1 Then
        If Not apLvl(nestlevel - 1) Then
          chkDivExp ActiveCell.Value, "^", lbraclvl, rbraclvl, nestlevel
        End If
      Else
        chkDivExp ActiveCell.Value, "^", lbraclvl, rbraclvl, nestlevel
      End If
    End If
  End If

  Range("lvl1Elt").Select
  ActiveCell.Offset(0, (3 * nestlevel) - 2).Select
  idx = 0
  Do While ActiveCell.Value <> ""
    elements(idx) = ActiveCell.Value
    signs(idx) = ActiveCell.Offset(0, -1).Value
    ActiveCell.ClearContents
    ActiveCell.Offset(0, -1).ClearContents
    ActiveCell.Offset(0, -2).ClearContents
    ActiveCell.Offset(1, 0).Select
    idx = idx + 1
  Loop

 'Go down 1 level without the need to adjust brackets (there
 'are none at the current level) or apLvl.
```

```
    nestlevel = nestlevel - 1
    Range("lvl1Elt").Select
    ActiveCell.Offset(0, (3 * nestlevel) - 2).Select
    Do While ActiveCell.Value <> ""
      ActiveCell.Offset(1, 0).Select    'Go to first empty cell.
    Loop
    If apLvl(nestlevel) Then            'This case occurs for
      ActiveCell.Offset(-1, 0).Select   'level n-1 with "(" as
    End If                              'its last element.

    For i = 0 To (idx - 1)              'Put elts in level n-1.

      If demo Then
        do_wait 1
      End If

      If ActiveCell.Offset(0, -1).Value <> "-" Then
        ActiveCell.Offset(0, -1).Value = signs(i)
      End If
      If i = 0 Then
        If apLvl(nestlevel) Then
          ActiveCell.Value = ActiveCell.Value & elements(i)
          apLvl(nestlevel) = False
        Else
          ActiveCell.Value = elements(i)
        End If
      Else
        ActiveCell.Value = elements(i)
      End If
      ActiveCell.Offset(1, 0).Select
    Next i
  End If
Else
  If ActiveCell.Row = 1 Then
    If (InStr(ActiveCell.Value, "/") > 0) Or _
       (InStr(ActiveCell.Value, "^") > 0) Then
      Range("lvl1Elt").Select
      ActiveCell.Offset(0, (3 * (nestlevel - 1)) - 2).Select
      lastElement
      If Right(ActiveCell.Value, 2) = "* " Then
        neg_mult = True
      Else
        neg_mult = False
      End If
      Range("lvl1Elt").Select
      ActiveCell.Offset(0, (3 * nestlevel) - 2).Select
    End If

    If InStr(ActiveCell.Value, "/") And _
       (Left(term, 2) <> ")^") Then

      If nestlevel > 1 Then
        If (Not apLvl(nestlevel - 1)) Or (neg_mult) Then
        'This condition ensures transcendental arguments do
        'not have their parentheses removed.
          chkDivExp ActiveCell.Value, "/", lbraclvl, rbraclvl, nestlevel
        End If
      Else  'At level 1 which cannot be collapsed.
        chkDivExp ActiveCell.Value, "/", lbraclvl, rbraclvl, nestlevel
      End If
    End If

    If InStr(ActiveCell.Value, "^") And _
       (Left(term, 2) <> ")/") Then
      If nestlevel > 1 Then
        If (Not apLvl(nestlevel - 1)) Or (neg_mult) Then
          chkDivExp ActiveCell.Value, "^", lbraclvl, rbraclvl, nestlevel
        End If
      Else
        chkDivExp ActiveCell.Value, "^", lbraclvl, rbraclvl, nestlevel
      End If
```

```
    End If
  End If
End If

If demo Then
  do_wait 1
End If

'Next, may need to collapse successive columns with completed
'subterms.
If nestlevel > 1 Then
Do While (nestlevel > 1) And (ActiveCell.Row = 1) And _
         (lbraclvl(nestlevel) = rbraclvl(nestlevel) Or _
          (apLvl(nestlevel - 1) And Left(ActiveCell.Value, 1) <> "(" _
           And Right(ActiveCell.Value, 1) = ")"))
'A bracketted subterm is complete or it is part of a bracketted
'term which began with a negative element (handled by last "or"
 'condition).

  subterm = ActiveCell.Value
  sign = ActiveCell.Offset(0, -1).Value
  tmpPri = ActiveCell.Offset(0, -2).Value

 'Collapse completed subterm down one level.
  goDown lbraclvl, rbraclvl, nestlevel

  If demo Then
    do_wait 2
  End If

 'Check for "/" or "*" operation or leading left bracket or the
 'return of a transcendental or function of state var argument
 'or a variable with an exponent argument.
  If ActiveCell.Row <> 1 Then
    ActiveCell.Offset(-1, 0).Select
  End If
  If (apLvl(nestlevel)) Or _
     (Right(ActiveCell.Value, 1) = "(") Then
    If (sign = "-") And (sign <> ActiveCell.Offset(0, -1).Value) Then
      subterm = sign + subterm
    End If

    update = False
    If (tmpPri > ActiveCell.Offset(0, -2).Value) Then
      If (Not hasFn(subterm)) Then
        Select Case Right(ActiveCell.Value, 1)
          Case "c", "n", "p", "s", "f", "g", ")", "I"
            'Subterm is a state or transcendental function
            'argument and the priority of the function will
            'be assigned later.
            update = True
          Case Else
            Select Case Right(ActiveCell.Value, 2)
              Case "c^", "n^", "s^", "g^", "f^"
                'Subterm is the power of a state or transc
                'function (raised to the power).  The priority
                'of the function will be assigned later.
                update = True
              Case Else
                If tmpPri < 28 Then
                'Exclude priorities of constants.
                  ActiveCell.Offset(0, -2).Value = tmpPri
                Else
                  'Check for the existence of a function.
                  update = hasFn(ActiveCell.Value)
                End If
            End Select
        End Select
      Else
        'The priority is assigned even if it is an argument.
        'But only if the argument itself contains a state or
```

```
      'transcendental function and the priority is not
      'referring to a state variable or constant.
      If tmpPri < 24 Then
        ActiveCell.Offset(0, -2).Value = tmpPri
      Else
        'Check for the existence of a function.
        update = hasFn(ActiveCell.Value)
      End If
    End If
  Else
    If (ActiveCell.Offset(0, -2).Value > 27) And _
       (tmpPri < 28) Then
    'Use priority of nonconstants over those of constants.
    'This provides greater precision in the sort.
      ActiveCell.Offset(0, -2).Value = tmpPri
    End If
  End If
  ActiveCell.Value = ActiveCell.Value & subterm

  If demo Then
    do_wait 2
  End If

  If update Then

    If demo Then
      Application.ScreenUpdating = False
    End If

    subterm = ActiveCell.Value
    Set lvlCell = ActiveCell
    ActiveSheet.Cells(48, 30).Select      'Table to be referenced.
    tmpPri = getPriority(subterm)
    Range(lvlCell.Address).Select
    If (tmpPri < 28) And _
        tmpPri > ActiveCell.Offset(0, -2).Value Then
      ActiveCell.Offset(0, -2).Value = tmpPri
    End If

    If demo Then
      Application.ScreenUpdating = True
      do_wait 2
    End If

  End If
  apLvl(nestlevel) = False

Else

 'Enter into the next empty cell and re-sort the column
 'if this entry causes a complete subterm.
  If ActiveCell.Value <> "" Then
    ActiveCell.Offset(1, 0).Select
  End If
  ActiveCell.Offset.Value = subterm
  If ActiveCell.Offset(0, -1) = "" Then
    ActiveCell.Offset(0, -1).Value = sign
  End If
  ActiveCell.Offset(0, -2).Value = tmpPri
  countBracs subterm, lbrac, rbrac

  If demo Then
    do_wait 2
  End If

  If nestlevel > 1 Then
    Set lvlCell = ActiveCell          'Verify that it is not a
    ActiveCell.Offset(0, -4).Select   'negative or mult term.
    lastElement
    If (ActiveCell.Value = "-") Or _
       (InStr(Right(ActiveCell.Offset(0, 1).Value, 2), "*") > 0) Then
```

```
    neg_mult = True
  Else
    neg_mult = False
  End If
  Range(lvlCell.Address).Select
Else
  neg_mult = True
End If

If (neg_mult = False) And (lbraclvl(nestlevel) = rbraclvl(nestlevel)) _
    And (InStr(Left(term, 3), "**") = 0) Then
'Remove unnecessary leading and trailing parentheses.
  simplify apLvl, lbraclvl, rbraclvl, nestlevel, 2
  lastElement
  countBracs ActiveCell.Value, lbrac, rbrac
End If

If (ActiveCell.Row <> 1) And (rbrac - lbrac = 1) Then
  subterm = sortSubTerm(tmpPri, 2)
  ActiveCell.Offset(0, -2).Value = tmpPri
  If Left(subterm, 1) = "-" Then
    ActiveCell.Offset(0, -1).Value = "-"
  Else
    ActiveCell.Offset(0, -1).Value = "+"
  End If
  subterm = LTrim(Mid(subterm, 2))
  Selection.NumberFormat = "@"
  ActiveCell.Value = subterm

  If demo Then
    do_wait 2
  End If

Else
  Do While ActiveCell.Row <> 1
    ActiveCell.Offset(-1, 0).Select
  Loop
  countBracs ActiveCell.Value, endLbrac, endRbrac
  lastElement
  If (nestlevel > 1 And rbrac = 0 And endLbrac = endRbrac) Then
    Range("lvl1Elt").Select
    ActiveCell.Offset(0, (3 * nestlevel) - 2).Select
    idx = 0
    Do While ActiveCell.Value <> ""
      elements(idx) = ActiveCell.Value
      signs(idx) = ActiveCell.Offset(0, -1).Value
      ActiveCell.ClearContents
      ActiveCell.Offset(0, -1).ClearContents
      ActiveCell.Offset(0, -2).ClearContents
      ActiveCell.Offset(1, 0).Select
      idx = idx + 1
    Loop

   'Go down 1 level without the need to adjust brackets (there
   'are none at the current level) or apLvl.
    nestlevel = nestlevel - 1
    Range("lvl1Elt").Select
    ActiveCell.Offset(0, (3 * nestlevel) - 2).Select
    Do While ActiveCell.Value <> ""
      ActiveCell.Offset(1, 0).Select    'Go to first empty cell.
    Loop
    For i = 0 To (idx - 1)              'Put elts in level n-1.

