QUICKPAK SCIENTIFIC

A QuickBASIC Numerical Analysis Toolbox for The IBM® PC and True Compatible Computers

---

Table of Contents

Introduction 1

What is QuickPak Scientific? 3

The files on your distribution diskette 7

Using QuickPak Scientific in your programs . . . 9

QuickPak Scientific Demonstration Programs . . . 11

- Linear Algebra 13
- Differential Equations 14
- Integration 16
- Differentiation 17
- Non-linear Equations 18
- Optimization 20
- Interpolation 22
- Curve Fit 23
- Fast Fourier 24
- Statistics 25
- Functions 26
- Complex Numbers 27
- Trigonometry 28
- Matrices 29
- Vectors 30
- Utility Programs 31
- Applications 32

TABLE OF CONTENTS

QuickPak Scientific Subroutines and Functions . . 33

Linear Algebra

LINEAR1 35
LINEAR2 37
LINEAR3 39
IMPROVE 41

Differential Equations

RK4 43
RKF45 46
RKF56 46
RKF78 46
NYM4 49
ADAMSPC 52
POISSON 54
HEAT 58
WAVE 60

Integration

SIMPSON 63
SPLINE 64
ROMBERG 65
INTEGRA2 67
INTEGRA3 70
ASIMPSON 72

TABLE OF CONTENTS

QuickPak Scientific Subroutines and Functions

Differentiation

DERIV1 73
DERIV2 75
DERIV3 77

Non-linear Equations

QUADRATIC 78
CUBIC 78
QUARTIC 78
POLYROOT 80
REALROOT 81
NEWTON 84
NLINEAR 86
MINIMIZ2 89

Optimization

MINIMA1 91
MINIMA2 93
MINIMIZ1 96
MINIMIZ2 100
NLP 102

TABLE OF CONTENTS

QuickPak Scientific Subroutines and Functions

Interpolation

CSFIT1 106
CSFIT2 108
INTERP1 110
INTERP2 111

Curve Fit

FFIT 113
LSQFIT 114
SURFIT 116

Fast Fourier

FFT1 120
PRTFFT1 122
FFT2 123
PRTFFT2 125

Statistics

CHI 127
FDIST 128
NORMAL 129
TDIST 130

QuickPak Scientific Subroutines and Functions

Functions

GAMMA 131
BESSEL 132
ERF 133
BETA 134

Complex Numbers

CMPXADD 135
CMPXDIV 136
CMPXMULT 136
CMPXRECP 137
CMPXPOWR 137
CMPXROOT 138
CMPXSQRT 138
CMPXSUB 139

Trigonometry

ACOS 140
ACOSH 140
ASIN 141
ASINH 141
ATAN3 142
ATANH 143
COSH 143
SINH 144
TANH 144

TABLE OF CONTENTS

QuickPak Scientific Subroutines and Functions

Matrices

INVERSE 145
DETERMIN 146
EIGEN1 147
EIGEN2 149
IMATRIX 150
MATADD 151
MATSUB 152
MATXMAT 153
MATXVEC 154
RANK 155
TRACE 156
TRANSPOS 157

Vectors

UVECTOR 158
VCROSS 159
VDOT 160
VECADD 161
VECMAG 162
VECSTP 163
VECSUB 164
VECVTP 165

QuickPak Scientific Subroutines and Functions

Utility

CDATE 166
JULIAN 167
XYPLOT 169
SCALE 171
ROUND 172

Bibliography and References 173

Appendices

A. Order Reduction of Differential Equations . . 175

B. Practical Applications

Optimal Impulsive Orbital Transfer . . . 178
A Graphics Display of Three-Body Motion . . 182
Time of Apogee and Perigee of the Moon . . 187

C. An Offer to Subscribe to Celestial Computing

Introduction

Thanks!

Thank you for purchasing QuickPak Scientific from Crescent Software!

We have put every effort into making this the most powerful and useful collection of QuickBASIC numerical analysis subroutines and functions available. We sincerely hope that you find it both useful and informative. If you have a comment, a complaint, or perhaps a suggestion for another product you would like to see, please let us know. We want to be your *favorite* software company.

Registration

Please take a few moments to fill out the enclosed registration card. Doing this entitles you to free technical support by phone, as well as insuring that you are notified of possible upgrades and new products. Many upgrades are offered at little or no cost, but we cannot tell you about them unless we know who you are!

Also, please mark the product serial number on your disk labels. License agreements and registration forms have an irritating way of becoming lost. Writing the serial number on the diskette will keep it handy.

You may also want to note the version number in a convenient location, since it is stored directly on the distribution disk in the volume label. If you ever have occasion to call us for assistance, we will probably need to know the version number you are using. To determine the version number for any Crescent Software product simply display a directory of the original disk. The first thing that appears is similar to:

Volume in drive A is QPSci V3.0

Upgrades

We are constantly improving all of our products, so you may want to call us periodically and ask for the current version number. Major upgrades are always announced, however minor fixes or additions are generally not. If you are having any problems at all–even if you are sure it is not with one of products–please call us. We support all versions of QuickBASIC, and can often provide better assistance than Microsoft.

What is QuickPak Scientific?

QuickPak Scientific is a comprehensive collection of subroutines and functions which provide numerical analysis computing capability for QuickBASIC programmers. Each routine is a flexible and easy to use digital computer algorithm designed to help you solve practical and challenging problems in engineering, science, and other technical applications involving one or more of the following types of numerical calculations:

- Systems of Linear Equations
- Ordinary Differential Equations
- Partial Differential Equations
- Integration
- Differentiation
- Non–linear Equations
- Optimization
- Interpolation
- Curve Fit
- Fast Fourier Transforms
- Statistics
- Complex numbers
- Special Functions
- Trigonometry
- Vectors
- Matrices
- Utility programs

The QuickPak Scientific software package is a comprehensive numerical analysis toolbox which consists of three key components designed to make your technical programming effort interesting and easier:

Structured Source Code

Complete source code is provided for both fundamental and state–of–the–art QuickBASIC subroutines and functions. These digital computer algorithms perform a variety of basic and sophisticated number-crunching tasks which may be too time–consuming or difficult to program and debug yourself. Structured programming practice is followed throughout in order to help you visualize each numerical process.

Complete Demonstration Programs

A considerable amount of time has gone into the effort to supply you with user-friendly and interactive demonstration programs. These programs illustrate the proper procedures for working with each QuickPak Scientific routine. They provide guidelines which you can follow when setting up, initializing, and solving your own numerical applications. Many of these programs also provide an example problem for solution, and each program displays information about the algorithm results. Several programs also provide the user with information about how well the software performed.

Comprehensive User's Manual

The QuickPak Scientific user's manual is a comprehensive document which describes the proper procedures for integrating the QuickPak Scientific algorithms into your QuickBASIC applications. Complete coding instructions are included for user-defined support routines, along with many examples. This documentation is a programmer's guide to the art and science of numerical analysis, and includes many programming hints, tips, and technical advice. An extensive bibliography is also included, along with an explanatory and applications appendix.

Every QuickPak Scientific algorithm includes a complete description of the required input and resultant output. For example, the following is the source code listing for the QuickPak Scientific matrix multiplication subroutine.

```
DEFINT I-N
DEFDBL A-H, O-Z

SUB MATXMAT (A(), B(), C(), L, M, N) STATIC

    ' Matrix multiplication subroutine

    ' [ C ] = [ A ] * [ B ]

    ' Input

    '  L   = number of rows of matrix [ A ]
    '  M   = number of columns of matrix [ A ]
           = number of rows of matrix [ B ]
    '  N   = number of columns of matrix [ B ]
    '  A() = matrix A ( L rows by M columns )
    '  B() = matrix B ( M rows by N columns )

    ' Output

    '  C() = matrix C ( L rows by N columns )

    FOR I = 1 TO L
        FOR J = 1 TO N
            S = 0#
            FOR K = 1 TO M
                S = S + A(I, K) * B(K, J)
            NEXT K
            C(I, J) = S
        NEXT J
    NEXT I

END SUB
```

Many of the QuickPak Scientific demo programs include an example problem for solution. The software will interactively prompt you for the required inputs, and several programs will also recommend typical values for these inputs. For example, a typical user prompt might look like:

Please input the convergence criteria
(a value of 1D–8 is recommended)

Several interactive programs allow the QuickBASIC programmer to assess the effects of such things as step size, number of algorithm iterations, convergence criteria, and other factors on the performance and behavior of a particular numerical method. The following is a typical output from the QuickPak Scientific adaptive Simpson integration program.

Program DEMOINT5

< Adaptive Integration of User-defined Functions >

Lower integration limit	=	*0*
Upper integration limit	=	*1*
Solution accuracy	=	*.00000001*
Integral value	=	*.74682413*
Estimated error	=	*1.0925039D-10*

For this example, the values for the lower and upper integration limits, and the solution accuracy are inputs provided by the user. The numbers shown for the integral value and estimated error were computed and displayed by the software. The estimated error informs the user about well the software estimated the actual integral. From this information he or she may decide to increase or decrease the accuracy.

The Files on Your Distribution Diskette

Your diskette contains more than 200 QuickBASIC source code files for a variety of numerical analysis subroutines and functions. The diskette also contains programs which demonstrate how to use each and every subroutine and function. The filenames for the demo programs all begin with the letters DEMO. For example, the demo program for the first linear equation subroutine is called DEMOLIN1. Each QuickPak Scientific demo program, subroutine, and function is saved on the disk in ASCII format and has a BAS filename extension.

There are also MAK files for each demo program. These short files contain a list of the required driver software and support subroutines and functions for a particular demo program. For example, the following is a list of the files contained in the DEMOMIN1.MAK file which are required in order to run the demonstration program DEMOMIN1:

DEMOMIN1.BAS

MINIMIA1.BAS

USERFUN5.BAS

KEYCHECK.BAS

MAK files are unique to QuickBASIC versions 4.0 and later, and must be used with the *Open Program* command of the *File* pulldown menu of QuickBASIC. Once you have started QuickBASIC, a demo program is brought into the programming environment with the *Open Program* command. This procedure first clears out any other program and then loads the selected file as the main program along with any required support functions and subroutines.

Any demonstration program can be compiled into a stand-alone executable program with the standard QuickBASIC *Compile* command and its many options.

Several QuickPak Scientific algorithms require subroutines which contain user–defined functions and other information. Many of these subroutines are stored on the QuickPak Scientific diskette in files which begin with the name USERFUN. The three demonstration programs

DEMORK4, DEMORKF and DEMONYM4

also require subroutines which define example systems of differential equations. These subroutines are stored in the files DERIVAT1, DERIVAT2, and DERIVAT3.

The QuickPak Scientific double and triple integration algorithms require two support subroutines called USERSUB1 and USERSUB2 which define the user's function and its integration limit functions. These files can also be found on the QuickPak Scientific diskette.

All demo programs also require a short subroutine called KEYCHECK which simply checks the user response to several software prompts.

System Requirements

All QuickPak Scientific code requires version 4.0 or later of the QuickBASIC software package. The amount of system RAM required will depend upon your particular application. Any version of DOS which is compatible with QuickBASIC will also work with the QuickPak Scientific software.

Using QuickPak Scientific in Your Programs

The individual QuickPak Scientific functions and subroutines are designed to be flexible, powerful, and easy to use in your own application programs. In order to use these algorithms effectively and correctly, several programming conventions have been adopted.

Many QuickPak Scientific routines are adaptations of FORTRAN algorithms. All QuickPak Scientific functions and subroutine names use the following default *type declarations*:

```
DEFINT I-N
DEFDBL A-H, O-Z
```

All QuickBASIC variable, constant, and array names also follow this type convention. Please note also that double precision computaions are used in all algorithms.

It is also important to note that all QuickPak Scientific functions and subroutines, except the special functions GAMMA and BESSEL, are *non-recursive*.

The default OPTION BASE 0 of QuickBASIC is active in all QuickPak Scientific algorithms. However, the first index used for most arrays is assumed to be 1.

The user should keep in mind that numerical analysis is both an art and a science. Numerical computing involves such things as convergence criteria, step sizes, iterations, and initial conditions. When using QuickPak Scientific, be creative and experiment! Do not constrain an algorithm. Try different step sizes, convergence criteria, and initial guesses. Begin with conservative numbers for items which control an algorithm, and then tighten these values once the software is successfully solving your problem.

Remember that numerical analysis only provides an approximate answer to your problem.

The QuickPak Scientific toolbox should work with a variety of problems encountered in many technical fields. However, there will always be a few tough problems, such as stiff differential equations and ill–conditioned matrices, which will cause algorithms to fail. For several analysis areas, two or more algorithms have been provided as part of the QuickPak Scientific software package. If one fails, don't be afraid to use the others. You can also help a numerical method by telling it everything you know about your problem. You can gain valuable *qualitative* insight into many problems by performing certain types of *pre-processing*. This includes such things as graphing your functions in order to understand their behavior, and isolating different aspects of your technical problem (divide and conquer).

The user may also want to set up and solve a problem for which he or she knows the answer. This helps to verify that a subroutine or function is working properly and helps to make you feel confident that you are using the software correctly. You are also encouraged to use the demo programs as software drivers for solving your particular problem. Many people learn quickly by following an example. Examine the source code and try to understand how both the driver program and subroutine or function work. Change the source code if you see a more efficient way to solve your problem.

Finally, many people feel comfortable using a certain type of solution method. For this reason several classic algorithms, such as the fourth–order Runge–Kutta and Lagrange's interpolation method, have been included in QuickPak Scientific. However, you are also encouraged to try the other, more efficient algorithms. The state-of-the-art of numerical analysis is improving all the time and our goal is to bring you the best algorithms.

QuickPak Scientific Demonstration Programs

This section is a brief discussion about each of the interactive demonstration programs provided as part of the QuickPak Scientific software package. These demo programs illustrate the proper procedures for setting up, initializing, and interacting with each subroutine and function. They can also be used a simple stand–alone driver programs by adding one or more QuickBASIC subroutines which define and support your specific mathematical problem.

Many of the demonstration programs include an example problem for solution. The software will interactively prompt you for the required inputs necessary to solve a particular problem, and several programs will also recommend typical values for these inputs as part of the display message. The following is a typical QuickPak Scientific user prompt:

Please input the convergence tolerance
(a value of 1D–8 is recommended)

The computer programs also check many user inputs for validity with the DO LOOP construct of QuickBASIC. If an invalid number is input, the software will redisplay the prompt as many times as necessary until a valid response is provided by the user.

Several programs provide information about how well a particular algorithm has performed. This information includes such things as the actual number of algorithm iterations, an error estimate, and the number of function evaluations. From this data, the user can assess the effects of such things as step size, algorithm iterations, and other factors on the performance and behavior of the different numerical methods. This information can also provide valuable insight into your technical problem which may need to be solved again with different algorithm controls.

Informative error messages are also provided by several QuickPak Scientific algorithms. These messages may occur for a variety of reasons. You may encounter such things as singular or ill-conditioned matrices during a solution process, attempts to find the root of a function which is not bracketed, and other types of implementation errors.

Problems may also occur because the user has improperly set up the driver program or incorrectly coded a required QuickBASIC support subroutine. These problems may involve such things as incorrect dimensioning of arrays, invalid initialization, or poor initial guesses. The user should be very careful to account for and avoid any singularities which may occur in a support subroutine or function. This includes such things as division by zero, square roots of negative numbers, and very small or large function values.

Several programs also display recommended actions which the user can take to correct a problem. Other user options may be obvious or intuitive. For example, if an iterative algorithm fails to solve your problem in the allotted number of iterations, simply increase the number of iterations allowed, and restart the program.

Finally, there is always the possibility of errors trapped and displayed by the QuickBASIC programming environment. Many QuickPak Scientific subroutines check for a variety of errors but cannot filter and trap all possible errors.