      If demo Then
        do_wait 1
      End If

      ActiveCell.Offset(0, -1).Value = signs(i)
      ActiveCell.Value = elements(i)
      ActiveCell.Offset(1, 0).Select
    Next i
```

```
        End If
      End If
    End If
   'Remove excess brackets enclosing division and exponent terms.
    If InStr(ActiveCell.Value, "/") And (Left(term, 2) <> ")^") Then
      chkDivExp ActiveCell.Value, "/", lbraclvl, rbraclvl, nestlevel
    End If
    If InStr(ActiveCell.Value, "^") And (Left(term, 2) <> ")/") Then
      chkDivExp ActiveCell.Value, "^", lbraclvl, rbraclvl, nestlevel
    End If

    If nestlevel = 1 Then
      Exit Do
    End If
  Loop
  End If

Case "/", "^"                          'Always append "/" or "^"
  If ActiveCell.Value = "" Then        'to a non-empty node.
    ActiveCell.Offset(-1, 0).Select
  End If

  'Put exponent base or numerator into constant list if its
  'a constant.
  aConst = Right(ActiveCell.Value, 1)
  If (aConst <> "y") And (aConst <> "t") And _
     (aConst <> "f") And (aConst <> "g") And _
     (Not preDefined(Right(ActiveCell.Value, 3), 1)) Then
    Select Case aConst
      Case "a" To "z", "~", "@", "!", "#", "A" To "H", "J" To "Z"

        If demo Then
          Application.ScreenUpdating = False
        End If

        enterList (Right(ActiveCell.Value, 1))

        If demo Then
          Application.ScreenUpdating = True
        End If

    End Select
  End If
  ActiveCell.Value = ActiveCell.Value & char

Case "*"                               'Append "*" to a non-
  If ActiveCell.Value = "" Then        'empty cell with spacing.
    ActiveCell.Offset(-1, 0).Select
  End If
  ActiveCell.Value = ActiveCell.Value & " " & char & " "

 'Update the priority with what is currently in the
 'multiplication expression unless it is a constant.

  If demo Then
    Application.ScreenUpdating = False
  End If

  subterm = ActiveCell.Value
  Set lvlCell = ActiveCell
  ActiveSheet.Cells(48, 30).Select      'Table to be referenced.
  tmpPri = getPriority(subterm)
  Range(lvlCell.Address).Select
  If (tmpPri < 28) And tmpPri > ActiveCell.Offset(0, -2).Value Then
    ActiveCell.Offset(0, -2).Value = tmpPri
  End If

  If demo Then
    Application.ScreenUpdating = True
  End If
```

```
  Case "-"
    If (Left(term, 2) = "- ") And (ActiveCell.Value <> "(") Then
      ActiveCell.Offset(0, -1).Value = "-"
    ElseIf ActiveCell.Value = "(" Then
    'Case of a bracketted term that begins with a negative element.
      apLvl(nestlevel) = True
      nestlevel = nestlevel + 1
      Range("lvllElt").Select
      ActiveCell.Offset(0, (3 * nestlevel) - 2).Select
      ActiveCell.Offset(0, -1).Value = "-"
    Else
    'The beginning of a nonbracketted negative element rather than
    'a subtraction operation.
      ActiveCell.Value = ActiveCell.Value & char
    End If

  Case "+"
    ActiveCell.Offset(0, -1).Value = "+"

  Case " "                          'Current term element completed.
    If (InStr(Right(ActiveCell.Value, 2), "**") = 0) And _
       (ActiveCell.Value <> "") And (ActiveCell.Value <> "(") Then
      ActiveCell.Offset(1, 0).Select    'Advance to next cell.
    End If

  Case Else                         'Build up current element.
    If (Right(ActiveCell.Value, 1) = "/") And (char <> "y") And _
       (char <> "t") And (char <> "f") And (char <> "g") And _
       (char <> "(") And Not preDefined(Left(term, 3), 1) Then
    'Put denominator constant into constant list.
      If Not is_num(char) Then

        If demo Then
          Application.ScreenUpdating = False
        End If

        enterList char

        If demo Then
          Application.ScreenUpdating = True
        End If

      End If
    End If
    ActiveCell.Value = ActiveCell.Value & char

  End Select

  term = Mid(term, 2)
  char = Left(term, 1)

  If demo Then
    Application.ScreenUpdating = True
    do_wait 1
  End If
Loop

If (nestlevel = 1) And _
   (Range("lvllElt").Offset(1, 0).Value <> "") Then
  subterm = sortSubTerm(tmpPri, 2)
  If (Left(subterm, 1) = "+") Then
    subterm = LTrim(Mid(subterm, 2))
  End If
  ActiveCell.Value = subterm
End If
If ActiveCell.Row <> 1 Then
  ActiveCell.Offset(-1, 0).Select
End If
term = ActiveCell.Offset(0, -1).Value + ActiveCell.Value

If demo Then
```

```
    do_wait 5
  End If

  ActiveCell.ClearContents
  ActiveCell.Offset(0, -1).ClearContents
  ActiveCell.Offset(0, -2).ClearContents

  If demo Then
    Beep
    Application.ScreenUpdating = False
  End If

End Sub


Sub goDown(lbraclvl, rbraclvl, nestlevel)
Attribute goDown.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sortRest to go back down one level due to the completion
'of a subterm.  This routine updates the bracket counts for levels n
'and n+1, clears the contents of the higher level, and brings the
'cursor to the first empty cell in level n.

  lbraclvl(nestlevel - 1) = lbraclvl(nestlevel - 1) + lbraclvl(nestlevel)
  rbraclvl(nestlevel - 1) = rbraclvl(nestlevel - 1) + rbraclvl(nestlevel)
  lbraclvl(nestlevel) = 0
  rbraclvl(nestlevel) = 0
  Range("lvl1Elt").Select
  ActiveCell.Offset(0, (3 * nestlevel) - 3).Select
  ActiveCell.ClearContents                          'Remove current
  ActiveCell.Offset(0, -1).ClearContents            'subterm.
  ActiveCell.Offset(0, 1).ClearContents
  nestlevel = nestlevel - 1
  Range("lvl1Elt").Select
  ActiveCell.Offset(0, (3 * nestlevel) - 2).Select
  Do While ActiveCell.Value <> ""                   'Get to first
    ActiveCell.Offset(1, 0).Select                  'blank in previous
  Loop                                              'column.

End Sub


Sub moveMinus(level)
Attribute moveMinus.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by sortRest if a negative elements is sorted to be at the
'head of a term.  This routine moves the first positive element to
'be at the head of the term.
Dim priArray(1 To 20) As Single
Dim signArray(1 To 20) As String
Dim termArray(1 To 20) As String
Dim i As Integer, j As Integer
Dim noPos As Boolean

  Sheets("defaults").Cells(1, (level + 1)).Select
  noPos = False
  If ActiveCell.Value = "-" Then
    Do While ActiveCell.Value <> "+"     'Advance to first positive element.
      ActiveCell.Offset(1, 0).Select
      If ActiveCell.Value = "" Then
        noPos = True
        Exit Do
      End If
    Loop

    If Not noPos Then
    i = 1
    priArray(i) = ActiveCell.Offset(0, -1).Value
    signArray(i) = ActiveCell.Value
    termArray(i) = ActiveCell.Offset(0, 1).Value
    ActiveCell.Offset(0, -1).Value = "x"

    Sheets("defaults").Cells(1, (level + 1)).Select
```

```
    Do While ActiveCell.Value <> ""
      If ActiveCell.Offset(0, -1).Value <> "x" Then
        i = i + 1
        priArray(i) = ActiveCell.Offset(0, -1).Value
        signArray(i) = ActiveCell.Value
        termArray(i) = ActiveCell.Offset(0, 1).Value
      End If
      ActiveCell.Offset(1, 0).Select
    Loop

    'Overwrite old list.
    Sheets("defaults").Cells(1, (level + 1)).Select
    For j = 1 To i
      ActiveCell.Offset(0, -1).Value = priArray(j)
      ActiveCell.Value = signArray(j)
      ActiveCell.Offset(0, 1).Value = termArray(j)
      ActiveCell.Offset(1, 0).Select
    Next j
    End If
  End If

End Sub


================================================================================
Attribute VB_Name = "PostSort"
--------------------------------

Function get_constPos(ByVal equation, constID) As Integer
Attribute get_constPos.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by routine SwapConst in order to find the position in the sorted
'equation of each unique display constant that were found by the sorting
'process.  This function returns the position of the first occurrence of
'constID once it has verified that it is not part of a transcendental
'word string.
Dim subterm1 As String, subterm2 As String
Dim subterm3 As String, pos As Integer
Dim index As Integer

  index = 0
  subterm1 = ""
  subterm2 = ""
  subterm3 = ""
  Do
    pos = InStr(equation, constID)              'Isolate all 3 character
    index = index + pos                         'substrings surrounding
    subterm1 = Mid(equation, pos, 3)            'the constant sought in
    If pos > 1 Then                             'order to ensure that it
      subterm2 = Mid(equation, (pos - 1), 3)    'is not part of a trans-
      If (pos - 1) > 1 Then                     'cendental.  This is
        subterm3 = Mid(equation, (pos - 2), 3)  'verified via the case
      End If                                    'construct.
    End If
    Select Case constID
    Case "a"
      If (InStr(subterm1, "arc") = 0) And (InStr(subterm2, "tan") = 0) Then
        Exit Do
      End If
    Case "c"
      If (InStr(subterm1, "cos") = 0) And (InStr(subterm1, "csc") = 0) And _
         (InStr(subterm3, "sec") = 0) And (InStr(subterm3, "arc") = 0) And _
         (InStr(subterm3, "csc") = 0) Then
        Exit Do
      End If
    Case "e"
      If (InStr(subterm1, "exp") = 0) And (InStr(subterm2, "sec") = 0) Then
        Exit Do
      End If
    Case "i"
      If (InStr(subterm2, "sin") = 0) And (InStr(subterm2, "pi") = 0) Then
        Exit Do
```

```
      End If
    Case "l"
      If (InStr(subterm1, "ln") = 0) Then
        Exit Do
      End If
    Case "n"
      If (InStr(subterm2, "ln") = 0) And (InStr(subterm3, "sin") = 0) And _
         (InStr(subterm3, "tan") = 0) Then
        Exit Do
      End If
    Case "o"
      If (InStr(subterm2, "cos") = 0) Then
        Exit Do
      End If
    Case "p"
    If (InStr(subterm1, "pi") = 0) And (InStr(subterm3, "exp") = 0) Then
        Exit Do
      End If
    Case "r"
      If (InStr(subterm2, "arc") = 0) Then
        Exit Do
      End If
    Case "s"
      If (InStr(subterm1, "sin") = 0) And (InStr(subterm1, "sec") = 0) And _
         (InStr(subterm2, "csc") = 0) And (InStr(subterm3, "cos") = 0) Then
        Exit Do
      End If
    Case "t"
      If (InStr(subterm1, "tan") = 0) Then
        Exit Do
      End If
    Case "x"
      If (InStr(subterm2, "exp") = 0) Then
        Exit Do
      End If
    Case "I"         'Symbol for Integral.
      Exit Do
    End Select
    equation = Mid(equation, (pos + 1))
  Loop
  get_constPos = index

End Function


Sub swapConst(equation)
Attribute swapConst.VB_ProcData.VB_Invoke_Func = " \n14"
'The constants entered by the user must be exchanged with those
'used in the database.  This can only take place after the equation
'has been sorted so that constants can be substituted in sequential
'order into the equation search string (without the occurrence of
'further sorting to change the ordering).  Any constants found by
'the Sort routine has been entered into the constList in the defaults
'sheet.  This routine searches for the position of the constList
'elements in the equation and replaces them in order with internal
'constants.
Dim constChar As String, constIdx As Integer
Dim pos As Integer, tempStr As String
Dim restConst As String, allSorted As Boolean

 'First, remove the enclosing parentheses required by the Sorter.
  pos = InStr(equation, "(")    'Eqn may begin with a sign or "(".
  equation = Mid(equation, pos + 1)
  equation = Left(equation, Len(equation) - 1)

  Sheets("defaults").Select
  Range("constList").Select
  Do While ActiveCell.Value <> Empty
    pos = 0                                          'Reset pos to search for
    Do                                               'next display constant.
      pos = InStr(pos + 1, equation, ActiveCell.Value)
```

```
    If Not preDefined(equation, pos) Then          'Get the first position
      ActiveCell.Offset(0, -3).Value = pos         'of each unique constant
      Exit Do                                      'occurring in the sorted
    End If                                         'equation.
  Loop
  ActiveCell.Offset(1, 0).Select                   'Advance to next const.
Loop

If pos > 0 Then                                    'Sort constants by their
  Range("constPos").Select                         'position in the equation
  If ActiveCell.Offset(1, 0).Value <> Empty Then   'before substituting.
    Selection.End(xlDown).Offset(0, 3).Select
    ActiveWorkbook.Names.add Name:="lastConst", RefersToR1C1:= _
              Application.ActiveWindow.ActiveCell
    Range("constPos:lastConst").Activate
    Selection.Sort Key1:=ActiveCell, Order1:=xlAscending, Header:= _
    xlGuess, OrderCustom:=1, MatchCase:=False, Orientation:=xlTopToBottom
  End If

  Range("constList").Select
  'ConstArray must have symbols prefixed with "_" to distinguish
  'internal constants with user defined ones that are the same
  'and which occur later on in the equation.  For example, this
  'avoids the exchange problem for "t(b)^(a) * t/b".
  constArray = Array("_a", "_b", "_c", "_d", "_h", "_k", "_m", "_n", "_q")

  constIdx = -1
  Do While ActiveCell.Value <> Empty               'Exchange display consts
    constIdx = constIdx + 1                        'with internal constants
    constChar = constArray(constIdx)              'stored in constArray.
    pos = InStr(ActiveCell.Offset(0, -3).Value, _
                equation, ActiveCell.Value)
    Do
      'Check that the position is not (1) a symbol that is part
      'of a predefined name e.g. a transcendental name, and
      '(2) a constant that has already been exchanged.

      If pos > 1 Then
        If (Not preDefined(equation, pos)) And _
           (Mid(equation, pos - 1, 1) <> "_") Then
        'Substitute display constant with internal.
          tempStr = Left(equation, pos - 1) + constChar
          equation = tempStr + Mid(equation, pos + 1)
        End If
      Else
        equation = constChar + Mid(equation, pos + 1)
      End If
      'Search and substitute any additional same display constant.
      pos = InStr((pos + 1), equation, ActiveCell.Value)

    Loop Until pos = 0
    ActiveCell.Offset(1, 0).Select                 'Advance to the next
  Loop                                             'display constant.
End If

'Must remove the "_" prefix of the internal constants.
pos = InStr(equation, "_")
allSorted = False
Do While pos > 0
  tempStr = Mid(equation, 1, pos - 1)

  If Not allSorted Then
    'If there are only constants left in the equation, ensure
    'that they are sorted in alphabetical order now that they
    'have been replaced with the internal variables.

    restConst = Mid(equation, pos)
    If (InStr(restConst, "y") = 0) And _
       (InStr(restConst, "t") = 0) Then

      restConst = "(" + restConst + ")"
```

```
      sortMult restConst

      lbrac = 0
      rbrac = 0
      i = Len(restConst)
      j = 1
      Do While j <= i
        Select Case Mid(restConst, j, 1)
          Case "("
            lbrac = lbrac + 1
          Case ")"
            rbrac = rbrac + 1
        End Select
        j = j + 1
      Loop

      If lbrac > rbrac Then
        i = lbrac - rbrac
        Do While i > 0
          restConst = restConst + ")"
          i = i - 1
        Loop
      End If
      If rbrac > lbrac Then
        i = rbrac - lbrac
        Do While i > 0
          restConst = "(" + restConst
          i = i - 1
        Loop
      End If

      sortRest restConst
      If (Left(restConst, 1) = "+") Or (Left(restConst, 1) = "-") Then
        restConst = Mid(restConst, 2)
      End If
      allSorted = True

      If lbrac > rbrac Then
        i = lbrac - rbrac
      End If

      If rbrac > lbrac Then
        i = rbrac - lbrac
        restConst = Mid(restConst, i + 1)
      End If

      restConst = Mid(restConst, 3, Len(restConst) - 3)
      equation = tempStr + restConst

    Else
      equation = tempStr + Mid(equation, pos + 1)
    End If
  Else
    equation = tempStr + Mid(equation, pos + 1)
  End If

  pos = InStr((pos + 1), equation, "_")
Loop

Range("constList").Select
If ActiveCell.Value <> Empty Then
'Copy display constant list to the constant section.

  If ActiveCell.Offset(1, 0).Value <> Empty Then
    Range("constList:lastConst").Select
  End If
  Selection.Copy
  Range("I6").PasteSpecial xlPasteValues

  Range("constList").Select
  If ActiveCell.Offset(1, 0).Value <> Empty Then
```

```
      Range("constList:lastConst").Select
    End If
    Selection.Clear

    Range("ConstPos").Select                'Clear positions index.
    If ActiveCell.Offset(1, 0).Value <> Empty Then
      Selection.End(xlDown).Select
      ActiveWorkbook.Names.add Name:="lastConst", RefersToR1C1:= _
                Application.ActiveWindow.ActiveCell
      Range("constPos:lastConst").Activate
      Selection.Clear
      ActiveWorkbook.Names("lastConst").Delete
    Else
      ActiveCell.Clear
    End If
  End If

End Sub


================================================================================
Attribute VB_Name = "AddEdit"
------------------------------
Dim dbaseChg As Boolean
Dim dbaseEqn As String
Dim dbaseSheet As String

Sub add(inputStr, equation)
Attribute add.VB_ProcData.VB_Invoke_Func = " \n14"
'Adds an equation and its solution into the database only if the
'equation is not already in the database.  Activated by selecting
'"Add ODE" at the Main Menu.
Dim location As String, displayStr As String
Dim soln_r1 As String, soln_r2 As String
Dim integConst As String, condList As String
Dim msg As String, title As String
Dim style As Integer, response As Integer
Dim found As Boolean

  dbaseEqn = equation
  dbaseChg = False

  If doSearch(equation, location) = False Then
  'The equation is not in the database.
    found = False
  Else
    found = True
  End If

  dbaseSheet = ActiveSheet.Name
  Sheets("AddForm").Select
  displayEqn inputStr, "A5"

  'Replace equation with display variables.
  displayStr = equation
  doDisplay displayStr
  Range("A7").Value = displayStr + " = 0"

  If Not found Then
    Range("A10, A11, A14, A17").ClearContents
  Else
  'Equation is already in the database.  Displays the equation
  'and solution.  If there are already multiple solutions,
  'shows the first solution only.  Also provides msg that a
  'solution exists.
    Sheets(dbaseSheet).Select
    Range(location).Offset(0, 4).Select
    soln_r1 = ActiveCell.Value
    soln_r2 = ActiveCell.Offset(1, 0).Value
    integConst = ActiveCell.Offset(0, 2).Value
    ActiveCell.Offset(0, -2).Select
```

```
    condList = ""
    Do While ActiveCell.Value <> Empty
      condList = condList + ", " + ActiveCell.Value
      ActiveCell.Offset(1, 0).Select
    Loop
    condList = Mid(condList, 3)

   'Convert the internal dbase variables into equivalent
   'display variables.
    If soln_r1 <> "" Then
      doDisplay soln_r1
    End If
    If soln_r2 <> "" Then
      doDisplay soln_r2
    End If
    If integConst <> "" Then
      doDisplay integConst
    End If
    If condList <> "" Then
      doDisplay condList
    End If

    Sheets("AddForm").Select
    Range("A10").Value = soln_r1
    Range("A11").Value = soln_r2
    Range("A14").Value = integConst
    Range("A17").Value = condList
    Application.ScreenUpdating = True
    Application.ScreenUpdating = False

    msg = "Please click 'Add New Sol'n' to add another solution for this ODE."
    style = vbOKOnly + vbInformation
    title = "ODEASY Has A Solution For This ODE"
    response = MsgBox(msg, style, title)
    Range("eqnAdded").Value = True
  End If
  Range("A10").Select
  Application.OnKey "{return}", "addEntryFrm"

End Sub


Sub addEntryFrm()
Attribute addEntryFrm.VB_ProcData.VB_Invoke_Func = " \n14"
'Called by procedure add to position the cursor at the entry bar
'for the solution, or integration constant, or conditions.  The
'positioning depends upon the current position of the cursor.