Linear Algebra

The QuickPak Scientific linear algebra routines will allow the QuickBASIC programmer to solve systems of linear equations using the methods of LU decomposition, Gauss–Jordan elimination, and iterative improvement. An algorithm is also included for solving tridiagonal systems of linear equations using Gaussian elimination with partial pivoting.

DEMOLIN1

Demo program for the subroutine LINEAR1 which solves a system of linear equations using the LU decomposition method.

DEMOLIN2

Demo program for the subroutine LINEAR2 which solves a system of linear equations using the Gauss–Jordan elimination method. This program also provides the matrix inverse.

DEMOLIN3

Demo program for the subroutine LINEAR3 which solves a tridiagonal system of linear equations using the method of Gauss elimination with partial pivoting.

DEMOIMPR

Demo program for the subroutine IMPROVE which solves a system of linear equations using iterative improvement. This routine is useful for refining a linear algebra problem which may be subject to noise or round-off errors.

Differential Equations

The QuickPak Scientific package includes a complete set of algorithms for solving first and second order systems of ordinary differential equations. These methods include the classic fourth–order Runge–Kutta and Nystrom methods, three adaptive Runge–Kutta–Fehlberg algorithms, and a variable order Adams–Bashforth–Moulton predictor–corrector subroutine. Efficient QuickBASIC subroutines are also provided for solving the Poisson, heat and wave partial differential equations.

DEMORK4

Demo program for the subroutine RK4 which solves a system of first order vector differential equations using the classical fourth–order Runge–Kutta method. This is a fixed step size algorithm.

DEMORKF

Demo program for the three QuickBASIC subroutines

RKF45, RKF56 and RKF78

which solve a system of first order vector differential equations using variable step size Runge–Kutta–Fehlberg methods with truncation error control.

DEMONYM4

Demo program for the subroutine NYM4 which solves a system of second order vector differential equations using a Nystrom method of fourth–order. This is a fixed step size algorithm.

Differential Equations

(continued)

DEMOAPC

Demo program for the subroutine ADAMSPC which solves a system of first order vector differential equations using a Runge–Kutta–Fehlberg starting method and a variable order Adams–Bashforth–Moulton predictor–corrector method.

DEMOPDE1

Demo program for the subroutine POISSON which solves the two–dimensional elliptic partial differential Poisson equation using the method of finite–differences.

DEMOPDE2

Demo program for the subroutine HEAT which solves the one–dimensional parabolic partial differential heat equation using the Crank–Nicolson method.

DEMOPDE3

Demo program for the subroutine WAVE which solves the one–dimensional hyperbolic partial differential wave equation using the method of finite–differences.

Integration

Tabular data can be integrated with both a Simpson or cubic spline subroutine. Single definite integrals of analytic user–defined functions can be quickly and accurately evaluated with a Romberg algorithm. A Composite Simpson method is also provided for integrating both double and triple definite integrals of analytic functions defined by the user. An adaptive integration method based on Simpson's method completes this series of algorithms.

DEMOINT1

Demo program for the two subroutines

SIMPSON and SPLINE

which integrate a scalar function of the form y = f(x) which may be tabulated at unequal intervals.

DEMOINT2

Demo program for the QuickBASIC subroutine ROMBERG which numerically integrates a user–defined function using Romberg's method and a multiple application trapezoid rule.

DEMOINT3

Demo program for the subroutine INTEGRA2 which approximates the value of a definite double integral using the Composite Simpson method.

DEMOINT4

Demo program for the subroutine INTEGRA3 which approximates the value of a definite triple integral. This program also uses the Composite Simpson method of numerical integration.

DEMOINT5

Demo program for the subroutine ASIMPSON which integrates a user–defined function using an adaptive Simpson method.

Differentiation

The derivatives of both user–defined analytic and tabulated functions can be calculated with the QuickPak Scientific differentiation subroutines. These three QuickBASIC routines perform numerical differentiation by the classic methods of finite–divided–differences, Lagrange's method, and cubic splines.

DEMODER1

Demo program for the subroutine DERIV1 which numerically estimates the first, second, third, and fourth derivatives of a user–defined analytic function of the form y = f(x) using the centered finite–divided–difference formulas.

DEMODER2

Demo program for the QuickBASIC subroutine DERIV2 which performs Lagrange numerical differentiation of tabulated x and y data of the form y = f(x) input by the user.

DEMODER3

Demo program for the subroutine DERIV3 which performs numerical differentiation of tabulated x and y data of the form y = f(x) input by the user. This method uses a cubic spline algorithm.

Non-linear Equations

QuickPak Scientific includes six flexible QuickBASIC subroutines for solving non–linear equations. Algorithms are provided which solve for the real roots of the general quadratic, cubic and quartic equations. This series of subroutines also includes a digital computer algorithm for computing the real and complex roots of any polynomial up to order 36. Two algorithms which solve single non–linear equations, and two algorithms for solving systems of non–linear equations, both with and without derivatives, are also part of this QuickBASIC toolbox.

DEMOPOLY

Demonstration program for the subroutines

QUADRATC, CUBIC, and QUARTIC

which solve for the real roots of a quadratic, cubic, or quartic equation respectively.

DEMOPNRT

Demo program for the subroutine POLYROOT which solves for the real and complex roots of a polynomial of order less than or equal to 36 using Newton's method.

DEMONLE1

Demo program for the subroutine REALROOT which solves for a real root of an unconstrained user–defined analytic function of the form y = f(x) using Brent's method. This algorithm does not require derivatives.

Non-linear Equations

(continued)

DEMONLE2

Demo program for the subroutine NEWTON which solves for the real root of an unconstrained non–linear analytic equation of the form y = f(x) using a combination Newton/secant method. The user must supply a QuickBASIC subroutine which evaluates the non–linear equation and its first derivative.

DEMONLE3

Demo program for the subroutine NLINEAR which solves for the real roots of an unconstrained system of non–linear equations using Newton's method. The user must provide a QuickBASIC subroutine which evaluates the system of non–linear equations and their first partial derivatives.

DEMONLE4

Demo program for the subroutine MINIMIZ2 which is used as a non–linear regression method to solve for the real roots of an unconstrained system of non–linear equations. This program uses an adaptive derivative subroutine and does not require the user to define the derivatives of the non–linear system.

Optimization

Non–linear optimization is the most powerful numerical method for technical applications. It is also the most difficult to implement. QuickPak Scientific provides five flexible and powerful algorithms for this purpose. These subroutines can be used to solve for a minimum or maximum of scalar functions of one or more variables. A complete computer program is also included for solving the constrained, non–linear optimization problem. The source program for this algorithm contains over 2000 lines of QuickBASIC code.

DEMOMIN1

Demo program for the subroutine MINIMA1 which solves for a minimum or maximum of a scalar function of the form y = f(x). This algorithm does not require function derivatives.

DEMOMIN2

Demo program for the subroutine MINIMA2 which uses Brent's method to calculate a minimum or maximum of a scalar function of the form y = f(x). This algorithm does not require derivatives.

DEMOMNZ1

Demo program for the subroutine MINIMIZ1 which solves for a minimum or maximum of an unconstrained scalar function of several variables using an analytic gradient supplied by the user. The user may choose either the Conjugate Gradient or Quasi–Newton method for solution.

Optimization

(continued)

DEMOMNZ2

Demo program for the subroutine MINIMIZ2 which solves for a minimum or maximum of an unconstrained scalar function of several variables using an adaptive numerical gradient computed by the software. Both the Conjugate Gradient and Quasi–Newton methods are available in this algorithm for solving this type of problem.

DEMONLP

Demo program for the series of QuickBASIC subroutines which solve the constrained non–linear optimization problem. This algorithm attempts to find a minimum or maximum of a scalar function of several variables subject to both equality and inequality constraints. This algorithm uses the Method of Multipliers and Quasi–Newton minimization. It solves the problem defined by:

minimize $y = f(\bar{x})$

subject to $h(\bar{x}) = 0$

$g(\bar{x}) \geq 0$

Interpolation

Important technical information often exists in the form of empirical or experimental data which must be carefully interpreted. To address this need, QuickPak Scientific includes several routines for interpolating tabulated data of the form y = f(x) using both a *natural* and *clamped* cubic spline technique. A QuickBASIC subroutine is also provided which can linearly interpolate both two and three–dimensional tabular data.

DEMOCSF1

Demo program for the subroutine CSFIT1 which performs a *natural* cubic spline interpolation of tabulated data of the form y = f(x) supplied by the user.

DEMOCSF2

Demo program for the subroutine CSFIT2 which performs a *clamped* cubic spline interpolation of tabulated data of the form y = f(x) supplied by the user.

DEMOLNT1

Demo program for the subroutine INTERP1 which performs a linear interpolation of tabulated data of the form y = f(x) input by the user.

DEMOLNT2

Demo program for the subroutine INTERP2 which performs a bilinear interpolation of tabulated data of the form z = f(x,y) input by the user.

Curve Fit

Curve–fitting of experimental data is very important in science, engineering and other fields. QuickPak Scientific provides three flexible subroutines for this purpose. The first algorithm can fit data to simple linear, log and exponential functions. The second algorithm performs a least squares fit to data of the form y = f(x), and the third QuickBASIC subroutine calculates fitting coefficients to a three–dimensional surface of the form z = f(x,y) using a Maclaurin series.

DEMOFFIT

Demo program for the subroutine FFIT which fits user–supplied tabular data of the form y = f(x) to simple functions. This QuickBASIC subroutine can fit data to functions of the form

$$y = a + b\,x, \quad y = a + b\,\log_{10} x, \quad \text{and } y = a\,e^{bx}.$$

DEMOLSQF

Demo program for the subroutine LSQFIT which performs a least squares curve fit of tabulated data of the form y = f(x) input by the user. This subroutine uses the method of orthogonal polynomials.

DEMOSFIT

Demo program for the subroutine SURFIT which calculates fitting coefficients for three–dimensional surfaces of the form z = f(x,y). The fitting function is a two–dimensional Maclaurin series of order less than or equal to 10 specified by the user.

Fast Fourier

QuickPak Scientific includes two algorithms which compute the forward and inverse Fast Fourier transforms of real or complex data. These QuickBASIC subroutines transform both one and two–dimensional data using the Danielson–Lanczos or bit reversal method. These subroutines are QuickBASIC versions of the FORTRAN algorithms described in the very popular book, *Numerical Recipes*.

DEMOFFT1

Demo program for the subroutine FFT1 which performs a one–dimensional Fast Fourier and inverse Fast Fourier transform of real and complex data. A subroutine named PRTFFT1 which performs a formatted print of the data generated by FFT1 is also part of this program.

DEMOFFT2

Demo program for the subroutine FFT2 which performs a two–dimensional Fast Fourier and inverse Fast Fourier transform of real and complex data. A subroutine named PRTFFT2 which performs a formatted print of the data generated by FFT2 is also part of this program.

Statistics

The QuickPak Scientific package includes routines for computing characteristics of the Normal, Chi–squared, F distribution and T distribution statistical functions. These algorithms can determine a point *x* from the probability and degrees of freedom defined by the user, or determine probability for a point *x* input by the user for each type of statistical distribution.

DEMOSTAT

Demo program for the following four statistics subroutines:

- CHI – Chi–squared distribution subroutine
- FDIST – F distribution subroutine
- NORMAL – normal distribution subroutine
- TDIST – T distribution subroutine

Functions

Methods for evaluating four important special functions are also included in QuickPak Scientific. Two of these numerical methods take advantage of QuickBASIC's ability to work with recursive functions and subroutines.

DEMOFUNC

Demo program which illustrates how to compute values of the following special functions:

- GAMMA – Recursive Gamma function
- BESSEL – Recursive Bessel functions of integer order
- ERF – Error function
- BETA – Beta function

Complex Numbers

QuickPak Scientific includes eight routines for performing calculations with complex numbers. These subroutines can be used to add, subtract, multiply and divide two complex numbers, raise a complex number to a power, compute its Nth and square root, and find the reciprocal of a complex number.

DEMOCMPX

Demo program for a series of QuickPak Scientific subroutines which allow the programmer to manipulate complex numbers.

- CMPXADD – complex number addition
- CMPXDIV – complex number division
- CMPXMULT – complex number multiplication
- CMPXPOWR – raising a complex number to a power
- CMPXRECP – reciprocal of a complex number
- CMPXROOT – root of a complex number
- CMPXSQRT – square root of a complex number
- CMPXSUB – complex number subtraction

Trigonometry

The built–in trigonometry functions of QuickBASIC are extended with the QuickPak Scientific trigonometry functions. These flexible functions provide easy to use inverse sine, cosine, tangent and hyperbolic functions for the QuickBASIC programmer.

DEMOTRIG

Demo program for the following QuickBASIC trigonometry functions:

- ACOS – inverse cosine function
- ACOSH – inverse hyperbolic cosine function
- ASIN – inverse sine function
- ASINH – inverse hyperbolic sine function
- ATAN3 – four quadrant inverse tangent function
- ATANH – inverse hyperbolic tangent function
- COSH – hyperbolic cosine function
- SINH – hyperbolic sine function
- TANH – hyperbolic tangent function

Matrices

The QuickPak Scientific matrix subprograms provide the programmer with flexible algorithms for performing a variety of calculations involving matrices. Included are subroutines for computing the inverse and determinant of a square matrix, eigenvalues and eigenvectors, rank of a matrix, and fundamental matrix operations such as addition, subtraction, and multiplication.

DEMOINVR

Demo program for the subroutine INVERSE which solves for the inverse of a square matrix using the LU decomposition method.

DEMODETM

Demo program for the subroutine DETERMIN which uses LU decomposition to calculate the determinant of a square matrix.

DEMOMATX

Demo program which provides the QuickBASIC programmer with the capability to perform the following fundamental matrix operations:

- EIGEN1 – real eigenvalues and eigenvectors
- EIGEN2 – real and complex eigenvalues
- IMATRIX – identity matrix subroutine
- MATADD – matrix addition subroutine
- MATSUB – matrix subtraction subroutine
- MATXMAT – matrix multiplication subroutine
- MATXVEC – matrix/vector multiplication subroutine
- RANK – rank of a matrix subroutine
- TRACE – trace of a square matrix subroutine
- TRANSPOS – matrix transpose subroutine

Vectors

The QuickPak Scientific vector routines allow the QuickBASIC programmer to easily perform numerical calculations involving vectors. These calculations include the dot and cross product of two vectors, fundamental operations such as vector addition, subtraction, and multiplication, and the triple scalar and vector products.

DEMOVECT

Demo program for the following eight vector manipulation subroutines:

- UVECTOR – unit vector subroutine
- VCROSS – vector cross product subroutine
- VDOT – vector dot product subroutine
- VECADD – vector addition subroutine
- VECMAG – vector magnitude subroutine
- VECSTP – vector scalar triple product subroutine
- VECSUB – vector subtraction subroutine
- VECVTP – vector triple product subroutine

Utility Programs

Two useful utility programs and subroutines are also part of the QuickPak Scientific package. These QuickBASIC routines can be used to perform calculations involving calendar dates, and plot data generated by your applications.

DEMODATE

Demo program which illustrates how to use the QuickPak Scientific CDATE and JULIAN subroutines to perform calculations involving dates. These subroutines provide the QuickBASIC programmer with the capability to determine the following types of information involving dates:

- Calculate the Julian date from the calendar date
- Calculate the calendar date from the Julian date
- Calculate the number of days between two dates
- Calculate the day of the week from a calendar date
- Calculate of the day of the year from a calendar date
- Calculate of the calendar date from the day of the year

DEMOPLOT

Demo program for the subroutine XYPLOT which displays a simple graphics plot of data of the form y = f(x) provided by the user. The QuickBASIC programmer can provide the subroutine with labels for the axes and a title for the graph. The subroutine XYPLOT will sort and auto–scale X and Y arrays of data and display a convenient graphics plot of this information. This subroutine supports the monochrome CGA, EGA, VGA, and Hercules graphics mode of the IBM–PC and true compatible computers.