  Select Case ActiveCell.Address
    Case "$A$10"
      Range("A11").Select
    Case "$A$11"
      Range("A14").Select
    Case "$A$14"
      Range("A17").Select
    Case "$A$17"
      Range("A10").Select
    Case Else
      Range("A10").Select
  End Select

End Sub


Function diff_soln(location) As Boolean
'Called by add_current when the input ODE matches a database ODE.
'This function determine whether the entered solution is also the
'same as the solution in database.  Parameter is the location in
'the active sheet of the matching database ODE.  Returns True if
'the solutions are different, False otherwise.
Dim soln_r1 As String, soln_r2 As String
```

```
Dim diffSoln As Boolean, idxCell As Object

  diffSoln = True
  soln_r1 = Sheets("AddForm").Range("A10").Value
  soln_r2 = Sheets("AddForm").Range("A11").Value
  swap_internal soln_r1, soln_r2, "", ""

  Range(location).Select
  Set idxCell = ActiveCell
  Do While True
    ActiveCell.Offset(0, 4).Select
    If (soln_r1 = ActiveCell.Value And _
        soln_r2 = ActiveCell.Offset(1, 0).Value) Or _
       (soln_r1 = ActiveCell.Offset(1, 0).Value And _
        soln_r2 = ActiveCell.Value) Then
      diffSoln = False
      Exit Do
    Else
      idxCell.Select
      ActiveCell.Offset(1, 0).Select
      Set idxCell = ActiveCell
      Do While (idxCell.Value <> "As Above") And _
               (idxCell.Offset(0, -1).Value = Empty)
        idxCell.Offset(1, 0).Select
        Set idxCell = ActiveCell
      Loop
      If idxCell.Offset(0, -1).Value = Empty Then
        Set idxCell = ActiveCell
      Else
        Exit Do
      End If
    End If
  Loop
  diff_soln = diffSoln

End Function


Sub add_current()
Attribute add_current.VB_ProcData.VB_Invoke_Func = " \n14"
'Attached to the Add Current Sol'n button on the Addform sheet.
'Changes the general solution, integration constant, and any
'conditions into the variables used by the database.  Selects
'the correct database sheet and adds the new entry.
Dim soln_r1 As String, soln_r2 As String
Dim integConst As String, cond_list As String
Dim condition As String, book As String
Dim page As String, problem As String
Dim notes As String, targetCell As Object
Dim eqnNum As Integer, lastRow As Integer
Dim msg As String, title As String
Dim style As Integer, response As Integer
Dim location As String, srchEqn As String
Dim msg2 As String, msg3 As String
Dim sameEqn As Boolean

If (Range("A10").Value = Empty) And (Range("A11").Value = Empty) Then
'No solution entry.
  msg = "Please enter a solution for this ODE."
  style = vbOKOnly + vbInformation
  title = "No Solution Entered"
  response = MsgBox(msg, style, title)
  Exit Sub
End If

sameEqn = False
Range("eqnAdded").Value = False
If dbaseSheet = "" Then
  msg = "This equation has already been added to the database."
  style = vbOKOnly + vbInformation
  title = "Current Equation Is In ODEASY"
```

```
  response = MsgBox(msg, style, title)
  Range("eqnAdded").Value = True
  Exit Sub
Else
  Application.ScreenUpdating = False
  If doSearch(dbaseEqn, location) Then
    If Not diff_soln(location) Then
    'Case of trying to add the same solution to an existing ODE.
      msg = "This equation and solution has been added to "
      msg2 = "the database.  If you wish to add another solution "
      msg3 = "for this equation, please click 'Add New Sol'n'."
      style = vbOKOnly + vbInformation
      title = "Current Entry Is In ODEASY"
      response = MsgBox(msg + msg2 + msg3, style, title)
      Sheets("AddForm").Select
      Range("eqnAdded").Value = True
      Exit Sub
    Else
      Sheets("AddForm").Select
      If Range("A17").Value = Empty Then
        msg = "Please enter one or more conditions to distinguish "
        msg2 = "this solution from existing solution(s) for this ODE."
        style = vbOKOnly + vbInformation
        title = "Condition(s) Required"
        response = MsgBox(msg + msg2, style, title)
        Range("A17").Select
        Exit Sub
      Else
        sameEqn = True
      End If
    End If
  End If
  Sheets("AddForm").Select
End If

If Range("A10").Value <> Empty Then      'There is a solution.
  Application.OnKey "{return}"
  With DialogSheets("AddDbx")
    .EditBoxes("Book").Text = ""
    .EditBoxes("Author").Text = ""
    .EditBoxes("Page").Text = ""
    .EditBoxes("ProbNo").Text = ""
    .EditBoxes("Notes").Text = ""
  End With
  If DialogSheets("AddDbx").Show Then    'Addl info is entered.
    With DialogSheets("AddDbx")
      If (.EditBoxes("Book").Text <> Empty) And _
         (.EditBoxes("Author").Text <> Empty) Then
        book = .EditBoxes("Book").Text + ", " + .EditBoxes("Author").Text
      Else
        If (.EditBoxes("Book").Text <> Empty) Then
          book = .EditBoxes("Book").Text
        Else
          If (.EditBoxes("Author").Text <> Empty) Then
            book = "By " + .EditBoxes("Author").Text
          End If
        End If
      End If
      page = .EditBoxes("Page").Text
      problem = .EditBoxes("ProbNo").Text
      notes = .EditBoxes("Notes").Text
    End With
  End If

  soln_r1 = Range("A10").Value
  soln_r2 = Range("A11").Value
  integConst = Range("A14").Value
  cond_list = Range("A17").Value     'May contain multiple conditions
                                     'each separated by a comma.

 'Convert the solution, integration constant, and conditions back
```

```
'to the internal variables used by the database.
 swap_internal soln_r1, soln_r2, integConst, cond_list

 Sheets(dbaseSheet).Select
 If sameEqn Then
 'Insert the right number of lines after the last solution for
 'this ODE and insert this solution.

   Range(location).Offset(0, -1).Select
   eqnNum = ActiveCell.Value + 1
   Do While (ActiveCell.Value <> "-")
   'Look for the next eqn # or end-of-file marker (-).
     If ActiveCell.Value <> eqnNum Then
       ActiveCell.Offset(1, 0).Select
     Else
       Exit Do
     End If
   Loop

   'Have reached either the next entry or the end-of-file.
   ActiveCell.Offset(-1, 0).Select
   Rows(ActiveCell.Row).Select
   Selection.Insert Shift:=xlDown
   Selection.Insert Shift:=xlDown
   ActiveCell.Offset(1, 1).Select
   ActiveCell.Value = "As Above"
   ActiveCell.Offset(0, 4).Value = soln_r1
   ActiveCell.Offset(0, 6).Value = integConst
   ActiveCell.Offset(0, 7).Value = book
   ActiveCell.Offset(0, 8).Value = page
   ActiveCell.Offset(0, 9).Value = problem
   ActiveCell.Offset(0, 10).Value = notes

   'Determine the number of rows to insert.
   lastRow = ActiveCell.Row
   If soln_r2 <> "" Then
     ActiveCell.Offset(1, 4).Value = soln_r2
     lastRow = lastRow + 1
   End If

   If cond_list <> "" Then
     pos = InStr(cond_list, ",")
     If pos > 0 Then
       ActiveCell.Offset(0, 2).Select
       condition = Mid(cond_list, 1, pos - 1)
       cond_list = Mid(cond_list, pos + 1)
       spaceCond condition
       ActiveCell.Value = condition        'First condition.
       ActiveCell.Offset(1, 0).Select
       pos = InStr(cond_list, ",")
       Do While (pos > 0) Or (cond_list <> Empty)
         If pos > 0 Then
           condition = Mid(cond_list, 1, pos - 1)
         Else
           condition = cond_list
           cond_list = ""
         End If
         spaceCond condition
         ActiveCell.Value = condition
         ActiveCell.Offset(1, 0).Select
         Rows(ActiveCell.Row).Select
         Selection.Insert Shift:=xlDown
         ActiveCell.Offset(0, 3).Select
         cond_list = Mid(cond_list, pos + 1)
         pos = InStr(cond_list, ",")
       Loop
     Else
     'Case of only 1 condition.
       spaceCond cond_list
       ActiveCell.Offset(0, 2).Value = cond_list
       If lastRow > ActiveCell.Row Then
```

```
        'Case of 2 solution lines.
          ActiveCell.Offset(2, 0).Select
          Rows(ActiveCell.Row).Select
          Selection.Insert Shift:=xlDown
        ElseIf (lastRow < ActiveCell.Row) Then
          ActiveCell.Offset(1, 0).Select
          Rows(ActiveCell.Row).Select
          Selection.Insert Shift:=xlDown
        End If
      End If
    Else
    'Case of no conditions.
      If lastRow > ActiveCell.Row Then
      'Case of 2 solution lines.
        ActiveCell.Offset(1, 0).Select
      End If
      ActiveCell.Offset(1, 0).Select
      Rows(ActiveCell.Row).Select
      Selection.Insert Shift:=xlDown
    End If
    Sheets("AddForm").Range("eqnAdded").Value = True

  Else
'Look for the end-of-file marker (-) so that the new entry can
'be inserted at the end.
    Columns("A:A").EntireColumn.Select
    With ActiveSheet.Columns("A:A")
      Set targetCell = .Find(What:="-", After:=ActiveCell, _
          LookIn:=xlFormulas, LookAt:=xlPart, _
          SearchOrder:=xlByRows, SearchDirection:=xlNext, _
          MatchCase:=False)
    End With

    If (Not targetCell Is Nothing) Then  'Go to last entry to find eqn #.
      Range(targetCell.Address).Select
      If (targetCell.Address <> "$A$7") Then
        Do While ActiveCell.Offset(-1, 0).Value = Empty
          ActiveCell.Offset(-1, 0).Select
        Loop
        eqnNum = ActiveCell.Offset(-1, 0).Value + 1
        Range(targetCell.Address).Select
      Else
        eqnNum = 1
      End If
      ActiveCell.Value = eqnNum
      ActiveCell.Offset(0, 1).Select
      ActiveCell.Value = dbaseEqn
      ActiveCell.Offset(0, 6).Select
      ActiveCell.Value = integConst
      ActiveCell.Offset(0, 1).Select
      ActiveCell.Value = book
      ActiveCell.Offset(0, 1).Select
      ActiveCell.Value = page
      ActiveCell.Offset(0, 1).Select
      ActiveCell.Value = problem
      ActiveCell.Offset(0, 1).Select
      ActiveCell.Value = notes

     'Check for the number of rows required for the solution
     'and the conditions to determine where the last row is.
     'Add solution first.
      ActiveCell.Offset(0, -6).Select
      lastRow = ActiveCell.Row
      ActiveCell.Value = soln_r1
      If soln_r2 <> "" Then
        ActiveCell.Offset(1, 0).Value = soln_r2
        lastRow = lastRow + 1
      End If

     'Parse conditions list and extract one condition per line.
      ActiveCell.Offset(0, -2).Select
```

```
      pos = InStr(cond_list, ",")
      Do While (pos > 0) Or (cond_list <> Empty)
        If pos > 0 Then
          condition = Mid(cond_list, 1, pos - 1)
        Else
          condition = cond_list    'Case of only one condition or
          cond_list = ""           'of the last condition.
        End If
        spaceCond condition
        ActiveCell.Value = condition
        ActiveCell.Offset(1, 0).Select
        cond_list = Mid(cond_list, pos + 1)
        pos = InStr(cond_list, ",")
      Loop

     'Update the end-of-file marker.
      ActiveCell.Offset(0, -3).Select
      If (lastRow + 1) > ActiveCell.Row Then
        lastRow = lastRow - ActiveCell.Row + 1
        ActiveCell.Offset(lastRow, 0).Select
      End If
      ActiveCell.Offset(1, 0).Select
      ActiveCell.Value = "-"
      Sheets("AddForm").Range("eqnAdded").Value = True
    End If
  End If

  If Sheets("AddForm").Range("eqnAdded").Value Then
    dbaseChg = True
    Sheets("AddForm").Select
    Application.ScreenUpdating = True
    Range("H2").Value = "           SOLUTION ADDED"
    For i = 1 To 2       'Simulate blinking on notification msg.
      Range("H2").Font.ColorIndex = 6
      Application.Wait Now + TimeValue("00:00:01")
      Beep
      Range("H2").Font.ColorIndex = 4
      Application.Wait Now + TimeValue("00:00:01")
    Next i
    Range("H2").Value = "       ODEASY ADD ENTRY"
    Range("H2").Font.ColorIndex = 2
    Range("A10").Select
    Application.ScreenUpdating = False
  End If
End If

End Sub


Sub add_newSoln()
'Activated by clicking the Add New Sol'n button in AddForm.  It
'first checks to see whether a current solution exists and if it
'does, whether it has been added to the database.  If it has not,
'it prompts the user to ensure that it should be discarded.  The
'current solution is erased from AddForm so that a new solution
'may be entered for the input equation.
Dim msg As String, title As String
Dim style As Integer, response As Integer

  If Not Range("eqnAdded").Value Then
    If Range("A10").Value <> Empty Then
      msg = "Do you wish to add the current solution to the database?"
      style = vbYesNo + vbInformation + vbDefaultButton1
      title = "Current Solution"
      response = MsgBox(msg, style, title)

      If response = vbYes Then
     'User has chosen to add the current solution to the database.
        add_current
      End If
    Else
```

```
    'msg to enter a soln for the ode displayed.
      msg = "Please enter a solution for this ODE."
      style = vbOKOnly + vbInformation
      title = "No Solution Entered"
      response = MsgBox(msg, style, title)
    End If
  Else
  'Reset the eqnAdded which is set by add_current when it has
  'added an eqn to the database.
    Range("eqnAdded").Value = False
  End If

  'Discard any current solution.
  Range("A10").ClearContents
  Range("A11").ClearContents
  Range("A14").ClearContents
  Range("A17").ClearContents
  Range("A10").Select     'Position cursor to accept a new solution.
  Application.OnKey "{return}", "addEntryFrm"

End Sub


Sub swap_internal(soln_r1, soln_r2, integConst, cond_list)
'Called by add_current to convert the solution, integration
'constant, and conditions that have been entered by the user back
'to the internal variables used by the database.
Dim dispSym As String, dbaseSym As String
Dim constAt As String, i As Integer
Dim s_r1Chk() As Boolean, s_r2Chk() As Boolean
Dim integChk() As Boolean, condChk() As Boolean

  If soln_r1 <> "" Then
    spaceSigns soln_r1
  End If
  If soln_r2 <> "" Then
    spaceSigns soln_r2
  End If
  If integConst <> "" Then
    spaceSigns integConst
  End If

  ReDim s_r1Chk(Len(soln_r1))
  ReDim s_r2Chk(Len(soln_r2))
  ReDim integChk(Len(integConst))
  ReDim condChk(Len(cond_list))

  If Sheets("defaults").Range("G2").Value <> _
     Sheets("defaults").Range("I2").Value Then
  'Substitute display variable for internal dependent variable.
    dbaseSym = Sheets("defaults").Range("G2").Value
    dispSym = Sheets("defaults").Range("I2").Value

    If soln_r1 <> "" Then
      swapVar soln_r1, dispSym, dbaseSym, s_r1Chk
    End If
    If soln_r2 <> "" Then
      swapVar soln_r2, dispSym, dbaseSym, s_r2Chk
    End If
    If integConst <> "" Then
      swapVar integConst, dispSym, dbaseSym, integChk
    End If
    If cond_list <> "" Then
      swapVar cond_list, dispSym, dbaseSym, condChk
    End If
  End If

  If Sheets("defaults").Range("G3").Value <> _
     Sheets("defaults").Range("I3").Value Then
  'Substitute display variable for internal independent variable.
    dbaseSym = Sheets("defaults").Range("G3").Value
```

```
    dispSym = Sheets("defaults").Range("I3").Value

    If soln_r1 <> "" Then
      swapVar soln_r1, dispSym, dbaseSym, s_r1Chk
    End If
    If soln_r2 <> "" Then
      swapVar soln_r2, dispSym, dbaseSym, s_r2Chk
    End If
    If integConst <> "" Then
      swapVar integConst, dispSym, dbaseSym, integChk
    End If
    If cond_list <> "" Then
      swapVar cond_list, dispSym, dbaseSym, condChk
    End If
  End If

  If (Sheets("defaults").Range("G4").Value <> _
      Sheets("defaults").Range("I4").Value) And      (Sheets("defaults").Range("I4").Value <> Empty)
Then
  'Substitute display variable for internal function of the
  'internal independent variable.
    dbaseSym = Sheets("defaults").Range("G4").Value
    dispSym = Sheets("defaults").Range("I4").Value

    If soln_r1 <> "" Then
      swapVar soln_r1, dispSym, dbaseSym, s_r1Chk
    End If
    If soln_r2 <> "" Then
      swapVar soln_r2, dispSym, dbaseSym, s_r2Chk
    End If
    If integConst <> "" Then
      swapVar integConst, dispSym, dbaseSym, integChk
    End If
    If cond_list <> "" Then
      swapVar cond_list, dispSym, dbaseSym, condChk
    End If
  End If

  If (Sheets("defaults").Range("G5").Value <> _
      Sheets("defaults").Range("I5").Value) And (Sheets("defaults").Range("I5").Value <> Empty) Then
  'Substitute display variable for internal function of the
  'dependent variable.
    dbaseSym = Sheets("defaults").Range("G5").Value
    dispSym = Sheets("defaults").Range("I5").Value

    If soln_r1 <> "" Then
      swapVar soln_r1, dispSym, dbaseSym, s_r1Chk
    End If
    If soln_r2 <> "" Then
      swapVar soln_r2, dispSym, dbaseSym, s_r2Chk
    End If
    If integConst <> "" Then
      swapVar integConst, dispSym, dbaseSym, integChk
    End If
    If cond_list <> "" Then
      swapVar cond_list, dispSym, dbaseSym, condChk
    End If
  End If

  i = 6
  constAt = "I" + CStr(i)
  Do While (Sheets("defaults").Range(constAt).Value <> Empty) And (i < 15)
    dbaseSym = Sheets("defaults").Range(constAt).Offset(0, -2).Value
    dispSym = Sheets("defaults").Range(constAt).Value

    If soln_r1 <> "" Then
      swapVar soln_r1, dispSym, dbaseSym, s_r1Chk
    End If
    If soln_r2 <> "" Then
      swapVar soln_r2, dispSym, dbaseSym, s_r2Chk
    End If
```

```
    If integConst <> "" Then
      swapVar integConst, dispSym, dbaseSym, integChk
    End If
    If cond_list <> "" Then
      swapVar cond_list, dispSym, dbaseSym, condChk
    End If
    i = i + 1
    constAt = "I" + CStr(i)
  Loop

  If soln_r1 <> "" Then
    i = InStr(soln_r1, "=")
    If i > 0 Then
      If Mid(soln_r1, i - 1, 1) <> " " Then
        soln_r1 = Mid(soln_r1, 1, i - 1) + " " + Mid(soln_r1, i)
      End If
      i = InStr(soln_r1, "=")
      If Mid(soln_r1, i + 1, 1) <> " " Then
        soln_r1 = Mid(soln_r1, 1, i) + " " + Mid(soln_r1, i + 1)
      End If
    End If
  End If
  If soln_r2 <> "" Then
    i = InStr(soln_r2, "=")
    If i > 0 Then
      If Mid(soln_r2, i - 1, 1) <> " " Then
        soln_r2 = Mid(soln_r2, 1, i - 1) + " " + Mid(soln_r2, i)
      End If
      i = InStr(soln_r2, "=")
      If Mid(soln_r2, i + 1, 1) <> " " Then
        soln_r2 = Mid(soln_r2, 1, i) + " " + Mid(soln_r2, i + 1)
      End If
    End If
  End If
  If integConst <> "" Then
    i = InStr(integConst, "=")
    If i > 0 Then
      If Mid(integConst, i - 1, 1) <> " " Then
        integConst = Mid(integConst, 1, i - 1) + " " + Mid(integConst, i)
      End If
      i = InStr(integConst, "=")
      If Mid(integConst, i + 1, 1) <> " " Then
        integConst = Mid(integConst, 1, i) + " " + Mid(integConst, i + 1)
      End If
    End If
  End If

End Sub


Sub spaceCond(condition)
'Called by add_current to ensure spacing between variables and
'operators.  The parameter represents one condition.
Dim tempCond

  condition = Trim(condition)
  tempCond = ""
  Do While Len(condition) > 0
    Select Case Left(condition, 1)
      Case "*"
        tempCond = tempCond + " * "
      Case "+"
        tempCond = tempCond + " + "
      Case "-"
        tempCond = tempCond + " - "
      Case "/"
        tempCond = tempCond + "/"
      Case "^"
        tempCond = tempCond + "^"
      Case "<"
        tempCond = tempCond + " <"
```

```
        If InStr(Mid(condition, 2, 3), ">") = 0 Then
          tempCond = tempCond + " "
        End If
      Case ">"
        If Right(tempCond, 1) <> "<" Then
          tempCond = tempCond + " "
        End If
        tempCond = tempCond + "> "
      Case "="
        tempCond = tempCond + " = "
      Case " "
      Case Else
        tempCond = tempCond + Left(condition, 1)
    End Select
      condition = Mid(condition, 2)
      condition = LTrim(condition)
  Loop
  condition = tempCond

End Sub


Sub add_ode()
'Activated by clicking the Add New ODE button on AddForm.
'Brings up the ODE Entry Form to accept a new equation to add.