Applications

Three stand-alone computer programs are also provided which illustrate how to apply QuickPak Scientific algorithms and solve more sophisticated and practical problems. These particular problems are from the field of Celestial Mechanics.

The first program uses one of the QuickPak Scientific root-finding algorithms to calculate the best way to transfer a space vehicle from one circular orbit to another. This is an orbital mechanics problem which must be solved for just about every space mission.

The second computer program graphically illustrates how the Runge-Kutta-Fehlberg algorithms automatically adjust the step size when solving differential equations. The problem solved in this example is called the circular-restricted, three-body problem. This classic astronomy problem has challenged lots of famous people for many, many years.

The third and final program demonstrates how to use the QuickPak Scientific bracketing and minimization algorithms to find the time of apogee and perigee of the Moon. Apogee occurs when the Moon is farthest from the Earth, and perigee is when the Moon is closest to the Earth. Naturally, the times of lunar apogee and perigee influence such things as the tides and eclipses.

Each of these applications is described in Appendix B.

QuickPak Scientific Subroutines and Functions

This section describes the proper programming procedure for interacting with each and every QuickPak Scientific subroutine and function. The required input and output for all algorithms is defined along with suggestions and examples to help the QuickBASIC programmer use the software. Additional information about the interaction between many QuickPak Scientific subroutines and their companion demonstration program can be found in this section.

Whenever possible, structured programming practice has been exercised within each algorithm. With the powerful and flexible constructs of QuickBASIC it is always possible to write source code from scratch which does not include a single GOTO statement. This technique makes the source code easier to follow and understand. However, many of these algorithms are adaptations of FORTRAN code, and in several cases the QuickBASIC algorithms required a few GOTO statements in order to avoid duplicating large portions of source code.

All QuickPak Scientific functions and subroutine names use the following default type declarations:

```
DEFINT I-N
DEFDBL A-H, O-Z
```

All QuickBASIC variable, constant, and array names also follow this type convention. Naturally, you can change the type of any variable by appending #, !, and % to the name. In fact, this is done in certain demo programs which need a real index variable in a FOR–NEXT construct, for example.

It is also important to note that all QuickPak Scientific functions and subroutines, except the special functions GAMMA and BESSEL, are non-recursive.

The default OPTION BASE 0 of QuickBASIC is active in all QuickPak Scientific algorithms. However, the first index used for most arrays is assumed to be 1.

Subroutines and functions are organized and grouped by major analysis area. For several areas there may be more than one subroutine which solves the same type of problem.

The QuickPak Scientific algorithm descriptions consist of the subroutine or function name, its purpose, calling syntax, parameter list definition, and comments. The comments section includes important technical, mathematical, and implementation information about each algorithm and its companion demonstration program. Please read these sections carefully.

The required input and resultant output for each subroutine and function is clearly defined within the parameter list description of each algorithm. In some cases a subroutine or function parameter is used as both input and output.

The correct type and dimensions of all QuickPak Scientific arrays are clearly defined. This description includes the number of rows and columns of each and every array. The dimensioning of working or temporary arrays used within a subroutine is handled by the respective algorithm. The memory used by these arrays is recovered with the ERASE command when the subroutine has completed its task.

Linear Algebra

Subroutine LINEAR1

Purpose:

The subroutine LINEAR1 solves a system of linear equations using the method of LU decomposition.

Syntax:

CALL LINEAR1 (N, A(), B(), X(), IER)

Where:

N	= number of equations [input]
A()	= matrix of coefficients [input] (2 dimensional array; N rows by N columns)
B()	= right hand column vector [input] (1 dimensional array; N rows by 1 column)
X()	= solution vector [output] (1 dimensional array; N rows by 1 column)
IER	= error flag [output] (0 = no error, 1 = singular matrix)

Comments:

The system of linear equations which are solved by the subroutine LINEAR1 must be of the form:

$$\left[A \right] \bar{X} = \bar{B}$$

This system must also be exactly determined. This means that the number of equations must be exactly equal to the number of unknowns. This implies that matrix [A] is square.

This can also be expressed as

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1N} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{N1} & A_{N2} & A_{N3} & \cdots & A_{NN} \end{bmatrix} \begin{Bmatrix} X_1 \\ \vdots \\ X_N \end{Bmatrix} = \begin{Bmatrix} B_1 \\ \vdots \\ B_N \end{Bmatrix}$$

and in the form

$$\begin{aligned} A_{11}X_1 + A_{12}X_2 + A_{13}X_3 + \ldots + A_{1N}X_N &= B_1 \\ A_{21}X_1 + A_{22}X_2 + A_{23}X_3 + \ldots + A_{2N}X_N &= B_2 \\ \vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots & \\ A_{N1}X_1 + A_{N2}X_2 + A_{N3}X_3 + \ldots + A_{NN}X_N &= B_N \end{aligned}$$

where the matrix [A] is real and square, and N is the number of equations in the system.

The subroutine LINEAR1 will check the matrix [A] for singularity. If the matrix is singular or cannot be factored, the error flag IER will be set to 1. Otherwise this flag is returned with the value 0.

Please note that the original matrix of coefficients, [A], is modified by the software. The user should make a copy of the original matrix before calling this subroutine.

All QuickPak Scientific linear algebra demo programs will request that the user input the elements of all matrices and column vectors by *rows*.

This algorithm is a QuickBASIC adaptation of the FORTRAN code described in Chapter 2 of Reference 1.

Linear Algebra

Subroutine LINEAR2

Purpose:

The subroutine LINEAR2 solves a system of linear equations using the method of Gauss–Jordan elimination. This subroutine also computes the matrix inverse.

Syntax:

CALL LINEAR2 (N, A(), B(), IER)

Where:

N	= number of equations [input]
A()	= input as the matrix of coefficients and output as the matrix inverse (2 dimensional array; N rows by N columns)
B()	= input as the right right hand column vector and output as the solution vector (1 dimensional array; N rows by 1 column)
IER	= error flag [output] (0 = no error, 1 = singular matrix)

Comments:

The system of linear equations which is to be solved using LINEAR2 must be of the form:

$$\left[A \right] \bar{X} = \bar{B}$$

where [A] is a real, square matrix of coefficients, $\bar{B}$ is the right hand column vector, and $\bar{X}$ is the solution vector.

The subroutine LINEAR2 will check the matrix [A] for singularity. If the matrix is singular, the error flag IER will be set to 1. Otherwise this flag is set to 0.

The demo program requests the user to input the elements of matrix [A] and the column vector $\overline{B}$ by *rows*.

This algorithm is described in Reference 1, Chapter 2.

Please note that the original matrix of coefficients, [A], and the right hand column vector $\overline{B}$ are modified by the software. To preserve the elements of the original matrix and vector, the user should store a copy of each before calling this subroutine.

Matrix [A] returns as the inverse of the original matrix, and vector $\overline{B}$ returns as the solution vector of the system of linear equations.

Linear Algebra

Subroutine LINEAR3

Purpose:

The subroutine LINEAR3 solves a system of tridiagonal linear equations using the method of Gauss elimination with partial pivoting.

Syntax:

CALL LINEAR3 (N, A(), B(), C(), F(), X(), IER)

Where:

N	= number of equations [input]
A()	= vector of subdiagonal coefficients [input] (1 dimensional array; N rows by 1 column)
B()	= vector of diagonal coefficients [input] (1 dimensional array; N rows by 1 column)
C()	= vector of superdiagonal coefficients [input] (1 dimensional array; N rows by 1 column)
F()	= right hand side vector [input] (1 dimensional array; N rows by 1 column)
X()	= solution vector [output] (1 dimensional array; N rows by 1 column)
IER	= error flag [output] (0 = no error, 1 = singular matrix)

Comments:

The system of linear equations which can be solved by the subroutine LINEAR3 must be of tridiagonal form.

A tridiagonal matrix is of the following form:

$$\begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & a_k & b_k & c_k & \\ & & & \ddots & \ddots & \ddots \\ & & & & a_N & b_N \end{bmatrix}$$

The c_i's are called the superdiagonal elements, the b_i's are called the diagonal elements, and the a_i's are the subdiagonal elements of this type of matrix. All the other elements of a tridiagonal matrix are equal to zero.

This system must also be exactly determined. This means that the number of equations is exactly equal to the number of unknowns.

The subroutine LINEAR3 will check the tridiagonal matrix for singularity. If the matrix is singular, the error flag IER will be set to 1. Otherwise this flag is set to 0.

Like the other linear algebra algorithms, the system of linear equations solved by LINEAR3 is of the form:

$$\left[A \right] \bar{X} = \bar{B}$$

where [A] is a real, square matrix of coefficients, $\bar{B}$ is the right hand column vector, and $\bar{X}$ is the solution vector.

The demo program for subroutine LINEAR3 will prompt the user for the tridiagonal elements of the matrix [A] by the actual row and column. It will also request the elements of the vector $\bar{B}$ by rows.

Linear Algebra

Subroutine IMPROVE

Purpose:

The subroutine IMPROVE solves a system of linear equations using a method of iterative improvement.

Syntax:

CALL IMPROVE (NITER, N, A(), B(), X(), IER)

Where:

NITER = number of iterations [input]

N = number of equations [input]

A() = matrix of coefficients [input]
(2 dimensional array; N rows by N columns)

B() = right hand column vector [input]
(1 dimensional array; N rows by 1 column)

X() = input as the initial solution vector and output as the improved solution vector
(1 dimensional array; N rows by 1 column)

IER = error flag [output]
(0 = no error, 1 = singular matrix)

Comments:

The subroutine IMPROVE is useful for solving a system of linear equations when only an estimate for the solution vector is known or when the solution vector is noisy, or may be subject to round–off error.

This linear algebra algorithm is described in Chapter 2 of Reference 1.

The system of linear equations is of the form:

$$\left[A \right] \bar{X} = \bar{B}$$

The algorithm assumes that matrix [A] of this linear system is square. This means that the number of unknowns is exactly equal to the number of equations.

The vector $\bar{X}$ is input as an initial estimate for the solution of the system of linear equations. It returns from the subroutine IMPROVE improved and changed by the actual number of iterations, NITER.

The subroutine IMPROVE requires the subroutines SUB LUD and SUB SOLVE. The main subroutine provides all communication with these two subroutines. The subroutines SUB LUD and SUB SOLVE are part of the IMPROVE file on the QuickPak Scientific diskette.

This algorithm improves an initial estimate for the solution vector $\bar{X}$ by solving the following equation iteratively:

$$\left[A \right] \delta\bar{X} = \left[A \right] \left(\bar{X} + \delta\bar{X} \right) - \bar{B}$$

where:

$\delta\bar{X}$ = error in the solution vector $\bar{X}$

The subroutine IMPROVE will check the matrix [A] for singularity. If the matrix is singular or cannot be factored, the error flag IER will be set to 1. Otherwise this flag is returned with a value of 0.

The QuickPak Scientific demo program requests the user to input the elements of the matrix [A] and the column vector $\bar{B}$ by *rows*.

Please note that matrix [A] is modified by the software.

Differential Equations

Subroutine RK4

Purpose:

The subroutine RK4 solves a system of first order vector differential equations using the classic Runge–Kutta fourth order method.

Syntax:

CALL RK4 (NEQ, TIME, DT, Y())

Where:

NEQ	= number of differential equations [input]
TIME	= simulation time [input]
DT	= integration step size [input]
Y()	= input as the integration vector at initial time = TIME and output as the integration vector at time = TIME + DT (1 dimensional array, NEQ rows by 1 column)

Comments:

This subroutine can be used to solve systems of first order vector differential equations of the form:

$$\frac{d\bar{y}}{dt} = f\ (\ \bar{y},\ t\ )$$

using the classic fourth–order Runge–Kutta method.

This algorithm is useful for solving systems of first order differential equations where the equations of motion may be a function of position and time.

The subroutine RK4 requires a second subroutine called DERIVATIVE which evaluates a system of first order vector differential equations defined by the user. This subroutine must be coded as:

```
SUB DERIVATIVE (T, Y(), Z()) STATIC
```

In the parameter list, T is the current integration time or independent variable, Y() is the vector of dependent variables, and Z() is the vector of differential equations. The main subroutine provides the proper interface to this subroutine which is stored on the QuickPak Scientific diskette in the file DERIVAT1.

Please note that this subroutine evaluates the differential equations for one step size each time it is called. The demo program illustrates a simple driver which keeps track of the current integration time and cycles the integration subroutine for an integration period input by the user.

The demo program DEMORK4 solves a system of three first order vector differential equations defined by:

$$\frac{dy_1}{dt}(y_1, y_2, y_3, t) = y_2 - y_3 + t$$

$$\frac{dy_2}{dt}(y_1, y_2, y_3, t) = 3t^2$$

$$\frac{dy_3}{dt}(y_1, y_2, y_3, t) = y_2 + e^{-t}$$

The user must provide a vector of initial conditions in Y(), and an integration period and step size. The time units used for the integration period and step size must be consistent.

For the initial conditions at t = 0 given by

$$y_1(0) = Y(1) = 1, \quad y_2(0) = Y(2) = 1, \quad y_3(0) = Y(3) = -1$$

this first order system has the exact solution

$$y_1(t) = -0.05t^5 + 0.25t^4 + t + 2 - e^{-t}$$

$$y_2(t) = t^3 + 1$$

$$y_3(t) = 0.25t^2 + t - e^{-t}$$

The QuickPak Scientific demo program computes and displays the error between the computed solution and the exact solution.

For this example, the derivative subroutine is:

```
SUB DERIVATIVE (T, Y(), YDOT()) STATIC
    ' System of differential equations subroutine
    '   Y1' = Y2 - Y3 + T
    '   Y2' = 3T^2
    '   Y3' = Y2 + e^(-T)
    ' Input
    '  T    = integration time
    '  Y() = vector of integration variables ( 3 rows )
    ' Output
    '  YDOT() = vector of differential equations ( 3 rows )
    YDOT(1) = Y(2) - Y(3) + T
    YDOT(2) = 3# * T ^ 2
    YDOT(3) = Y(2) + EXP(-T)
END SUB
```

Differential Equations

Subroutines RKF45, RK56 and RKF78

Purpose:

These subroutines can be used to solve systems of first order vector differential equations using three Runge–Kutta–Fehlberg methods with step size control.

Syntax:

```
CALL RKF45 (NEQ, TOL, TIME, DT, Y())
CALL RKF56 (NEQ, TOL, TIME, DT, Y())
CALL RKF78 (NEQ, TOL, TIME, DT, Y())
```

Where:

NEQ = number of differential equations [input]

TIME = simulation time [input]

TOL = truncation error tolerance [input]

DT = integration step size [input]

Y() = input as the integration vector at initial time = TIME and output as the integration vector at time = TIME + DT
(1 dimensional array; NEQ rows by 1 column)

Comments:

These three subroutines can be used to solve systems of first order vector differential equations of the form:

$$\frac{d\bar{y}}{dt} = f\ (\ \bar{y},\ t\ )$$

using variable step size Runge–Kutta–Fehlberg methods.

These algorithms are useful for solving systems of first order differential equations where the equations of motion may be a function of position and time.

Each integration subroutine requires a second subroutine called DERIVATIVE which evaluates a system of first order vector differential equations defined by the user. This subroutine should be coded as:

```
SUB DERIVATIVE (T, Y(), Z()) STATIC
```

In the parameter list, T is the current integration time or independent variable, Y() is the vector of dependent variables, and Z() is the vector of differential equations. Each subroutine provides the proper interface to this subroutine which is stored on the QuickPak Scientific diskette in the file DERIVAT1.

Please note that each subroutine evaluates the differential equations for one step size each time it is called. The demo program illustrates a simple driver which keeps track of the current integration time and step size, and cycles the integration subroutine for an integration period input by the user.

The demo program DEMORKF solves a system of three first order vector differential equations defined by:

$$\frac{dy_1}{dt}(y_1, y_2, y_3, t) = y_2 - y_3 + t$$

$$\frac{dy_2}{dt}(y_1, y_2, y_3, t) = 3t^2$$

$$\frac{dy_3}{dt}(y_1, y_2, y_3, t) = y_2 + e^{-t}$$

The user must provide a vector of initial conditions in Y(), and an integration period and step size. The time units used for the integration period and step size must be consistent.

For the initial conditions at t = 0 given by

$Y(1) = y_1(0) = 1, \quad Y(2) = y_2(0) = 1, \quad Y(3) = y_3(0) = -1$

this first order system has the exact solution

$$y_1(t) = -0.05t^5 + 0.25t^4 + t + 2 - e^{-t}$$

$$y_2(t) = t^3 + 1$$

$$y_3(t) = 0.25t^2 + t - e^{-t}$$

The QuickPak Scientific demo program for these subroutines computes and displays the error between the computed solution and the exact solution. This allows the user to assess the effects of such things as step size, initial conditions, and error tolerance on the performance of the each of the Runge–Kutta–Fehlberg integration algorithms.

The subroutine RKF45 is a fourth order method with fifth order truncation error control. The subroutine RKF56 is a fifth order method with sixth order truncation error control, and the subroutine RK78 is a seventh order method with eight–order error control.

Please note that these algorithms will use a step size which may change at each integration step, but will never be larger than the initial step size input by you. To force these three subroutines to use a constant step size, pass each subroutine a large value like 1D+99 for the truncation error tolerance.

These algorithms are discussed in Chapter 10 of Reference 4.

Differential Equations

Subroutine NYM4

Purpose:

The subroutine NYM4 solves a system of second–order vector differential equations using a fourth–order Nystrom method.

Syntax:

CALL NYM4 (N, TP, DT, X1(), XDOT1())

Where:

N	= number of differential equations [input]
TP	= simulation time [input]
DT	= integration step size [input]
X1()	= input as X–vector at initial time = TP and output as X–vector at time = TP + DT (1 dimensional array; N rows by 1 column)
XDOT1()	= input as the integration vector at initial the time = TP and output as the integration vector at the time = TP + DT (1 dimensional array; N rows by 1 column)

Comments:

This subroutine solves a system of second–order vector differential equations of the form:

$$\frac{d\bar{x}^2}{dt^2} = f\left(\bar{x}, \frac{d\bar{x}}{dt}, t\right) = f\left(\bar{x}, \bar{v}, t\right)$$

which may be a function of position $\bar{x}$, velocity $\bar{v}$, and time t.

The subroutine NYM4 requires a subroutine called DERIVATIVE which evaluates a system of vector differential equations defined by the user. This subroutine is coded as:

```
SUB DERIVATIVE(T, X(), XDOT(), XDDOT()) STATIC
```

In the parameter list, T is the current integration time, X() is the vector of independent variables, XDOT() is the vector of first–order derivatives (velocity), and XDDOT() is the vector of second–order derivatives (acceleration). The subroutine NYM4 provides the proper interface to this subroutine which is stored on the QuickPak Scientific diskette in the file DERIVAT2.

Please note that the subroutine NYM4 evaluates the differential equations for one step size each time it is called. The demo program illustrates a simple driver which calls the subroutine NYM4 for an integration period specified by the user. This driver simply checks to see if the time remaining is less than the step size input by the user. If this is true, it sets the final step size to this value. The QuickBASIC source code for this is as follows:

```
WHILE (ABS(TF – TI) > .00000001#)
    A = TF – TI
    IF (ABS(A) < DT) THEN DT = A * SGN(A)
    CALL NYM4(NEQ, TI, DT, R(), V())
    TI = TI + DT
WEND
```

The demo program solves the system of three vector differential equations which describe the unperturbed motion of an Earth–orbiting spacecraft. The program DEMONYM4 requests an initial time, integration period, and step size from the user. A step size between 10 and 60 seconds is recommended. The initial time and integration period must also be specified in the units of seconds.

Initial conditions for the position and velocity of the spacecraft are input by the user. The position vector should be input in units of feet and the velocity vector input in feet per second. The demo program will print the results of the numerical integration in the same units.

The system of three second–order vector differential equations solved by the companion demo program are as follows:

$$\frac{d^2x_1}{dt^2} = \frac{-\mu R_x}{R^3}, \qquad \frac{d^2x_2}{dt^2} = \frac{-\mu R_y}{R^3}, \qquad \frac{d^2x_3}{dt^2} = \frac{-\mu R_z}{R^3}$$

where:

R_x, R_y, R_z = position vector of the spacecraft

R = position magnitude = $\sqrt{R_x^2 + R_y^2 + R_z^2}$

μ = gravitational constant of the earth

These three differential equations are also called the equations of motion of the spacecraft.

Typical position (feet) and velocity vectors (feet per second) for the Space Shuttle are as follows:

R_x = –19472500.3 R_y = 6587457.02 R_z = 7367882.5

V_x = –4687.10293 V_y = –23436.308 V_z = 8566.3774

The orbital period is the time required for the Space Shuttle to complete one orbit around the Earth. For this case, the period is 5404.124635 seconds. With these initial conditions, the demo program can be used to assess the effect of different step sizes on how well the orbit closes by integrating the equations for exactly one orbital period.

Differential Equations

Subroutine ADAMSPC

Purpose:

The subroutine ADAMSPC solves systems of first order vector differential equations using an Adams/Bashforth/Moulton predictor–corrector method.

Syntax:

CALL ADAMSPC (NEQ, NORD, TIME, DT, Y())

Where:

NEQ = number of equations in system [input]

NORD = order of integration [input]

TIME = input as the initial simulation time = TIME and output as the time = TIME + DT

DT = integration step size [input]

Y() = input as the integration vector at TIME and output as the integration vector at TIME + DT (1 dimensional array; NEQ rows by 1 column)

Comments:

The subroutine ADAMSPC uses a Runge–Kutta–Fehlberg 5(6) starter, an Adams/Bashforth predictor, and an Adams/Moulton corrector method to solve systems of first order vector differential equations of the form:

$$\frac{d\bar{y}}{dt} = f\ (\ \bar{y},\ t\ )$$

The computer code for this algorithm is described in Chapter 10 of Reference 4.

The integration subroutine ADAMSPC requires a support subroutine called DERIVATIVE which evaluates a system of first order vector differential equations defined by the user. This subroutine is coded as:

```
SUB DERIVATIVE (T, Y(), YDOT()) STATIC
```

In the parameter list, T is the current integration time or independent variable, Y() is the vector of dependent variables, and YDOT() is the vector of differential equations. The subroutine ADAMSPC provides the proper interface to this subroutine which is stored on the QuickPak Scientific diskette in the file DERIVAT3.

The demo program attempts to solve the system of first order differential equations given by:

$$\frac{dy_1}{dt}(y_1, y_2, t) = -4y_1 - 2y_2 + \cos t + 4\sin t$$

$$\frac{dy_2}{dt}(y_1, y_2, t) = 3y_1 + y_2 - 3\sin t$$

For the initial conditions given by

$$Y(1) = y_1(0) = 0, \qquad Y(2) = y_2(0) = -1$$

this first order system has the exact solution

$$y_1(t) = 2e^{-t} - 2e^{-2t} + \sin t$$

$$y_2(t) = -3e^{-t} + 2e^{-2t}$$

The QuickPak Scientific demo program computes and displays the error between the computed solution and the exact solution.

Differential Equations

Subroutine POISSON

Purpose:

The subroutine POISSON solves the two–dimensional elliptic partial differential equation defined by Poisson's equation using the method of finite–differences.

Syntax:

```
CALL POISSON (A, B, C, D, NX, NY, TOL, NITER,
                                  X(), Y(), W())
```

Where:

A	= x initial endpoint [input]
B	= x final endpoint [input]
C	= y initial endpoint [input]
D	= y final endpoint [input]
NX	= number of x grid lines [input]
NY	= number of y grid lines [input]
TOL	= convergence tolerance [input]
NITER	= input as maximum number of iterations allowed output as actual number of iterations required
X()	= vector of x mesh values [output] (1 dimensional array; NX – 1 rows by 1 column)
Y()	= vector of y mesh values [output] (1 dimensional array; NY – 1 rows by 1 column)
W()	= array of solution values [output] (2 dimensional array; NX – 1 rows by NY – 1 columns)

Comments:

The subroutine POISSON attempts to solve the two–dimensional elliptic partial differential equation given by:

$$\frac{\partial^2 u}{\partial x^2}(x, y) + \frac{\partial^2 u}{\partial y^2}(x, y) = f(x, y) \qquad \begin{matrix} a \le x \le b \\ c \le y \le d \end{matrix}$$

subject to the boundary conditions

$$u(x, y) = g(x, y) \qquad \text{if } x = a \text{ or } x = b \text{ and } c \le y \le d$$

$$u(x, y) = g(x, y) \qquad \text{if } y = c \text{ or } y = d \text{ and } a \le x \le b$$

This subroutine requires two support subroutines called FFUN and GFUN which define the *forcing* function f(x, y) and boundary conditions function g(x, y), respectively. These QuickBASIC subroutines must be coded as follows:

```
SUB FFUN(X, Y, FVAL) STATIC
SUN GFUN(X, Y, GVAL) STATIC
```

The first subroutine returns the function value and the second returns the boundary condition for any X and Y value. The main subroutine provides all communications with these support subroutines.

The companion demo program attempts to solve the following form of Poisson's equation:

$$\frac{\partial^2 u}{\partial x^2}(x, y) + \frac{\partial^2 u}{\partial y^2}(x, y) = xe^y \qquad \begin{matrix} 0 < x < 2 \\ 0 < y < 1 \end{matrix}$$

with boundary conditions given by

$$u(0, y) = 0 \qquad u(2, y) = 2e^y \qquad 0 \le y \le 1$$

and

$$u(x, 0) = x \qquad u(x, 1) = ex \qquad 0 \le x \le 2.$$

For this particular example, the source code for the forcing and boundary conditions functions is:

```
SUB FFUN (X, Y, FVAL) STATIC
    ' "Forcing" function subroutine
    ' z = f(x,y)
    ' Input
    '   X = x value in forcing function
    '   Y = y value in forcing function
    ' Output
    '   FVAL = forcing function value at X, Y
    FVAL = X * EXP(Y)
END SUB

SUB GFUN (X, Y, GVAL) STATIC
    ' Boundary conditions subroutine
    ' z = f(x,y)
    ' Input
    '   X = x boundary value
    '   Y = y boundary value
    ' Output
    '   GVAL = boundary condition at X, Y
    IF (X = 0#) THEN
       GVAL = 0#
    ELSEIF (X = 2#) THEN
       GVAL = 2# * EXP(Y)
    ELSEIF (Y = 0#) THEN
       GVAL = X
    ELSE
       GVAL = EXP(1#) * X
    END IF
END SUB
```

The demo program will prompt the user for the number of X and Y grid lines, initial and final X and Y endpoints, a convergence tolerance, and the maximum number of iterations permitted. A tolerance of 1D–10 should be adequate and 200 is a good value for the number of iterations.

The demo program will display the solution to Poisson's equation as a two–dimensional array. The following is a typical display generated by this software. For this example, five X and four Y grid lines were used. The X initial and final endpoints were 0 and 2, respectively, and the Y initial and final endpoints were 0 and 1, respectively.

	J = 1	2	3	4
I = 1	4.072646D-01	4.974832D-01	6.075961D-01	7.420071D-01
I = 2	8.145237D-01	9.949576D-01	1.215183D+00	1.484009D+00
I = 3	1.221766D+00	1.492405D+00	1.822743D+00	2.225993D+00
I = 4	1.628964D+00	1.989778D+00	2.430227D+00	2.967928D+00
I = 5	2.036042D+00	2.486958D+00	3.037506D+00	3.709724D+00

The corresponding X and Y spatial coordinates for any array location are X = I * DX and Y = J * DY where

$$DX = \frac{X_f - X_i}{NX} \quad \text{and} \quad DY = \frac{Y_f - Y_i}{NY}$$

and NX and NY are the number of user–specified X and Y grid lines, respectively.

The convergence tolerance for this problem was 1D–10. The demo program will also display the actual number of iterations required for the solution, which in this case was 52.

The finite–difference method implemented in subroutine POISSON is described in Chapter 12 of Reference 2.

Differential Equations

Subroutine HEAT

Purpose:

The subroutine HEAT solves the one–dimensional parabolic partial differential heat equation using the Crank–Nicolson solution method.

Syntax:

```
CALL HEAT (ALPHA, NX, NT, XL, T, S())
```

Where:

ALPHA	= equation constant [input]
NX	= number of spatial increments [input]
NT	= number of time increments [input]
XL	= final spatial value [input]
T	= final time value [input]
S()	= solution array [output] (2 dimensional array; NX rows by NT columns)

Comments:

The subroutine HEAT attempts to solve the one–dimensional parabolic partial differential equation given by:

$$\frac{\partial u}{\partial t}(x, t) - \alpha^2 \frac{\partial^2 u}{\partial x^2}(x, t) = 0 \qquad \begin{matrix} 0 < x < L \\ 0 \le t < T \end{matrix}$$

subject to the boundary conditions

$$u(0, t) = u(L, t) = 0 \quad \text{for} \quad 0 < t < T$$

and the initial conditions

$$u(x, 0) = f(x) \quad \text{for} \quad 0 \leq x \leq L$$

This subroutine requires a support subroutine called FFUN which defines the *forcing* function f(x). This QuickBASIC subroutine must be coded as follows:

```
SUB FFUN(X, FVAL) STATIC
```

The subroutine returns the forcing function value for any spatial X value. The main subroutine provides all required communications with this support subroutine.

The companion demo program solves the following form of the heat equation:

$$\frac{\partial u}{\partial t}(x, t) - \frac{\partial^2 u}{\partial x^2}(x, t) = 0 \qquad \begin{array}{l} 0 < x < 1 \\ t > 0 \end{array}$$

subject to the conditions

$$u(0, t) = u(1, t) = 0 \quad \text{for} \quad t > 0$$

and

$$u(x, 0) = \sin(\pi x) \quad \text{for} \quad 0 \leq x \leq 1.$$

Note that the equation constant α for this case is equal to 1 and the forcing function is $f(x) = \sin(\pi x)$.

The demo program will prompt the user for this constant, the number of spatial and time increments to use, and the spatial endpoint and maximum time.

The companion demo program will display the solution as a two–dimensional array. The format of this array is discussed within the software description for subroutine POISSON. For this example, the I index corresponds to spatial coordinates and the J index is the time coordinate.

Differential Equations

Subroutine WAVE

Purpose:

The subroutine WAVE solves the one–dimensional hyperbolic partial differential wave equation using a finite–difference solution method.

Syntax:

CALL WAVE (ALPHA, NX, NT, XL, T, S())

Where:

ALPHA	= equation constant [input]
NX	= number of spatial increments [input]
NT	= number of time increments [input]
XL	= final spatial value [input]
T	= final time value [input]
W()	= solution array [output] (2 dimensional array; NX rows by NT columns)

Comments:

The subroutine WAVE attempts to solve the one–dimensional *hyperbolic* partial differential equation given by:

$$\frac{\partial^2 u}{\partial t^2}(x, t) - \alpha^2 \frac{\partial^2 u}{\partial x^2}(x, y) = 0 \qquad \begin{matrix} 0 < x < L \\ 0 < t < T \end{matrix}$$

subject to the boundary conditions

$$u(0, t) = u(L, t) = 0 \quad \text{for} \quad 0 < t < T$$

and the initial conditions

$$u(x, 0) = f(x) \qquad \text{for} \quad 0 \le x \le L$$

$$\frac{\partial u}{\partial t}(x, 0) = g(x) \qquad \text{for} \quad 0 \le x \le L$$

This subroutine requires two support subroutines called FFUN and GFUN which define the initial condition functions f(x) and g(x), respectively. These QuickBASIC subroutines must be coded as follows:

```
SUB FFUN(X, FVAL) STATIC

SUN GFUN(X, FVAL) STATIC
```

Each subroutine returns the initial condition function value any X value. The main subroutine provides all communications with these support subroutines.