  Application.OnKey "{return}"
  get_entry (0)

End Sub


Sub goto_main()
'Activated by the Goto Main button in the AddForm and EditForm.
'If there has been a change in the database, the user is queried
'as to whether it should be saved.  Brings the user back to the
'main menu.
Dim msg As String, title As String
Dim style As Integer, response As Integer

  Application.OnKey "{return}"
  If dbaseChg Then
    msg = "Do you wish to save the updated ODEASY database?"
    style = vbYesNo + vbInformation + vbDefaultButton1
    title = "The ODEASY Database Has Been Changed"
    response = MsgBox(msg, style, title)

    If response = vbYes Then
      ActiveWorkbook.Save
    End If
    dbaseChg = False
  End If
  showMain

End Sub


Sub edit()
Attribute edit.VB_ProcData.VB_Invoke_Func = " \n14"

   If (Not targetCell Is Nothing) Then        'Solution has been found.
    solution = targetCell.Offset(0, 4).Value
    integConst = targetCell.Offset(0, 6).Value
    Sheets("Result").Select
    Range("A6").Value = equation
    Range("A10").Value = solution
    Range("A13").Value = integConst
  Else
    Sheets("Add").Select
    Range("A6").Value = equation
    Range("A10").Select
```

```
  End If

End Sub


=============================================================================
Attribute VB_Name = "Search"
---------------------------
Dim dbase As String
Dim idxAddr As String

Function get_order(ByVal term) As Integer
Attribute get_order.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns the highest order in the term which is in y' form.
Dim highOrdr As Integer, curOrdr As Integer
Dim pos As Integer

  highOrdr = 0
  Do
    curOrdr = 0
    pos = InStr(term, "'")
    If pos > 0 Then
      Do While Mid(term, pos, 1) = "'"  'Find the order of
        curOrdr = curOrdr + 1           'the current derivative
        pos = pos + 1                   'in the term.
      Loop
      If curOrdr > highOrdr Then
        highOrdr = curOrdr
      End If
      term = Mid(term, pos)             'Prepare term to look for
                                        'next possible derivative.
    Else
      Exit Do
    End If
  Loop
  get_order = highOrdr

End Function


Function denom_dVar(ByVal term) As Boolean
Attribute denom_dVar.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns True if the dependent variable is in the denominator of the term.
'Called by function first_degree to determine whether the dependent variable
'is part of a denominator of a term so that the equation containing the term
'would not be of the first degree (which would constitute a non-linear eqn).
Dim lBracCnt As Integer, rBracCnt As Integer
Dim pos As Integer, found As Boolean
Dim depVar As String

  found = False
  depVar = Range("G2").Value
  pos = InStr(term, "/")
  If pos > 0 Then
    term = Mid(term, pos + 1)
    Select Case Left(term, 1)
    Case depVar
      found = True
    Case "("
      lBracCnt = 1
      rBracCnt = 0
      Do
        term = Mid(term, 2)
        Select Case Left(term, 1)
          Case "("
            lBracCnt = lBracCnt + 1
          Case ")"
            rBracCnt = rBracCnt + 1
          Case depVar
            found = True
        End Select
```

```
      Loop Until (lBracCnt = rBracCnt) Or (found)
    End Select
  End If
  denom_dVar = found

End Function


Function first_degree(ByVal term) As Boolean
Attribute first_degree.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns True if the dependent variable, y, and its derivatives occur to
'the first degree only in term (one of the 3 conditions for linearity).
'Returns False otherwise.  Called by function get_linearity.
Dim degPos As Integer, varPos As Integer
Dim lBracPos As Integer, lBracCnt As Integer
Dim rBracPos As Integer, rBracCnt As Integer
Dim is_first As Boolean, determined As Boolean
Dim y_inside As Boolean, counter As Integer

  If denom_dVar(term) Then        'The dependent variable is a denominator
    first_degree = False          'of the term --> not of the first degree.
  ElseIf InStr(term, "^") > 0 Then
    is_first = True
    lBracCnt = 0
    rBracCnt = 0
    lBracPos = 0
    rBracPos = 0
    degPos = 0
    varPos = 0
    counter = 1
    y_inside = False
    determined = False
    Do While (Not determined) And (Len(term) > 0)
      Select Case (Left(term, 1))
      Case "y"
        If Mid(term, 2, 1) = "'" Then     'Advance to the end of a
          Do                              'derivative.
            term = Mid(term, 2)
            counter = counter + 1
          Loop Until (Mid(term, 2, 1) <> "'")
        End If
        varPos = counter
        If (lBracPos < varPos) And (lBracCnt <> rBracCnt) Then
          y_inside = True                 'Indicate that y is inside
        End If                            'a set of parenthesis.
      Case "("
        lBracPos = counter
        lBracCnt = lBracCnt + 1
      Case ")"
        rBracPos = counter
        rBracCnt = rBracCnt + 1
        If ((y_inside) And (Mid(term, 2, 1) <> "^")) Then
        'Reset y_inside if brackets now match since the bracketted
        'elements are not taken to a power.
          If (lBracCnt = rBracCnt) Then
            y_inside = False
          End If
        End If
      Case "^"
        degPos = counter
        If (degPos = (varPos + 1)) Or _
           ((degPos = (rBracPos + 1)) And (lBracCnt = rBracCnt) _
             And (y_inside)) Or _
           ((degPos = (rBracPos + 1)) And (varPos > lBracPos) _
             And (lBracCnt <> rBracCnt)) Then
          is_first = False                'Cases of "y^", "y'''^",
          determined = True               '"(...y...)^", or "(...y'''...)^".
        Else
          If (InStr(2, term, "^") = 0) Then 'No further degrees possible.
            determined = True
          End If                          'Else a potential degree of y
```

```
        End If                                  'exists, so continue processing.
      End Select
      term = Mid(term, 2)
      counter = counter + 1
    Loop
    first_degree = is_first
  Else
    first_degree = True                 'no degree occurs in term --> linear.
  End If

End Function


Function trans_y(term) As Boolean
Attribute trans_y.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns True if the dependent variable is encountered within parentheses,
'False otherwise.  This function is used to check whether transcendental
'functions have the dependent variable, y, (or its derivatives) as an
'argument.  Input term is assumed to begin with a left parenthesis.
'Called by function check_trans.
Dim lBracCnt As Integer, rBracCnt As Integer
Dim inBracs As Boolean

  lBracCnt = 1
  rBracCnt = 0
  inBracs = False
  term = Mid(term, 2)
  Do
    Select Case Left(term, 1)
      Case "y"
        inBracs = True
      Case "("
        lBracCnt = lBracCnt + 1
      Case ")"
        rBracCnt = rBracCnt + 1
    End Select
    term = Mid(term, 2)
  Loop Until (lBracCnt = rBracCnt) Or (inBracs)
  trans_y = inBracs

End Function


Function check_trans(ByVal term) As Boolean
Attribute check_trans.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns True if the term contains no transcendental function of
'the dependent variable or its derivatives (one of the 3 conditions
'for linearity), False otherwise.  Called by function get_linearity.
Dim pos As Integer, index As Integer
Dim leftPart As String, rightPart As String
Dim no_y As Boolean

  transArray = Array("exp", "ln^", "ln", "sin^", "sin", "cos^", "cos", _
                     "tan^", "tan", "sec^", "sec", "csc^", "csc", "arcsin^", _
                     "arcsin", "arccos^", "arccos", "arctan^", "arctan")

  no_y = True         'Assume that there is no transc fn of y or y'.
  Do While (Len(term) > 0) And (no_y)
    For index = 0 To 18
      pos = InStr(term, transArray(index))
      If pos > 0 Then
        Exit For
      End If
    Next index
    If pos > 0 Then
      leftPart = Mid(term, 1, pos - 1)
      rightPart = Mid(term, (pos + Len(transArray(index))))
      If InStr(transArray(index), "^") > 0 Then
        stripExp rightPart                  'Move past exponent of transcendental.
      End If
      If (Left(rightPart, 1) = "y") Or _
```

```
        ((Left(rightPart, 1) = "(") And (trans_y(rightPart))) Then
        no_y = False
      End If
    Else
      Exit Do                          'No transcendentals of y --> linear.
    End If
    term = leftPart + rightPart
  Loop
  check_trans = no_y

End Function


Function prod_y(term) As Integer
Attribute prod_y.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns the number of times, up to two, that the dependent variable
'occurs within brackets without being interrupted by either a "+" or "-"
'operator.  This function determines whether there are any y's (or y
'derivatives) or product of y's (or y derivatives) nested within brackets.
'For a product, no more than 2 y's are required and hence, the parsing ends
'when 2 multiplicative y's have been found.  Input term is assumed to begin
'with a left parenthesis.  Called by function check_prod.
Dim last_op As Boolean, r_brac As Boolean
Dim y_count As Integer

  last_op = False
  r_brac = False
  y_count = 0
  Do
    term = Mid(term, 2)
    Select Case Left(term, 1)
      Case "y"
        If Not last_op Then          'Halt count if last character was an
          y_count = y_count + 1      'operator --> this y cannot be a
        Else                         'product with the last y.
          last_op = False
        End If
      Case "+", "-"
        If y_count > 0 Then
          last_op = True
        End If
      Case ")"
        r_brac = True
    End Select
    If Left(term, 1) = "(" Then
      y_count = y_count + prod_y(term)
    End If
  Loop Until (r_brac) Or (y_count = 2) Or (Len(term) = 0)
  prod_y = y_count

End Function


Function check_prod(ByVal term) As Boolean
Attribute check_prod.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns True if there are no products of the dependent variable or
'its derivatives detected in the input term, False otherwise.
'Called by function get_linearity.
Dim prior_y As Boolean, no_prod As Boolean
Dim pos As Integer, y_num As Integer

  pos = InStr(term, "y")
  If pos > 0 Then
    y_num = 0
    no_prod = True
    prior_y = False
    Do While (Len(term) > 0) And (no_prod)
      Select Case Left(term, 1)
      Case "("
        y_num = prod_y(term)
        Select Case y_num
```

```
          Case 1
            If prior_y Then
              no_prod = False
            Else
              prior_y = True
            End If
          Case 2
            no_prod = False
        End Select
      Case "+", "-"
        prior_y = False
      Case "y"
        If prior_y Then
          no_prod = False
        Else
          prior_y = True
        End If
      End Select
      If Left(term, 1) <> "(" Then
        term = Mid(term, 2)
      End If
    Loop
    check_prod = no_prod
  Else
    check_prod = True       'No y found --> linear.
  End If

End Function


Function get_linearity(ByVal equation) As Boolean
Attribute get_linearity.VB_ProcData.VB_Invoke_Func = " \n14"
'Returns True if the input equation is linear, False otherwise.  To
'determine linearity, it checks (1) whether the dependent variable and
'its derivatives occur in the first degree only, (2) whether there are
'any transcendentals of the dependent variable or its derivatives, and
'(3) whether there are any products of the dependent variable and/or
'its derivatives.
Dim linear As Boolean, term As String