The companion demo program for this subroutine solves the following form of the wave equation:

$$\frac{\partial^2 u}{\partial t^2}(x, t) - 4\frac{\partial^2 u}{\partial x^2}(x, y) = 0 \qquad \begin{matrix} 0 < x < 1 \\ 0 < t < T \end{matrix}$$

with the boundary conditions

$$u(0, t) = u(1, t) = 0 \qquad \text{for} \quad t > 0$$

and the initial conditions

$$u(x, 0) = \sin(\pi x) \qquad \text{for} \quad 0 \le x \le 1$$

$$\frac{\partial u}{\partial t}(x, 0) = 0 \qquad \text{for} \quad 0 \le x \le 1$$

Note that the equation constant α for this case is equal to 2 and the two initial condition functions are $f(x) = \sin(\pi x)$ and $g(x) = 0$. For this example, they are coded as follows:

```
SUB FFUN (X, FVAL) STATIC
    ' Initial conditions function subroutine
    ' y = f(x)
    ' Input
    '   X = x value
    ' Output
    '   FVAL = forcing function value at X, Y
    PI = 3.141592653589793#
    FVAL = SIN(PI * X)
END SUB

SUB GFUN (X, FVAL) STATIC
    ' Initial conditions derivative subroutine
    ' y = du/dt(x,0)
    ' Input
    '   X = x value in derivative function
    ' Output
    '   FVAL = derivative value at X
    FVAL = 0#
END SUB
```

The demo program will prompt the user for the equation constant, the number of spatial and time increments to use, the spatial endpoint and maximum time.

The companion demo program will display the solution as a two–dimensional array. The format of this array is discussed within the software description for subroutine POISSON. For this example, the I index corresponds to spatial coordinates and the J index is the time coordinate.

The finite–difference algorithm implemented in subroutine WAVE is described in Chapter 12 of Reference 2.

Integration

Subroutine SIMPSON

Purpose:

The subroutine SIMPSON numerically integrates a table of X and Y data of the form y = f(x) input by the user.

Syntax:

CALL SIMPSON (N, X(), Y(), SUM)

Where:

N	= number of X and Y data points [input]
X()	= array of X data points [input] (1 dimensional array; N rows by 1 column)
Y()	= array of Y data points [input] (1 dimensional array; N rows by 1 column)
SUM	= integral from X(1) to X(N) [output]

Comments:

In this algorithm the Y() array is the dependent data and the X() array is the independent data;

$$Y_i = f(X), \quad i = 1, 2, \ldots, N.$$

The number of X and Y data points must be odd and should be input by the user in ascending order in the array X(). The demo program checks for an odd number of data points but does not check for ascending order.

The data may be tabulated at unequal X intervals.

Reference 10 describes this algorithm.

Integration

Subroutine SPLINE

Purpose:

The subroutine SPLINE numerically integrates a table of X and Y data of the form y = f(x) input by the user using the method of cubic splines.

Syntax:

CALL SPLINE (N, X(), Y(), SUM)

Where:

N	= number of X and Y data points [input]
X()	= array of X data points [input] (1 dimensional array; N rows by 1 column)
Y()	= array of Y data points [input] (1 dimensional array; N rows by 1 column)
SUM	= integral from X(1) to X(N) [output]

Comments:

In this algorithm the Y() array is the dependent data and the X() array is the independent data;

$$Y_i = f(X), \quad i = 1, 2, \ldots, N.$$

A minimum of two X and Y data points must be provided and should be input by the user in ascending order in the array X(). The demo program checks for a minimum number of data points but does not check for ascending order.

The data may be tabulated at unequal X intervals.

Integration

Subroutine ROMBERG

Purpose:

The subroutine ROMBERG numerically integrates a user–defined analytic function of the form y = f(x) using Romberg's method.

Syntax:

CALL ROMBERG (A, B, MAXITER, EPS, NITER, S)

Where:

A	= lower integration limit [input]
B	= upper integration limit [input]
MAXITER	= maximum number of iterations [input]
EPS	= convergence criteria [input]
NITER	= actual number of iterations [output]
S	= integral from A to B [output]

Comments:

This subroutine numerically integrates a user–defined function as follows:

$$S(x) = \int_A^B f(x)\, dx$$

over the limits A and B specified by the user.

This algorithm uses combination of Romberg's method and multiple–application trapezoidal rule. Additional information can be found in Chapter 16 of Reference 3.

The function must be defined by the user in a subroutine called USERFUNC which is coded as:

```
SUB USERFUNC(X, FVAL) STATIC
```

In the parameter list, X is the function argument and FVAL is the function value at X.

The subroutine ROMBERG also requires a subroutine called TRAPEZOID which is a QuickBASIC implementation of a multiple–application trapezoidal rule. All communication with this subroutine is provided by the subroutine ROMBERG. The subroutine TRAPEZOID is included as part of the file ROMBERG on the QuickPak Scientific diskette.

Values between five and ten are recommended for the maximum number of iterations, and a convergence criteria of 1D–8 is reasonable for most functions.

The demo program numerically integrates the function

$$y = f(x) = e^{-x^2}$$

over a lower and upper integration limit specified by the user. A subroutine defining this function is stored on the QuickPak Scientific diskette in the file USERFUN1 which is coded as follows:

```
SUB USERFUNC (X, FVAL) STATIC
    ' User function subroutine
    ' f(x) = e^(–x^2)
    ' Input
    '  X = function argument
    ' Output
    '  FVAL = function value = f(X)
    FVAL = EXP(–X * X)
END SUB
```

Integration

Subroutine INTEGRA2

Purpose:

The subroutine INTEGRA2 can be used to numerically estimate the value of a definite double integral of a user–defined function of the form z = f(x, y) using a Composite Simpson solution method.

Syntax:

CALL INTEGRA2 (XA, XB, M, N, XJ)

Where:

XA	= lower X end point [input]
XB	= upper X end point [input]
M	= number of Y subdivisions [input]
N	= number of X subdivisions [input]
XJ	= value of double integral [output]

Comments:

This subroutine uses a Composite Simpson method to numerically estimate the value of a definite double integral of the form:

$$S(x,y) = \int_a^b \int_{c(x)}^{d(x)} f(x, y)\, dy\, dx$$

over the x and y limits shown. This integration method is described in Reference 2, Chapter 4.

The subroutine INTEGRA2 requires three other subroutines which define both the integration limit functions and the user–defined integration function.

The lower integration limit function is coded as

SUB FCX (X, F) STATIC

and defines the y = c(x) limit function. The upper integration limit function is coded as

SUB FDX (X, F) STATIC

and defines the y = d(x) limit function. Both of these subroutines return a function value F for any input argument X.

The user–defined integration function is coded as

SUB FXY (X, Y, F) STATIC

and returns the function value F for any combination of the two arguments X and Y.

These three subroutines are on the QuickPak Scientific disk in the file USERSUB1.

The demo program integrates the following definite double integral

$$S(x,y) = \int_{0.1}^{0.5} \int_{x^3}^{x^2} e^{y/x}\, dy\, dx$$

over lower and upper x integration limits specified by the user. The user may also specify the number of x and y subdivisions. Note that the integral will be more accurate as the number of subdivisions increases.

The user should be careful when coding the required support subroutines to make sure any function singularities are handled properly. It may be necessary to divide an integration function into two or more functions and integrate each individually.

The following is the QuickBASIC source code for the two limit functions and the integration function contained in USERSUB1.

Lower integration limit function

```
SUB FCX (X, F) STATIC

    ' y = fc(x) subroutine

    F = X ^ 3

END SUB
```

Upper integration limit function

```
SUB FDX (X, F) STATIC

    ' y = fd(x) subroutine

    F = X ^ 2

END SUB
```

Integration function

```
SUB FXY (X, Y, F) STATIC

    ' f(x,y) subroutine

    F = EXP(Y / X)

END SUB
```

Integration

Subroutine INTEGRA3

Purpose:

The subroutine INTEGRA3 can be used to numerically estimate the value of a definite triple integral of a user–defined function of the form w = f(x, y, z) using a Composite Simpson integration method.

Syntax:

CALL INTEGRA3 (XA, XB, L, M, N, XJ)

Where:

XA	= lower X end point [input]
XB	= upper X end point [input]
L	= number of Z subdivisions [input]
M	= number of Y subdivisions [input]
N	= number of X subdivisions [input]
XJ	= value of triple integral [output]

Comments:

This subroutine uses a Composite Simpson method (Chapter 4, Reference 2) to numerically estimate the value of a definite triple integral of the form:

$$S(x,y,z) = \int_{a}^{b} \int_{c(x)}^{d(x)} \int_{a(x,y)}^{b(x,y)} f(x, y, z)\, dz\, dy\, dx$$

over the x, y and z limits shown.

The subroutine INTEGRA3 requires five other subroutines which define both the integration limit functions and the user–defined integration function.

The lower and upper y integration limit functions are coded as

```
SUB FCX (X, F) STATIC    ⇒ y = c(x)
SUB FDX (X, F) STATIC    ⇒ y = d(x)
```

These subroutines return a function value F for any input argument X.

The lower and upper z integration limit functions are coded as

```
SUB FAXY (X, Y, F) STATIC    ⇒ z = fa(x, y)
SUB FBXY (X, Y, F) STATIC    ⇒ z = fb(x, y)
```

These subroutines return a function value F for any input arguments X and Y.

The user–defined integration function is coded as

```
SUB FXYZ (X, Y, Z, F) STATIC
```

and returns a function value F for any three arguments X, Y and Z.

These three subroutines are on the QuickPak Scientific disk in the file USERSUB2.

The demo program integrates the following definite triple integral:

$$S(x,y,z) = \int_{-2}^{2} \int_{-\sqrt{4-x^2}}^{\sqrt{4-x^2}} \int_{\sqrt{x^2+y^2}}^{2} z\sqrt{x^2+y^2}\, dz\, dy\, dx$$

Integration

Subroutine ASIMPSON

Purpose:

The subroutine ASIMPSON numerically integrates a user–defined analytic function of the form y = f(x) using an adaptive Simpson solution method.

Syntax:

CALL ASIMPSON (A, B, ACC, SUM, ESTERR, IFLAG)

Where:

A	= lower integration limit [input]
B	= upper integration limit [input]
ACC	= solution accuracy [input]
SUM	= integral form A to B [output]
ESTERR	= relative error [output]
IFLAG	= error flag [output]

1 = no error
2 = more than 30 levels
3 = subinterval too small
4 = more than 2000 function evaluations

Comments:

The demo program numerically integrates the function

$$y = f(x) = e^{-x^2}$$

over a lower and upper integration limit specified by the user. A subroutine defining this function is stored on the QuickPak Scientific diskette in the file USERFUN1.

Differentiation

Subroutine DERIV1

Purpose:

The subroutine DERIV1 numerically estimates the first, second, third and fourth derivatives of an analytic function of the form y = f(x) defined by the user.

Syntax:

CALL DERIV1 (X, H, FP1, FP2, FP3, FP4)

Where:

X	= X value of interest [input]
H	= step size [input]
FP1	= first derivative [output]
FP2	= second derivative [output]
FP3	= third derivative [output]
FP4	= fourth derivative [output]

Comments:

This subroutine numerically estimates the first four derivatives of an analytic function of the form

$$y = f(x)$$

defined by the user.

The first, second, third, and fourth derivatives are estimated for the user–defined function using the method of central divided differences.

The subroutine DERIV1 requires a subroutine which evaluates a function defined by the user. This function is specified in a subroutine called USERFUNC which is coded as:

```
SUB USERFUNC (X, FVAL) STATIC
```

In the parameter list, X is the function argument and FVAL is the function evaluated at X.

A step size between .01 and .0001 is recommended. The "best" value for the step size will depend on the behavior of the function in the vicinity of the point of interest. Smaller step sizes should be used whenever the slope of the function is changing dramatically.

The demo program estimates the derivatives of

$$y = f(x) = \sqrt{\sin(x) + 2.5\ e^{x}}$$

for a value of x input by the user.

The subroutine for this function is stored on the QuickPak Scientific diskette in the file USERFUN2 and is coded as:

```
SUB USERFUNC (X, FX) STATIC
    ' User-defined function subroutine
    ' f(x) = sqrt(sin(x) + 2.5 exp(x))
    ' Input
    '   X = function argument
    ' Output
    '   FX = function value at X
    FX = SQR(SIN(X) + 2.5# * EXP(X))
END SUB
```

This method of numerical differentiation is described in Chapter 17 of Reference 3.

Differentiation

Subroutine DERIV2

Purpose:

The subroutine DERIV2 estimates the first derivative of user–supplied tabulated data of the form y = f(x) using Lagrange's method.

Syntax:

CALL DERIV2 (NPTS, NDEG, X(), Y(), XVAL, YVAL)

Where:

NPTS = number of X and Y data points [input]

NDEG = degree of interpolation [input]

X() = array of X data points [input]
(1 dimensional array; NPTS rows by 1 column)

Y() = array of Y data points [input]
(1 dimensional array; NPTS rows by 1 column)

XVAL = X argument [input]

YVAL = derivative value [output]

Comments:

In this algorithm the Y() array is the dependent data and the X() array is the independent data;

$$Y_i = f(X), \quad i = 1, 2, \ldots, NPTS.$$

A minimum of *four* X and Y data points must be provided and should be input by the user in ascending order in the array X(). The demo program checks for a minimum of four data points but does not check for ascending order.

The degree of interpolation will be a number between 1 and 9 and depends on the total number of data points.

The highest degree of interpolation is equal to the smaller of (NPTS – 2) or 9. The demo program will compute the highest degree of interpolation possible and include this number as part of the user prompt.

The valid range for an X value for evaluation is also a function of the number of data points and the degree of interpolation. This range will also be computed by the demo program, and the user will only be allowed to input a number in this range. The range of valid X values is:

$$X(1) \le X \le X(NPTS - NDEG)$$

This numerical method for calculating derivatives is based on Lagrange's method. It is described in Chapter 5 of Reference 7. Numerical derivatives can be calculated by differentiating the following form of Lagrange's interpolation formula of order N:

$$g(x) = \frac{(x - x_1)(x - x_2) \ldots (x - x_N)}{(x_0 - x_1)(x_0 - x_2) \ldots (x_0 - x_N)} f_0$$

$$+ \frac{(x - x_0)(x - x_2) \ldots (x - x_N)}{(x_1 - x_0)(x_1 - x_2) \ldots (x_1 - x_N)} f_1$$

$$\vdots$$

$$+ \frac{(x - x_0)(x - x_1) \ldots (x - x_{N-1})}{(x_N - x_0)(x_N - x_1) \ldots (x_N - x_{N-1})} f_N$$

Differentiation

Subroutine DERIV3

Purpose:

The subroutine DERIV3 estimates the first derivative of user–supplied tabulated data of the form y = f(x) using the method of cubic splines.

Syntax:

CALL DERIV3 (N, X(), Y(), XVAL, DVAL)

Where:

N	= number of X and Y data points [input]
X()	= array of X data points [input] (1 dimensional array; N rows by 1 column)
Y()	= array of Y data points [input] (1 dimensional array; N rows by 1 column)
XVAL	= X argument [input]
DVAL	= derivative value [output]

Comments:

In this algorithm the Y() array is the dependent data and the X() array is the independent data;

$$Y_i = f(X), \quad i = 1, 2, \ldots, NPTS.$$

A minimum of *two* X and Y data points must be provided and should be input by the user in ascending order in the array X(). The demo program checks for a minimum of two data points but does not check for ascending order.

Non-linear Equations

Subroutines QUADRATC, CUBIC and QUARTIC

Purpose:

The subroutines QUADRATIC, CUBIC, and QUARTIC solve for the real roots of a quadratic, cubic or quartic equation, respectively.

Syntax:

CALL QUADRATIC (C1, C2, C3, X1, X2, NROOT)

CALL CUBIC (B1, B2, B3, B4, X1, X2, X3, NROOT)

CALL QUARTIC (A1, A2, A3, A4, A5,
Q1, Q2, Q3, Q4, NROOT)

Where:

C1, C2, C3 = quadratic equation coefficients [input]

B1, B2, B3, B4 = cubic equation coefficients [input]

A1, A2, A3, A4, A5 = coefficients of the quartic equation [input]

X1, X2, X3, Q1, Q2, Q3, Q4 = real roots [output]

NROOT = number of real roots [output]

Comments:

The form of the quadratic equation is

$$C_1x^2 + C_2x + C_3 = 0$$

and the solution is provided by the subroutine QUADRATIC.

The form of the cubic equation is

$$B_1x^3 + B_2x^2 + B_3x + B_4 = 0$$

which can be solved with the subroutine CUBIC.

The quartic equation is given by

$$A_1x^4 + A_2x^3 + A_3x^2 + A_4x + A_5 = 0$$

which is solved using the subroutine QUARTIC.

The subroutine QUADRATIC is a stand–alone subroutine. However, the subroutine CUBIC requires the subroutine QUADRATIC, and the subroutine QUARTIC requires both the subroutines CUBIC and QUADRATIC.

The demo program will ask the user which of the three polynomial equations he or she would like to solve. The demo program DEMOPOLY will also check to make sure the first coefficient for a cubic or quartic equation input by the user is non–zero. This prevents a divide by zero error in the subroutines CUBIC and QUARTIC.

The user must be careful not to call either of these subroutines with a value of zero for the coefficient of the highest order term in x or the software will malfunction.

The quadratic equation subroutine is in the file QUADRATC on the QuickPak Scientific diskette. The cubic equation subroutine is in the file CUBIC, and the quartic equation subroutine can be found in the file QUARTIC.

Non-linear Equations

Subroutine POLYROOT

Purpose:

The subroutine POLYROOT solves for the real and complex roots of a polynomial using Newton's method.

Syntax:

```
CALL POLYROOT (COEF(), NPOLY, ROOTR(),
                ROOTI(), NFLG)
```

Where:

COEF()	= polynomial coefficients [input] (1 dimensional array; NPOLY + 1 rows by 1 column)
NPOLY	= order of the polynomial [input]
ROOTR()	= real parts of roots [output] (1 dimensional array; NPOLY + 1 rows by 1 column)
ROOTI()	= imaginary parts of roots [output] (1 dimensional array; NPOLY + 1 rows by 1 column)
NFLG	= error flag [output] (0 = no error, 1 = iterations > 500)

Comments:

The polynomial is assumed to be of the form:

$$f(x) = A^{N+1} x^N + A^N x^{N-1} + A^{N-1} x^{N-2} + \ldots + A_1$$

The order of the polynomial must be less than or equal to 36. *Please note that the leading coefficient* A^{N+1} *cannot be zero.*

Non-linear Equations

Subroutine REALROOT

Purpose:

The subroutine REALROOT solves for the real root of a user–defined function of the form y = f(x) using Brent's method. This algorithm does not require the user to supply function derivatives.

Syntax:

```
CALL REALROOT (XL, XU, TOL, MAXITER, NITER,
                                XROOT, FROOT)
```

Where:

XL	= lower bound of search interval [input]
XU	= upper bound of search interval [input]
TOL	= convergence criteria [input]
MAXITER	= maximum number of iterations [input]
NITER	= number of actual iterations [output]
XROOT	= real root of y = f(x) = 0 [output]
FROOT	= function value [output]

Comments:

This algorithm will calculate a single real root of a user–defined function of the form y = f(x). The user must specify a search interval which is known to contain a single real root of the defined function. The bounds of the search interval are passed to the subroutine through the two variables XL and XU.

The subroutine REALROOT requires a subroutine which evaluates a function defined by the user. This function is specified in a subroutine called USERFUNC which is coded as:

```
SUB USERFUNC (X, FVAL) STATIC
```

In the parameter list, X is the function argument and FVAL is the function evaluated at X. The subroutine REALROOT handles all interaction with this subroutine.

A convergence tolerance of 1D–8 is recommended. The number of iterations permitted and required depends on the size of the search interval. A value between 10 and 20 is recommended.

The demo program attempts to find the real root of

$$y = f(x) = x\ e^x - 10$$

within an interval specified by the user.

Note that x = 1.745528 is a root of this equation.

A subroutine defining this function is stored on the QuickPak Scientific diskette in the file USERFUN3. The algorithm is described in Reference 5. For this example USERFUNC is coded as follows:

```
SUB USERFUNC (X, FVAL) STATIC
    ' User–defined function subroutine
    ' F(X) = X * e^X – 10
    ' Input
    '   X = function argument
    ' Output
    '   FVAL = function value at X
    FVAL = X * EXP(X) – 10#
END SUB
```

The demo program also illustrates a simple method for bracketing the root of any user–defined function. The software will prompt the user for a value of X for initiating the search, a step size, a step size multiplier, and a rectification interval. If the step size is positive or negative, the algorithm will search to the right or left of the initial point, respectively. The step size multiplier geometrically accelerates the search and must not be too large or it may "step" over a bracket. The rectification interval is the maximum distance the algorithm will search before "resetting" the search parameters. This prevents the method from using very large step sizes during its search.

The bracketing algorithm will also request a maximum number of bracketing iterations. A value of 50 is recommended.

The bracketing subroutine is stored on the QuickPak Scientific disk under the filename BROOT. The proper syntax for calling this algorithm is as follows:

```
CALL BROOT (XI, DX, XMULT, DXMAX, NROOT,
                                XB1, XB2, IER)
```

Where:

XI	= initial X search value [input]
DX	= initial X step size [input]
XMULT	= X step size multiplier [input] (XMULT > 0)
DXMAX	= bracketing rectification interval [input]
NROOT	= maximum number of iterations [input]
XB1	= first X bracket value [output]
XB2	= second X bracket value [output]
IER	= error flag [output] (0 = no error, 1 = iterations > NROOT)

If the algorithm is successful, the root will be bracketed between XB1 and XB2, and the error flag IER will return with the value 0. If the algorithm attempts more than NROOT searches, this flag will be set to the number 1.

Non-linear Equations

Subroutine NEWTON

Purpose:

The subroutine NEWTON solves for a single real root of a user–defined non–linear equation using a combination Newton–Raphson and bisection method. The first derivative of the user–defined function is also required.

Syntax:

CALL NEWTON (X1, X2, TOL, MAXITER, ROOT, NITER)

Where:

X1	= initial X value of search interval [input]
X2	= final X value of search interval [input]
TOL	= convergence tolerance [input]
MAXITER	= maximum number of iterations [input]
ROOT	= root of non-linear equation [output]
NITER	= number of actual iterations [output]

Comments:

A real root *must* be bracketed by the values of X1 and X2 in order for this subroutine to work properly.

This subroutine requires a user–supplied support subroutine called USERFUNC which returns values for the function and its first derivative for any X value.

This QuickBASIC subroutine must be coded as follows:

```
SUB USERFUNC (X, FX, FPX) STATIC
```

In the parameter list, X is the argument, FX is the function value, and FPX is the derivative value at X.

The QuickPak Scientific demo program searches for a single real root of the following non–linear equation:

$$y = f(x) = x - e^{\frac{1}{x}}$$

The first derivative of this function is

$$\frac{dy}{dx} = 1 + \frac{e^{\frac{1}{x}}}{x^2}$$

A subroutine which calculates this function and its derivative is stored on the QuickPak Scientific diskette in the file USERFUN9.

For this algorithm, a convergence criteria of 1D–10 is recommended. A number between 50 and 100 should work for the maximum number of iterations.

Please note that the demo program will check the values of X1 and X2 to make sure that a single real root is bracketed. If a root is not bracketed, the program will display the message

Root must be bracketed!!

and request new values for X1 and X2. The function defined here has a root at X = 1.7632228.

The demo program will display the root and function values, and the actual number of algorithm iterations.

Chapter 9 of Reference 1 contains a discussion and FORTRAN listing of this Newton–Raphson/bisection algorithm.

Non-linear Equations

Subroutine NLINEAR

Purpose:

The subroutine NLINEAR can be used to solve for the real roots of an unconstrained system of non–linear equations using Newton's method and analytic partial derivatives which must be supplied by the user.

Syntax:

CALL NLINEAR (N, MAXITER, XTOL, FTOL, NITER, X(), IER)

Where:

N	= number of equations in the system [input]
MAXITER	= maximum number of iterations [input]
XTOL	= X convergence criteria [input]
FTOL	= function convergence criteria [input]
NITER	= number of actual iterations [output]
X()	= solution vector [output] (1 dimensional array; N rows by 1 column)
IER	= error flag [output] (0 = no error, 1 = singular matrix)

Comments:

This subroutine requires the subroutine LINEAR and a subroutine called USERFUNC which returns the *negative* function values and partial derivatives of a user–defined system of non–linear equations. The subroutine NLINEAR handles all communications with these two support subroutines.

The subroutine USERFUNC is coded as:

```
SUB USERFUNC (X(), FX(), DFDX()) STATIC
```

In the parameter list, X() is the function argument vector, FX() is the vector of *negative* function values evaluated at X(), and DFDX() holds the partial derivatives of the function, also evaluated at X().

The two-dimensional matrix of partial derivatives, DFDX(), is called the *Jacobian* and must be coded in the form:

$$\nabla f = \text{DFDX}(i, j) = \partial f_i(\bar{x}) / \partial x_j$$

for i = 1 to N and j = 1 to N.

The subroutine NLINEAR iteratively solves the following matrix equation:

$$\left[\sum_{j=1}^{N} \frac{\partial f_i}{\partial x_j} \right] \delta x_j = - f_i \qquad i = 1, 2, \ldots, N$$

The correction to an old guess x_i^{old} is

$$x_i^{new} = x_i^{old} + \delta x_i \qquad i = 1, 2, \ldots, \text{NITER}$$

The demo program attempts to solve the non–linear system of equations given by:

$$f_1(x_1, x_2, x_3, x_4) = - x_1^2 - x_2^2 - x_3^2 + x_4 = 0$$

$$f_2(x_1, x_2, x_3, x_4) = x_1^2 + x_2^2 + x_3^2 + x_4^2 - 1 = 0$$

$$f_3(x_1, x_2, x_3, x_4) = x_1 - x_2 = 0$$

$$f_4(x_1, x_2, x_3, x_4) = x_2 - x_3 = 0$$

The first few terms of the Jacobian for this non-linear system are as follows:

$$DFDX(1, 1) = \frac{\partial f_1}{\partial x_1} = -2x_1 \qquad DFDX(1, 2) = \frac{\partial f_1}{\partial x_2} = -2x_2$$

$$DFDX(1, 3) = \frac{\partial f_1}{\partial x_3} = -2x_3 \qquad DFDX(1, 4) = \frac{\partial f_1}{\partial x_4} = 1$$

The user must input an initial guess for the solution vector $\bar{X}$. The software will check for either an X value or function value convergence. A convergence criteria of 1D–6 is recommended for both the X value and function value convergence criteria. The maximum number of iterations allowed should range from 5 to 10.

A solution to this non–linear system is

$x_1 = .4538847, \qquad x_2 = .4538847,$
$x_3 = .4538847, \qquad x_4 = .61803398.$

A subroutine defining this system of equations and its gradient is stored on the QuickPak Scientific diskette in the file USERFUN4.

If a singular matrix or other problem is encountered during the solution process, the error flag IER will be set to 1. Otherwise it will be set to 0. If an error does occur, the user should first try a new initial guess. Another source of errors may be incorrect coding of the gradient or function.

This QuickPak Scientific subroutine is a QuickBASIC implementation of the algorithm described in Chapter 9 of Reference 1.

Non-linear Equations

Subroutine MINIMIZ2

Purpose:

The non–linear optimization subroutine MINIMIZ2 can also be used as a non–linear regression method to solve for the real roots of an unconstrained system of non–linear equations.

Syntax:

The proper procedure for using the subroutine MINIMIZ2 is described on pages 100-101 of this manual.

Comments:

A general system of N non–linear equations can be expressed in the form:

$$f_1(x_1, x_2, \ldots, x_N) = 0$$

$$f_2(x_1, x_2, \ldots, x_N) = 0$$

$$\vdots$$

$$f_N(x_1, x_2, \ldots, x_N) = 0$$

We define a minimizing function as follows:

$$f(x_1, x_2, \ldots, x_N) = f_1^2(x_1, x_2, \ldots, x_N) + f_2^2(x_1, x_2, \ldots, x_N) + \cdots + f_N^2(x_1, x_2, \ldots, x_N)$$

If $x_1, x_2, \ldots, x_N$ are roots of the non–linear system, then the minimum function value should be equal to zero (within a small tolerance). We can use the subroutine MINIMIZ2 to find the minimum of the function f. However, this approach is valid if and only if the function value for $f \approx 0$. It may happen that the converged values of $x_1, x_2, \ldots, x_N$ which minimize f are *not* roots of the non–linear system.

The demo program attempts to solve the system of three non–linear equations defined by:

$$f_1(x_1, x_2, x_3) = 3x_1 - \cos(x_2 x_3) - \frac{1}{2}$$

$$f_2(x_1, x_2, x_3) = x_1^2 - 81(x_2 + 0.1)^2 + \sin x_3 + 1.06$$

$$f_3(x_1, x_2, x_3) = e^{-x_1 x_2} + 20x_3 + \frac{10\pi - 3}{3}$$

The format of the user–coded subroutine USERFUNC which calculates the value of the minimizing function is described on page 100. The gradient of this system is computed with the adaptive numerical method described in Reference 8.

A subroutine which computes the function values of this system of non–linear equations is stored on the QuickPak Scientific disk in the file USERFUN0.

This non–linear system of equations has a solution at

$$x_1 = 0.5, \quad x_2 = 0, \text{ and } x_3 = -0.52359877.$$

If the minimum function value printed by the demo program is not equal to zero (within a small tolerance), the x values are not roots of the non–linear system. The user should try different initial conditions and repeat this process.

Optimization

Subroutine MINIMA1

Purpose:

The subroutine MINIMA1 solves for a minimum or maximum of an unconstrained scalar function of one variable. This numerical method does not require derivatives.

Syntax:

```
CALL MINIMA1 (X1, DX, XMULT, XTOL, NITER,
                                XMIN, FMIN)
```

Where:

X1	=	initial guess for solution [input]
DX	=	search step size for X [input]
XMULT	=	step size multiplier for X [input]
XTOL	=	convergence criteria [input]
NITER	=	number of iterations [input]
XMIN	=	minimum or maximum X value [output]
FMIN	=	minimized function value [output]

Comments:

This algorithm is useful when an interval containing a function minimum or maximum is not known. The software will simply search to the right or left of an initial guess supplied by the user in the direction of the closest function minimum or maximum (downhill or uphill, respectively). This numerical method does not require the calculation of function derivatives.

The search for a minimum or maximum is accelerated by the step size multiplier XMULT. Do not make this value too large or the algorithm may step over the minimum or maximum.