  Do While (Len(equation) > 0)
    term = get_term(equation)
    linear = first_degree(term)
    If linear Then
      linear = check_trans(term)
    End If
    If linear Then
      linear = check_prod(term)
    End If
    If Not linear Then      'Once a nonlinear term is found, the entire
      Exit Do               'equation is nonlinear --> searching stops.
    End If
  Loop
  get_linearity = linear

End Function


Function doSearch(equation, addr) As Boolean
Attribute doSearch.VB_ProcData.VB_Invoke_Func = " \n14"
'Determines the order and linearity of the sorted equation and
'searches in the appropriate database.  Returns True for a
'successful search and places the address of the index in
'parameter "addr".  Returns False if the equation is not found
'in the in the database.

  order = get_order(equation)
  Select Case order
    Case 1
      targetSheet = "1st Ordr "
    Case 2
```

```
      targetSheet = "2nd Ordr "
    Case 3
      targetSheet = "3rd Ordr "
    Case 4
      targetSheet = "4th Ordr "
    Case 5
      targetSheet = "5th Ordr "
  End Select

  linear = get_linearity(equation)
  If linear Then
    targetSheet = targetSheet + "Lnr"
  Else
    targetSheet = targetSheet + "NonLnr"
  End If
  addr = findSoln(targetSheet, equation)
  If addr = "" Then
    doSearch = False
  Else
    doSearch = True
  End If

End Function


Function findSoln(targetSheet, equation) As String
Attribute findSoln.VB_ProcData.VB_Invoke_Func = " \n14"
'Attempts to find the solution to the given equation in the
'targetSheet.  Called by function search.  Returns the
'address of the found cell or returns an empty string if
'the equation is not found.
Dim targetCell As Object

  Sheets(targetSheet).Select
  Columns("B:B").EntireColumn.Select

 'Set targetCell = Columns("B").Find(equation, After:=ActiveCell)
  With ActiveSheet.Columns("B:B")
    Set targetCell = .Find(What:=equation, After:=ActiveCell, LookIn:=xlFormulas, LookAt _
        :=xlWhole, SearchOrder:=xlByRows, SearchDirection:=xlNext, MatchCase:= _
        False)
  End With

  If (Not targetCell Is Nothing) Then        'Solution has been found.
    findSoln = targetCell.Address
  Else
    findSoln = ""
  End If

End Function


Sub searchEqn(InputEqn, equation)
'Searches for equation in the appropriate database.  Presents the
'solution if it is found.  The solution is first changed into the
'variables used by the user.  If there are additional solutions,
'the next and previous sol'n buttons are enabled.  Activated by
'selecting "Search ODE" at the Main Menu.
Dim msg As String, title As String
Dim moreSoln As Boolean
Dim response As Integer, style As Integer
Dim soln_r1 As String, soln_r2 As String
Dim eqnNum As String, integConst As String
Dim book As String, page As String
Dim problem As String, notes As String
Dim condList As String, pos As Integer

  If doSearch(equation, idxAddr) = False Then
  'The equation is not in the database.
    Sheets("EntryForm").Select
    Range("A16").Select
```

```
    msg = "The solution for this ODE is not in the database."
    title = "Equation Not Found"
    style = vbOKOnly + vbInformation
    response = MsgBox(msg, style, title)
    Exit Sub
  End If

  dbase = ActiveSheet.Name
  eqnNum = Range(idxAddr).Offset(0, -1).Value
  soln_r1 = Range(idxAddr).Offset(0, 4).Value
  soln_r2 = Range(idxAddr).Offset(1, 4).Value
  integConst = Range(idxAddr).Offset(0, 6).Value
  book = Range(idxAddr).Offset(0, 7).Value
  page = Range(idxAddr).Offset(0, 8).Value
  problem = Range(idxAddr).Offset(0, 9).Value
  notes = Range(idxAddr).Offset(0, 10).Value

'Build up list of conditions.
  Range(idxAddr).Offset(0, 2).Select
  condList = ActiveCell.Value
  Do While ActiveCell.Value <> Empty
    ActiveCell.Offset(1, 0).Select
    If ActiveCell.Value <> "" Then
      condList = ", " + ActiveCell.Value
    End If
  Loop

'Check for the possibility of another solution.  First assume
'that there are multiple solutions.
  moreSoln = True
  Range(idxAddr).Offset(1, 0).Select
  Do While (ActiveCell.Value <> "As Above")
    If (ActiveCell.Offset(0, -1).Value <> Empty) Then
      moreSoln = False
      Exit Do
    End If
    ActiveCell.Offset(1, 0).Select
  Loop

  Sheets("dispForm").Select
  If Not moreSoln Then
    ActiveSheet.DrawingObjects(Array("NxtSoln1", "PrvSoln1", _
                                     "NxtSoln2", "PrvSoln2")).Select
    Selection.Interior.ColorIndex = 19
    Selection.Font.ColorIndex = 15
    ActiveSheet.DrawingObjects(Array("NxtSoln1", "PrvSoln1", _
                "NxtSoln2", "PrvSoln2")).Enabled = False
  Else

    ActiveSheet.DrawingObjects(Array("PrvSoln1", "PrvSoln2")).Select
    Selection.Interior.ColorIndex = 19
    Selection.Font.ColorIndex = 15
    ActiveSheet.DrawingObjects(Array("PrvSoln1", _
                                     "PrvSoln2")).Enabled = False

    ActiveSheet.DrawingObjects(Array("NxtSoln1", "NxtSoln2")).Select
    Selection.Interior.ColorIndex = 8
    Selection.Font.ColorIndex = 1
    ActiveSheet.DrawingObjects(Array("NxtSoln1", _
                                     "NxtSoln2")).Enabled = True
  End If

'Convert the internal dbase variables into equivalent
'display variables.
  If soln_r1 <> "" Then
    doDisplay soln_r1
  End If
  If soln_r2 <> "" Then
    doDisplay soln_r2
  End If
  If integConst <> "" Then
```

```
    doDisplay integConst
  End If
  If condList <> "" Then
    doDisplay condList
  End If

 'Fill in page 1.
  displayEqn InputEqn, "A5"
  Range("H7").Value = eqnNum
  Range("P7").Value = dbase
  Range("A10").Value = soln_r1
  Range("A11").Value = soln_r2
  Range("A14").Value = integConst
  Range("A17").Value = condList

 'Fill in page 2.
  InputEqn = Range("A5").Value
  displayEqn InputEqn, "A34"
  If InStr(book, ",") > 0 Then
    pos = InStr(book, ",")
    Range("C38").Value = Mid(book, 1, pos - 1)
    Range("D42").Value = LTrim(Mid(book, pos + 1))
  Else
    Range("C38").Value = book
  End If
  Range("C40").Value = page
  Range("N40").Value = problem
  Range("D44").Value = notes

  Range("A1").Select
  ActiveCell.Offset(17, 0).Select

End Sub


Sub search_ode()
'Activated by clicking the New Search button on DispForm.
'Brings up the ODE Entry Form to accept a new equation to search.

  Application.OnKey "{return}"
  get_entry (3)

End Sub


Sub turn_page2()
'Activated by either the Next Page button on the DispForm to
'turn to page 2.

  Range("A1").Select
  ActiveCell.Offset(36, 0).Select
  Range("A45").Select

End Sub


Sub turn_page1()
'Activated by either the Previous Page button on the DispForm to
'turn to page 1.

    Range("A1").Select
    ActiveCell.Offset(17, 0).Select

End Sub


Sub nextSoln()
'Activated by clicking the Next Sol'n buttons on page 1 and
'page 2 of DispForm to bring on screen the next solution for the
'current ODE.  These buttons are disabled if the display is at
'the last solution.
```

```
Dim isNext As Boolean

  If Not Sheets("DispForm").DrawingObjects("NxtSoln1").Enabled Then
    Beep
    Exit Sub
  End If

  Application.ScreenUpdating = False
  isNext = True
  Sheets(dbase).Select
  Range(idxAddr).Offset(1, 0).Select

 'Look for the next solution.
  Do While (ActiveCell.Value <> "As Above")
    If ActiveCell.Offset(0, -1).Value <> Empty Then
    'The current solution is the last solution.
      isNext = False
      Exit Do
    End If
    ActiveCell.Offset(1, 0).Select
  Loop

  If Not isNext Then
  'Currently displaying the last soln. Disable the Next Soln buttons.

    With Sheets("DispForm").DrawingObjects(Array("NxtSoln1", _
                                                 "NxtSoln2"))
      .Interior.ColorIndex = 19
      .Font.ColorIndex = 15
      .Enabled = False
    End With
  Else
  'Check if this new solution is the last solution.

    idxAddr = ActiveCell.Address
    ActiveCell.Offset(1, 0).Select
    Do While (ActiveCell.Value = Empty) And _
             (ActiveCell.Offset(0, -1).Value = Empty)
      ActiveCell.Offset(1, 0).Select
    Loop

    If ActiveCell.Offset(0, -1).Value <> Empty Then
    'Disable the Next Soln buttons since this is the last solution.

      With Sheets("DispForm").DrawingObjects(Array("NxtSoln1", _
                                                   "NxtSoln2"))
        .Interior.ColorIndex = 19
        .Font.ColorIndex = 15
        .Enabled = False
      End With
    End If
    Range(idxAddr).Select
  End If

 'Enable the Previous Soln buttons.
  With Sheets("DispForm").DrawingObjects(Array("PrvSoln1", _
                                               "PrvSoln2"))
    .Interior.ColorIndex = 8
    .Font.ColorIndex = 1
    .Enabled = True
  End With

  newSoln

End Sub


Sub prevSoln()
'Activated by clicking the Previous Sol'n buttons on page 1 and
'page 2 of DispForm to bring on screen the previous solution for
'the current ODE.  These buttons are disabled if the display is
```

```
'at the first solution.
Dim isPrev As Boolean

  If Not Sheets("DispForm").DrawingObjects("PrvSoln1").Enabled Then
    Beep
    Exit Sub
  End If

  Application.ScreenUpdating = False
  isPrev = True
  Sheets(dbase).Select
  Range(idxAddr).Offset(-1, 0).Select

 'Look for the previous solution.
  Do While (ActiveCell.Value <> "As Above")
    If ActiveCell.Offset(0, -1).Value <> Empty Then
      isPrev = False
      Exit Do
    End If
    ActiveCell.Offset(-1, 0).Select
  Loop

  If Not isPrev Then
  'Currently displaying the first solution. Disable the
  'Previous Soln buttons.
    With Sheets("DispForm").DrawingObjects(Array("PrvSoln1", _
                                                 "PrvSoln2"))
      .Interior.ColorIndex = 19
      .Font.ColorIndex = 15
      .Enabled = False
    End With
  Else
  'Check if this new solution is the first solution.