The user must supply an analytic function for which a minimum or maximum is desired. This function is defined in a subroutine called USERFUNC which is coded as:

```
SUB USERFUNC (X, FVAL) STATIC
```

In the parameter list, X is the function argument and FVAL is the negative or positive value of the function evaluated at X. The *negative* function value should be returned if the algorithm is used to find a minimum, and the *positive* function value returned when searching for a function maximum. The main subroutine provides all interaction with this subroutine.

A value of 1.25 is recommended for the step size multiplier and a convergence criteria of 1D–6 is reasonable for most functions. The "best" value for the step size multiplier depends on the behavior of the function in the vicinity of a minimum or maximum. A graphics plot of this function would provide valuable insight about the characteristics of any user–defined function.

The demo program DEMOMIN1 attempts to find a minimum of the function defined by

$$f(x) = x^4 - 12\ x^3 + 15\ x^2 + 56\ x - 60$$

given an initial guess for x.

A minimum of this function is located at x = –.870173.

A subroutine defining this function is stored on the QuickPak Scientific diskette in the file USERFUN5.

Optimization

Subroutine MINIMA2

Purpose:

The subroutine MINIMA2 solves for a minimum or maximum of an unconstrained scalar function of one variable using Brent's method which does not require derivatives.

Syntax:

```
CALL MINIMA2 (A, B, EPS, MAXITER, NITER,
                                  XMIN, FMIN)
```

Where:

A	= initial X search value [input]
B	= final X search value [input]
EPS	= convergence criteria [input]
MAXITER	= maximum number of iterations [input]
NITER	= actual number of iterations [output]
XMIN	= minimum or maximum X value [output]
FMIN	= minimized function value [output]

Comments:

This algorithm is useful when an interval containing the function minimum or maximum is known. The search interval is specified to the subroutine through the values of A and B. This method is based on Brent's technique and no derivative calculations are required. It is very important to pass this subroutine an interval which contains a function minimum or maximum. Otherwise the software may malfunction.

The user must supply an analytic function for which the minimum or maximum is desired. This function is defined in a subroutine called USERFUNC which has the following syntax:

```
SUB USERFUNC (X, FVAL) STATIC
```

In the parameter list, X is the function argument and FVAL is the negative or positive value of the function evaluated at X. The *positive* function value should be returned if the algorithm is used to find a minimum, and the *negative* function value returned when searching for a function maximum. The main subroutine provides all interaction with this subroutine.

An integer value of 100 is recommended for the maximum number of iterations, and a convergence criteria of 1D–8 is reasonable for most functions.

Both an ALGOL and FORTRAN version of Brent's algorithm are described in Reference 5. Chapter 10 of Reference 1 also discusses a slightly modified version of this algorithm.

The companion demo program for this algorithm attempts to find a single minimum of the function defined by

$$y = f(x) = \frac{x^3 + 5}{\sqrt{x + 2}}$$

within a search interval specified by the user.

A minimum of this function is located at x = .57934785.

A subroutine defining this function is stored on the QuickPak Scientific diskette in the file USERFUN6.

For this example, the source code of the USERFUN subroutine is as follows;

```
SUB USERFUNC (X, FX) STATIC
    ' Objective function subroutine
    ' F(X) = X^3 + 5 / SQR(X + 2)
    ' Input
    '  X = function argument
    ' Output
    '  FX = value of objective function at X
    FX = (X ^ 3 + 5#) / SQR(X + 2#)
END SUB
```

The demo program also illustrates a simple routine which *brackets* the minimum of this function. This subroutine requires several inputs in order to operate correctly. These inputs include such things as an initial search value, a search step size, a step size multiplier, and a maximum number of bracketing iterations.

The correct syntax for using this subroutine is as follows:

```
CALL BMINIMA (XI, DX, XMULT, NBRAC,
                    XB1, XB2, IER)
```

where

XI	= initial X search value [input]
DX	= X step size [input]
	+ = search forward
	– = search backward
XMULT	= X step size multiplier [input]
NBRAC	= maximum number of iterations [input]
XB1	= first X bracket value [output]
XB2	= second X bracket value [output]
IER	= error flag [output]
	0 = no error
	1 = iterations required exceeds NBRAC

Optimization

Subroutine MINIMIZ1

Purpose:

The subroutine MINIMIZ1 solves for a minimum or maximum of an unconstrained scalar function of several variables using analytic partial derivatives supplied by the user.

Syntax:

CALL MINIMIZ1 (METHOD, N, EPS, MAXITER, IFLAG, NITER, F, X())

Where:

METHOD = method of solution [input]
- 1 = conjugate gradient
- 2 = quasi–newton)

N = number of variables [input]

EPS = convergence criteria [input]

MAXITER = maximum number of iterations [input]

IFLAG = diagnostic flag [output]
- 0 = converged solution
- 1 = maximum number of function evaluations
- 2 = linear search failure
- 3 = search vector failure

NITER = actual number of iterations [output]

F = minimized function value [output]

X() = input as an initial guess for the solution vector and output as the final solution vector (1 dimensional array; N rows by 1 column)

Comments:

This algorithm provides two methods for solving multi–variable minimum and maximum problems. These are the conjugate gradient and the quasi–newton methods.

One method may be superior to the other for different multi–variable functions and initial solution guesses. It is best to try both methods and experiment with the convergence criteria and initial guesses.

The user must supply an analytic function for which the minimum or maximum is desired. This function is coded in a subroutine called USERFUNC as:

SUB USERFUNC (X(), F, GRADIENT()) STATIC

In the parameter list, X() is the function argument vector, F is the negative or positive value of the function evaluated at X(), and GRADIENT() is the function gradient evaluated at X(). The *positive* function value should be returned if the algorithm is used to find a minimum and the *negative* function value returned when searching for a function maximum. The main subroutine handles all interaction with this subroutine.

The elements of the gradient array are coded as:

$$\nabla f = \text{GRADIENT}(i) = \partial f/\partial x_i \qquad i = 1, 2, \ldots, N$$

where N is the number of variables.

The demo program DEMOMNZ1 attempts to find a minimum of Wood's function which is a popular test for optimization algorithms. This function is stored on your QuickPak Scientific disk in the file USERFUN7.

A value of 100 to 300 is recommended for the maximum number of iterations and a convergence criteria of 1D–8 is reasonable for most functions.

Wood's function is a scalar function of four variables and is given by the following expression:

$$f(x_1, x_2, x_3, x_4) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 + 90(x_4 + x_3^2)^2 + (1 - x_3)^2 + 10.1\left[(x_2 - 1)^2 + (x_4 - 1)^2\right] + 19.9(x_2 - 1)(x_4 - 1)$$

The gradient or partial derivatives of Wood's function can be determined from

$$\text{GRADIENT}(1) = \frac{\partial f}{\partial x_1} = -400x_1(x_2 - x_1^2) - 2(1 - x_1)$$

$$\text{GRADIENT}(2) = \frac{\partial f}{\partial x_2} = 200(x_2 - x_1^2) + 20.2(x_2 - 1) + 19.8(x_4 - 1)$$

$$\text{GRADIENT}(3) = \frac{\partial f}{\partial x_3} = -360x_3(x_4 - x_3^2) - 2(1 - x_3)$$

$$\text{GRADIENT}(4) = \frac{\partial f}{\partial x_4} = 180(x_4 - x_3^2) + 20.1(x_4 - 1) + 19.8(x_2 - 1)$$

Note that this function has a minimum at $x_1 = x_2 = x_3 = x_4 = 1$.

You might want to try initial guesses which are both near and far from these values.

For this example, the USERFUNC subroutine is coded as:

```
SUB USERFUNC (X(), F, GRADIENT()) STATIC
  ' User–defined function and gradient subroutine
  ' F(x) = 100(x2 – x1^2)^2 + (1 – x1)^2 + 90(x4 – x3^2)^2
  '        + (1 – x3^2)^2 + 10.1((x2 – 1)^2 + (x4 – 1)^2)
  '        + 19.9(x2 – 1)(x4 – 1)
  ' Input
  '  X() = function argument vector
  ' Output
  '  F            = scalar value of objective function at X()
  '  GRADIENT() = function gradient vector evaluated at X()
  A = X(2) – X(1) * X(1)
  B = X(4) – X(3) * X(3)
  ' calculate function value
  F = 100# * A * A + (1# – X(1)) ^ 2 + 90# * B * B
       + (1# – X(3)) ^ 2
  F = F + 10.1# * ((X(2) – 1#) ^ 2 + (X(4) – 1#) ^ 2)
  F = F + 19.8# * (X(2) – 1#) * (X(4) – 1#)
  ' calculate elements of gradient
  GRADIENT(1) = –2# * (200# * X(1) * A + 1# – X(1))
  GRADIENT(2) = 2# * (100# * A + 10.1# * (X(2) – 1#)
                + 9.9# * (X(4) – 1#))
  GRADIENT(3) = –2# * (180# * X(3) * B + 1# – X(3))
  GRADIENT(4) = 2# * (90# * B + 10.1# * (X(4) – 1#)
                + 9.9# * (X(2) – 1#))
END SUB
```

MINIMIZ1 is a QuickBASIC implementation of ACM Algorithm # 500.

Optimization

Subroutine MINIMIZ2

Purpose:

The subroutine MINIMIZ2 solves for a minimum or maximum of a scalar function of several variables using an adaptive numerical gradient computed by the software.

Syntax:

CALL MINIMIZ2 (METHOD, N, EPS, MAXITER, IFLAG, NITER, F, X())

Where:

METHOD = method of solution [input]
- 1 = conjugate gradient
- 2 = quasi–newton)

N = number of variables [input]

EPS = convergence criteria [input]

MAXITER = maximum number of iterations [input]

IFLAG = diagnostic flag [output]
- 0 = converged solution
- 1 = maximum number of function evaluations
- 2 = linear search failure
- 3 = search vector failure

NITER = actual number of iterations [output]

F = minimized function value [output]

X() = input as an initial guess for solution vector and output as the final solution vector (1 dimensional array; N rows by 1 column)

Comments:

This algorithm is similar to subroutine MINIMIZ1 except that a numerical approximation of the function gradient is used.

This approach is useful when the gradient is difficult to derive analytically. The function gradient is computed with an adaptive numerical method. This method is described in Reference 8.

The user must supply an analytic function for which the minimum or maximum is desired. This function is defined in a subroutine called USERFUNC which is coded as:

```
SUB USERFUNC (X(), F) STATIC
```

In the parameter list, X() is the function argument vector, and F is the negative or positive value of the function evaluated at X(). The positive value should be returned if the algorithm is used to find a minimum and the negative function value returned when searching for a function maximum. The subroutine MINIMIZ2 handles all interaction with this subroutine.

A value of 100 to 300 is recommended for the maximum number of iterations, and a convergence criteria of .01 is reasonable for most functions.

The demo program for this subroutine also attempts to find a minimum of Wood's function. This mathematical form of this function is described on page 98.

One solution to this problem is $x_1 = x_2 = x_3 = x_4 = 1$.

This function is stored on the QuickPak Scientific diskette in the file USERFUN8.

This algorithm is a QuickBASIC implementation of the FORTRAN version of ACM Algorithm #500.

Optimization

Program DEMONLP

Purpose:

This computer program solves the multi–variable, constrained, non–linear optimization problem using the Method of Multipliers and Quasi–Newton minimization.

Syntax:

The non–linear optimization algorithm is a very large program which is designed to be used as a stand–alone program. The user simply provides a QuickBASIC subroutine which defines the objective function, and any equality and inequality constraints. The software interactively prompts the user for such things as initial guesses, the type of derivative computation desired, and the maximum number of iterations.

Comments:

The numerical problem which can be solved by this digital computer program is stated as follows:

$$\text{Minimize} \qquad y = f(\bar{x})$$

$$\text{Subject to} \qquad h(\bar{x}) = 0$$

$$g(\bar{x}) \geq 0$$

where

$f(\bar{x})$ is a vector system of non–linear equations
$h(\bar{x})$ is a vector of *equality* constraints
$g(\bar{x})$ is a vector of *inequality* constraints

The scalar y is called the *objective function* and the elements of the vector $\bar{x}$ are called the *control variables.*

A support subroutine named USRFUN contains the source code which defines both the objective function and constraints. This subroutine must be coded as follows:

```
SUB USRFUN (X(), F(), G(), H()) STATIC
```

In this parameter list, the array X() is the vector of control variables, F() is returned as the objective function vector, G() is the vector of inequality constraints, and the H() array contains the vector of equality constraints. Please note that the array defined by F() contains only one element which represents the scalar objective function evaluated at the current values of the control variables X().

Extreme care should be used when coding this subroutine for your particular problem. It is important to avoid any singularities in both the objective function and constraints.

The DEMONLP program attempts to minimize the multi–variable objective function defined by

$$f(x_1, x_2, x_3, x_4, x_5) = e^{x_1 x_2 x_3 x_4 x_5}$$

subject to the three equality constraints given by

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 10 = 0$$

$$x_2 x_3 - 5x_4 x_5 = 0$$

$$x_1^3 + x_2^3 + 1 = 0$$

and the single inequality constraint defined by

$$h(1) = x_4 - x_5 \geq 0$$

This problem has one solution at

$$x_1 = -1.7171 \qquad x_2 = 1.5957 \qquad x_3 = 1.8272$$

$$x_4 = -.7636 \qquad x_5 = -.7636$$

For this example, the USRFUN source code is as follows:

```
SUB USRFUN (X(), F(), G(), H()) STATIC
   ' Objective function and constraints subroutine
   ' F(X) = e^(X1*X2*X3*X4*X5)
   ' Input
   '  X() = array of control variables (5 rows)
   ' Output
   '  F() = objective function vector (1 element)
   '  G() = vector of inequality constraint functions (3 rows)
   '  H() = vector of equality constraint functions (1 row)
   ' objective function
   ARG = X(1) * X(2) * X(3) * X(4) * X(5)
   ' check for large arguments
   IF (ARG > 200#) THEN ARG = 200#
   F(1) = EXP(ARG)
   ' define equality constraints
   H(1) = X(1) ^ 2 + X(2) ^ 2 + X(3) ^ 2 + X(4) ^ 2
            + X(5) ^ 2 - 10#
   H(2) = X(2) * X(3) - 5# * X(4) * X(5)
   H(3) = X(1) ^ 3 + X(2) ^ 3 + 1#
   ' define inequality constraint
   G(1) = X(4) - X(5)
END SUB
```

To avoid problems when evaluating the QuickBASIC EXP function for very large function arguments, this subroutine defaults to a value of ARG = 200# whenever the function argument is larger than 200..

The software will interactively prompt you for such things as an initial guess for each element of the control variable vector, the type of derivative computations desired, and the maximum number of non–linear iterations.

The control variable prompt will ask you to input each element of the vector by *rows*. There are two types of numerical derivative calculations available; one–sided finite difference and symmetric finite difference. The symmetric algorithm is usually more accurate but requires twice as many function evaluations and more CPU time to execute. A value of 20 is usually adequate for the number of iterations.

The driver program will also ask if you would like to display the intermediate results of iterations. If you select *y* for yes, the program will print important information about how the algorithm is proceeding as it searches for a solution.

Initializing the NLP Software

There are three important variables you must edit in order for the software to work properly. These QuickBASIC variables are at the beginning of the main program and involve the problem definition. The first is called NX and defines the number of control variables. The second is NG and it defines the number of inequality constraints in your problem. The last is named NH and it specifies the number of equality constraints. Simply change these within the QuickBASIC environment.

The rest of the software initialization is performed in a support subroutine called NLPINZ. This subroutine *hardwires* such things as convergence tolerances and other values, and allocates array space. It also checks for errors in your problem definition and displays appropriate error messages.