    If ActiveCell.Offset(0, -1).Value <> Empty Then
      If (ActiveCell.Offset(0, -1).Value <> _
          Sheets("DispForm").Range("H7").Value) Then
      'Disable the Prev. Soln buttons since this is the first soln.

        With Sheets("DispForm").DrawingObjects(Array("PrvSoln1", _
                                                     "PrvSoln2"))
          .Interior.ColorIndex = 19
          .Font.ColorIndex = 15
          .Enabled = False
        End With
      End If
    End If
  End If

  idxAddr = ActiveCell.Address

 'Enable the Next Soln buttons.
  With Sheets("DispForm").DrawingObjects(Array("NxtSoln1", _
                                               "NxtSoln2"))
    .Interior.ColorIndex = 8
    .Font.ColorIndex = 1
    .Enabled = True
  End With

  newSoln

End Sub


Sub newSoln()
'Called by NextSoln and PrevSoln to display a new solution in
'DispForm.
Dim soln_r1 As String, soln_r2 As String
Dim integConst As String, condList As String
Dim book As String, page As String
Dim problem As String, notes As String
```

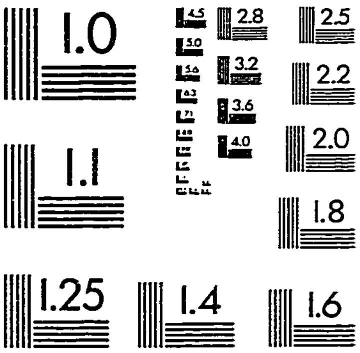
```
  Sheets(dbase).Select
 'Extract data for this solution.
  soln_r1 = Range(idxAddr).Offset(0, 4).Value
  soln_r2 = Range(idxAddr).Offset(1, 4).Value
  integConst = Range(idxAddr).Offset(0, 6).Value
  book = Range(idxAddr).Offset(0, 7).Value
  page = Range(idxAddr).Offset(0, 8).Value
  problem = Range(idxAddr).Offset(0, 9).Value
  notes = Range(idxAddr).Offset(0, 10).Value

 'Build up list of conditions.
  Range(idxAddr).Offset(0, 2).Select
  condList = ActiveCell.Value
  Do While ActiveCell.Value <> Empty
    ActiveCell.Offset(1, 0).Select
    If ActiveCell.Value <> "" Then
      condList = ", " + ActiveCell.Value
    End If
  Loop

 'Convert the internal dbase variables into equivalent
 'display variables.
  If soln_r1 <> "" Then
    doDisplay soln_r1
  End If
  If soln_r2 <> "" Then
    doDisplay soln_r2
  End If
  If integConst <> "" Then
    doDisplay integConst
  End If
  If condList <> "" Then
    doDisplay condList
  End If

  Sheets("DispForm").Select

 'Fill in page 1.
  Range("A10").Value = soln_r1
  Range("A11").Value = soln_r2
  Range("A14").Value = integConst
  Range("A17").Value = condList
  Range("A30").Select

 'Fill in page 2.
  If InStr(book, ",") > 0 Then
    pos = InStr(book, ",")
    Range("C38").Value = Mid(book, 1, pos - 1)
    Range("D42").Value = LTrim(Mid(book, pos + 1))
  Else
    Range("C38").Value = book
    Range("D42").Value = ""
  End If
  Range("C40").Value = page
  Range("N40").Value = problem
  Range("D44").Value = notes

  Range("A1").Select                'Position screen display.
  If ActiveCell.Row < 20 Then
    ActiveCell.Offset(17, 0).Select
  Else
    ActiveCell.Offset(36, 0).Select
    Range("A45").Select
  End If

End Sub


=============================================================================
Attribute VB_Name = "Display"
```

```
-----------------------------

Sub doDisplay(sortedEqn)
'Changes the internal variables in the sorted equation back to
'the display variables.
Dim dispSym As String, dbaseSym As String
Dim constAt As String, dispChk() As Boolean
Dim i As Integer, tempStr As String
Dim lastPos As Integer, pos_y As Integer
Dim pos_t As Integer, lbrac As Integer
Dim rbrac As Integer, curntSheet As String

  ReDim dispChk(Len(sortedEqn))
  If Sheets("defaults").Range("G2").Value <> _
     Sheets("defaults").Range("I2").Value Then
  'Substitute display variable for internal dependent variable.
    dbaseSym = Sheets("defaults").Range("G2").Value
    dispSym = Sheets("defaults").Range("I2").Value
    swapVar sortedEqn, dbaseSym, dispSym, dispChk
  End If

  If Sheets("defaults").Range("G3").Value <> _
     Sheets("defaults").Range("I3").Value Then
  'Substitute display variable for internal independent variable.
    dbaseSym = Sheets("defaults").Range("G3").Value
    dispSym = Sheets("defaults").Range("I3").Value
    swapVar sortedEqn, dbaseSym, dispSym, dispChk
  End If

  If (Sheets("defaults").Range("G4").Value <> _
      Sheets("defaults").Range("I4").Value) And (Sheets("defaults").Range("I4").Value <> Empty) Then
  'Substitute display variable for internal function of the
  'internal independent variable.
    dbaseSym = Sheets("defaults").Range("G4").Value
    dispSym = Sheets("defaults").Range("I4").Value
    swapVar sortedEqn, dbaseSym, dispSym, dispChk
  End If

  If (Sheets("defaults").Range("G5").Value <> _
      Sheets("defaults").Range("I5").Value) And (Sheets("defaults").Range("I5").Value <> Empty) Then
  'Substitute display variable for internal function of the
  'dependent variable.
    dbaseSym = Sheets("defaults").Range("G5").Value
    dispSym = Sheets("defaults").Range("I5").Value
    swapVar sortedEqn, dbaseSym, dispSym, dispChk
  End If

  i = 6
  constAt = "I" + CStr(i)
  Do While (Sheets("defaults").Range(constAt).Value <> Empty) _
           And (i < 15)
    dbaseSym = Sheets("defaults").Range(constAt).Offset(0, -2).Value
    dispSym = Sheets("defaults").Range(constAt).Value
    swapVar sortedEqn, dbaseSym, dispSym, dispChk
    i = i + 1
    constAt = "I" + CStr(i)
  Loop

  'If the swap has changed the alpha ordering of constants at the
  'end of the equation, then sort them here.
  lastPos = 0
  pos_y = InStr(sortedEqn, "y")
  Do While pos_y > 0
    lastPos = pos_y
    pos_y = InStr(pos_y + 1, sortedEqn, "y")

    'Move past any differential symbols.
    Do While Mid(sortedEqn, pos_y + 1, 1) = "'"
      pos_y = pos_y + 1
      lastPos = pos_y
    Loop
```

```
Loop
pos_y = lastPos        'Last position of dependent var.

lastPos = 0
pos_t = InStr(sortedEqn, "t")
Do While pos_t > 0
  lastPos = pos_t
  pos_t = InStr(pos_t + 1, sortedEqn, "t")
Loop
pos_t = lastPos        'Last position of independent var.

If pos_y > pos_t Then
  lastPos = pos_y
Else
  lastPos = pos_t
End If

tempStr = Mid(sortedEqn, 1, lastPos)
sortedEqn = Mid(sortedEqn, lastPos + 1)
If Left(sortedEqn, 1) = " " Then
  tempStr = tempStr + " "
  sortedEqn = Mid(sortedEqn, 2)
End If

Select Case Left(sortedEqn, 1)
  Case "+", "-"
    'No change required to prepare for sorting for display.

  Case "*", "/", "^"
    'Move past multiplicative expression with dependent or
    'independent variable; or move past denominator; or
    'move past exponent.
    If Left(sortedEqn, 1) = "*" Then
      tempStr = tempStr + "* "
      sortedEqn = Mid(sortedEqn, 3)
    Else
      tempStr = tempStr + Left(sortedEqn, 1)
      sortedEqn = Mid(sortedEqn, 2)
    End If

    lbrac = 0
    rbrac = 0
    If Left(sortedEqn, 1) = "(" Then
    'Move past bracketted expression.
      Do
        Select Case Left(sortedEqn, 1)
          Case "("
            lbrac = lbrac + 1
          Case ")"
            rbrac = rbrac + 1
        End Select
        tempStr = tempStr + Left(sortedEqn, 1)
        sortedEqn = Mid(sortedEqn, 2)
      Loop Until (lbrac = rbrac)
    Else
      tempStr = tempStr + Left(sortedEqn, 1)
      sortedEqn = Mid(sortedEqn, 2)
    End If
End Select

If Left(sortedEqn, 1) = " " Then
  tempStr = tempStr + " "
  sortedEqn = Mid(sortedEqn, 2)
End If

If Len(sortedEqn) > 1 Then
  curntSheet = ActiveSheet.Name
  sortedEqn = "(" + sortedEqn + ")"
  sortMult sortedEqn
  sortRest sortedEqn
  Sheets(curntSheet).Select
```

```
    sortedEqn = Mid(sortedEqn, 3)
    sortedEqn = Left(sortedEqn, Len(sortedEqn) - 1)
    If Left(sortedEqn, 1) = "-" Then
      sortedEqn = "- " + Mid(sortedEqn, 2)
    End If
    tempStr = tempStr + sortedEqn
  Else
    tempStr = tempStr + sortedEqn
  End If
  sortedEqn = tempStr

End Sub


Sub swapVar(eqn, fromSymb, toSymb, dispChk)
'Called by procedure doDisplay to change the internal database
'variables (fromSymb) into the equivalent ones (toSymb)entered
'by the user. Also called by swap_internal to change display
'variables (fromSymb) back to the equivalent internal variables
'(toSymb).
Dim pos As Integer

  Select Case toSymb
    Case "@"
      toSymb = Sheets("defaults").Range("I15").Value
    Case "~"
      toSymb = Sheets("defaults").Range("I16").Value
    Case "!"
      toSymb = Sheets("defaults").Range("I17").Value
    Case "#"
      toSymb = Sheets("defaults").Range("I18").Value
  End Select

  pos = InStr(eqn, fromSymb)
  Do While pos > 0
    If Not dispChk(pos) Then
    'This prevents a predefined variable from being
    'repeatedly replaced.
      dispChk(pos) = True
      If Not preDefined(eqn, pos) Then
      'Ensure that it is not part of a predefined
      'function.
        Mid(eqn, pos, 1) = toSymb
      End If
    End If
    pos = InStr(pos + 1, eqn, fromSymb)
  Loop

End Sub
```

IMAGE EVALUATION TEST TARGET (QA-3)



150mm

6″

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989