For the majority of non–linear problems, the constants defined in subroutine NLPINZ will work fine. However, the purpose of each is clearly commented so that you may experiment with other values.

Interpolation

Subroutine CSFIT1

Purpose:

The subroutine CSFIT1 calculates a *natural* cubic spline interpolation of tabulated x and y data of the form y = f(x) provided by the user.

Syntax:

CALL CSFIT1 (N, X(), Y(), XVAL, FVAL)

Where:

N	= number of X and Y data points [input]
X()	= array of X data points [input] (1 dimensional array; N rows by 1 column)
Y()	= array of Y data points [input] (1 dimensional array; N rows by 1 column)
XVAL	= X argument for interpolation [input]
FVAL	= interpolated function value [output]

Comments:

This subroutine uses the method of cubic splines and iterative successive over–relaxation (SOR) to interpolate tabulated information of the form y = f(x).

The general form of a cubic polynomial is

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$

for j = 0, 1, 2, . . , N – 1.

For a *free* or *natural* boundary cubic spline

$$S''(x_0) = S''(x_N) = 0$$

This condition implies that the second derivative of the cubic spline is zero at the two endpoints of the interval of discrete data points.

This algorithm provides an interpolated function value for any X data point within the valid range defined by

$$X(1) \leq X \leq X(N).$$

A minimum of *two* X and Y data points must be provided and should be input by the user in ascending order in the array X(). The demo program checks for a minimum of two data points but does not check for ascending order.

Ascending order implies that

$$X(1) < X(2) < X(3) < \ldots < X(N)$$

where N is the total number of X and Y data points.

Interpolation

Subroutine CSFIT2

Purpose:

The subroutine CSFIT2 calculates a *clamped* cubic spline interpolation of tabulated x and y data of the form y = f(x) input by the user.

Syntax:

CALL CSFIT2 (N, X(), Y(), YP1, YPN, X, Y)

Where:

N	= number of X and Y data points [input]
X()	= array of X data points [input] (1 dimensional array; N rows by 1 column)
Y()	= array of Y data points [input] (1 dimensional array; N rows by 1 column)
YP1	= first derivative at data point 1 [input]
YPN	= first derivative at data point N [input]
X	= X data point for interpolation [input]
Y	= interpolated Y data point [output]

Comments:

The general form of a cubic polynomial is

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$

for j = 0, 1, 2, . . , N – 1.

For a *clamped* boundary cubic spline

$$S'(x_0) = f'(x_0) \qquad \text{and} \qquad S'(x_N) = f'(x_N)$$

This condition implies that the slope or first derivative of the cubic spline matches that of the original function f on the boundaries of the interval.

This algorithm provides an interpolated Y value for an X value input by the user within the valid range determined by the data points.

The range of valid X data is $X(1) \leq X \leq X(N)$.

A minimum of two X and Y data points must be provided and should be input by the user in ascending order in the array X(). The demo program checks for a minimum of two data points but does not check for ascending order.

Ascending order implies that

$$X(1) < X(2) < X(3) < \ldots < X(N)$$

where N is the total number of X and Y data points.

The user should consider using one of the QuickPak Scientific numerical differentiation routines which work with tabulated data to calculate the two derivative values required by this subroutine.

This algorithm is described in Chapter 3 of Reference 1.

Interpolation

Subroutine INTERP1

Purpose:

The subroutine INTERP1 performs a linear interpolation of tabulated x and y data of the form y = f(x) input by the user.

Syntax:

CALL INTERP1 (N, X(), Y(), XVAL, FVAL)

Where:

N	= number of X and Y data points [input]
X()	= array of X data points [input] (1 dimensional array; N rows by 1 column)
Y()	= array of Y data points [input] (1 dimensional array; N rows by 1 column)
XVAL	= X argument for interpolation [input]
FVAL	= interpolated function value [output]

Comments:

A minimum of two X and Y data points must be provided and should be input by the user in ascending order in the array X(). The demo program checks for a minimum of two data points but does not check for ascending order.

Ascending order implies that

$$X(1) < X(2) < X(3) < \ldots < X(N)$$

The range of valid X data which can be interpolated is

$$X(1) \leq X \leq X(N).$$

Interpolation

Subroutine INTERP2

Purpose:

The subroutine INTERP2 performs a bilinear interpolation of tabulated x, y and z data of the form z = f(x,y) input by the user.

Syntax:

```
CALL INTERP2 (NX, NY, X(), Y(), Z(), XVAL,
                                YVAL, ZVAL)
```

Where:

NX = number of X data points [input] (NX ≥ 2)

NY = number of Y data points [input] (NY ≥ 2)

X() = array of X data points [input]
(1 dimensional array; NX rows by 1 column)

Y() = array of Y data points [input]
(1 dimensional array; NY rows by 1 column)

Z() = array of function values [input]
(2 dimensional array; NX rows by NY columns)

XVAL = X value for interpolation [input]

YVAL = Y value for interpolation [input]

ZVAL = interpolated function value [output]

Comments:

A minimum of *two* data points for X, Y and Z must be provided and should be input by the user in ascending order in the arrays X() and Y().

Ascending order implies that

$$X(1) < X(2) < X(3) < \ldots < X(N)$$

and

$$Y(1) < Y(2) < Y(3) < \ldots < Y(N)$$

The number of X and Y data points do not need to be the same, but the function values at coordinate intersections must be consistent.

The demo program for this subroutine illustrates how to use INTERP2 to interpolate data for the function defined by:

$$z = f(x,y) = SIN(x + y)$$

The following QuickBASIC code generates a set of X, Y and Z data points for this function.

```
FOR I =1 TO NX
    X(I) = I * DX
    FOR J = 1 TO NY
        Y(J) = J * DY
        Z(I, J) = SIN(X(I) + Y(J))
    NEXT J
NEXT I
```

where NX and NY are the number of X and Y data points specified by the user, respectively. DX and DY are increments in each direction and are also input by you.

Since we know the function value for any set of x and y coordinates (because z is an analytic function), the demo program also provides an interpolation error estimate.

The two domains of valid X and Y data points which can be interpolated are

$$X(1) \leq X \leq X(N) \quad \text{and} \quad Y(1) \leq Y \leq Y(N).$$

Curve Fit

Subroutine FFIT

Purpose:

The subroutine FFIT performs a curve fit to simple equations of tabulated data of the form y = f(x) supplied by the user.

Syntax:

CALL FFIT (N, X(), Y(), ITYPE, A, B)

Where:

N = number of X and Y data points [input]

X() = array of X data points [input]
(1 dimensional array; N rows by 1 column)

Y() = array of Y data points [input]
(1 dimensional array; N rows by 1 column)

ITYPE = type of function fit [input]

1	= linear	Y = A + B * X
2	= logarithmic	Y = A + B * LOG(X)
3	= exponential	Y = A * EXP(B * X)

A = first function fit coefficient [output]

B = second function fit coefficient [output]

Comments:

A minimum of *three* X and Y data points must be provided and should be input by the user in ascending order in the independent variable X. The demo program checks for a minimum number of data points but does not check for ascending order.

Curve Fit

Subroutine LSQFIT

Purpose:

The subroutine LSQFIT performs a least squares curve fit of tabulated x and y data of the form y = f(x) input by the user.

Syntax:

CALL LSQFIT (NPTS, X(), Y(), MDEG, SDEV, COEF())

Where:

NPTS	= number of X and Y data points [input]
X()	= array of X data points [input] (1 dimensional array; NPTS rows by 1 column)
Y()	= array of Y data points [input] (1 dimensional array; NPTS rows by 1 column)
MDEG	= degree of curve fit [input]
SDEV	= standard deviation of fit [output]
COEF()	= curve fit coefficients [output] (1 dimensional array; MDEG rows by 1 column)

Comments:

The form of the least squares curve fit is:

$$y = f(x) = C_1 + C_2x + C_3x^2 + C_4x^3 + \ldots + C_Nx^N$$

In this algorithm the Y() array is the dependent data and the X() array is the independent data;

$$Y_i = f(X), \quad i = 1, 2, \ldots, \text{NPTS}.$$

A minimum of *three* X and Y data points must be input by the user in ascending order in the array X(). The demo program checks for a minimum of three data points but does not check for ascending order.

Ascending order implies that

$$X(1) < X(2) < X(3) < \ldots < X(N)$$

The degree of the curve fit must be less than or equal to (NPTS – 2).

The demo program will ask the user if he or she would like to fit an X data point with the curve fit coefficients generated by the program. The range of valid X values which can be fit is as follows:

$$X(1) \leq X \leq X(NPTS).$$

The X data point is fit with the following equation:

$$y = f(x) = \sum_{i=1}^{MDEG} COEF_i \left[x^{i-1} \right]$$

where the array of fitting coefficients is contained in the QuickBASIC COEF() array.

This numerical method for curve fitting is based on the method of orthogonal polynomials, and is discussed in Chapter 1 of Reference 7.

Curve Fit

Subroutine SURFIT

Purpose:

The subroutine SURFIT performs a 2–dimensional curve fit of x, y and z tabulated data of the form z = f(x,y) input by the user.

Syntax:

CALL SURFIT (NDEG, NDATA, XDATA(), YDATA(), ZDATA(), COEF())

Where:

NDEG = degree of surface fit [input]

NDATA = number of x, y and z data points [input]

XDATA() = array of X data values [input] (1 dimensional array; NDATA rows by 1 column)

YDATA() = array of Y data values [input] (1 dimensional array; NDATA rows by 1 columns)

ZDATA() = array of Z data values [input] (1 dimensional array; NDATA rows by 1 column)

COEF() = array of surface fit coefficients [output] (2 dimensional array; 11 rows by 11 columns)

Comments:

This algorithm calculates fitting coefficients for surfaces of the form z = f(x,y) based on a 2–dimensional Maclaurin series. Subroutine SURFIT is a QuickBASIC implementation of the FORTRAN program described in Reference 9.

The fitting polynomial can be expressed as:

$$z = f(x,y) = \sum_{k=0}^{N} \sum_{j=0}^{N} A_{jk}\, x^j\, y^i$$

where N is the degree of the polynomial, A_{jk} are the polynomial coefficients, and x and y are the independent variables.

Please note that the polynomial degree must be ≤ 10.

The fitting coefficients calculated by SURFIT are printed out in the following order:

A_{00}

$A_{10}x$, $A_{01}y$

$A_{20}x^2$, $A_{11}xy$, $A_{02}y^2$ and so forth.

The x, y, and xy elements in a particular row are in the form:

x^N, $x^{N-1}y$, $x^{N-2}\,y^2$, . . . , y^N

The demo DEMOSFIT program calculates fitting coefficients for the following user–defined function:

$$z = f(x,y) = -\cos\left(\frac{3\pi x}{10}\right)\cos\left(\frac{\pi y}{10}\right)$$

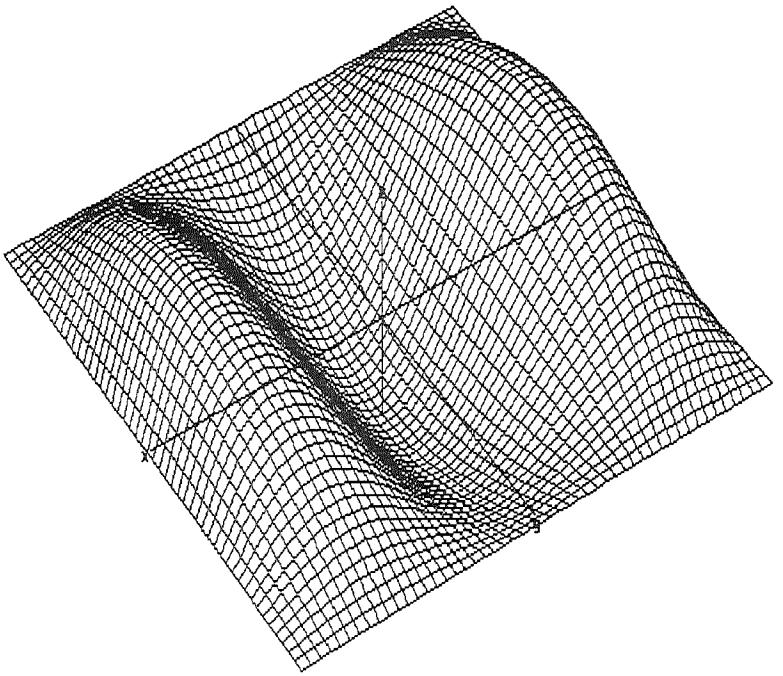
The demo program first generates arrays of x, y and z data points for the range of x and y values given by:

$-5 \le x \le 5$ and $-5 \le y \le 5$

This is done with the following QuickBASIC code:

```
NDATA = 0
FOR I# = -5# TO 5# STEP 1#
    FOR J# = -5# TO 5# STEP 1#
        NDATA = NDATA + 1
        X = I#
        XDATA(NDATA) = I#
        Y = J#
        YDATA(NDATA) = J#
        Z = -COS(3# * PI * X / 10#) * COS(PI * Y / 10#)
        ZDATA(NDATA) = Z
    NEXT J#
NEXT I#
```

The following figure illustrates the shape of this function over the range $-10 \leq X \leq +10$ and $-10 \leq Y \leq +10$.



The demo program will also fit an x and y data point input by the user. This is accomplished in the following subroutine:

SUB SFUNCTION (NDEG, COEF(), X, Y, Z) STATIC

where X, and Y are the data points to fit, and Z is the calculated function value. The other variables are as defined for subroutine SURFIT.

The demo program will also calculate and print the percentage of fit. This is an indication of how well the surface has been fitted by the algorithm. The calculated percentage of fit is based on r.m.s. (root–mean–square) values as follows:

$$\text{Fit} = \frac{z_{r.m.s.}}{z_{r.m.s.} + R_{r.m.s.}} \times 100$$

where

$$z_{r.m.s.} = \sqrt{\sum z_i^2 / M} = \text{z data r.m.s. value}$$

and the *residual* r.m.s. is defined to be

$$R_{r.m.s.} = \sqrt{\sum [z_i^2 - f(x_i, y_i)]^2 / M}$$

In these two equations, M is the polynomial degree.

The actual percentage of fit will depend upon the order of the algorithm, the number of x, y and z data points, and the form of the function being fitted.

Although this algorithm is intended to fit polynomial surfaces, it works well with other types of functions.

Fast Fourier

Subroutine FFT1

Purpose:

The subroutine FFT1 performs the forward and inverse Fast Fourier transform of one–dimensional real and complex data.

Syntax:

CALL FFT1 (NN, X(), ISIGN)

Where:

NN = number of data points [input]

X() = input as the array of data points and output as the FFT transform (1 dimensional array; 2 * NN rows by 1 column)

ISIGN = type of FFT transform [input]
+ 1 = forward FFT transform
– 1 = inverse FFT transform

Comments:

This subroutine computes the forward and inverse Fast Fourier Transform (FFT) of one–dimensional real or complex data using the Danielson–Lanczos or bit reversal method. This algorithm is described in Chapter 12 of Reference 1.

The companion demo program illustrates how to use subroutine FFT1 to calculate both the forward and inverse FFT of Runge's function which is defined by the equation

$$f(x) = \frac{1}{1 + 25x^2}$$

for an initial and final time input by the user.

Please note that the number of data points must be an integer power of two (2, 4, 8, 16, 32, 64, etc.) in order for this subroutine to work correctly. The demo program will check the user input to make sure that this condition is satisfied.

Because QuickBASIC does not directly support complex variables and arrays, both the input and output data are stored in real, one–dimensional arrays of length 2 * NN. Each of these arrays can visualized as consisting of sets of two elements where the first array element is the real part and the second element is the imaginary component of the data. For example, the input data is stored as follows:

X(1) = element 1 = real part of first data point, f_0
X(2) = element 2 = imaginary part of first data point
.
.
X(2N – 1) = element 1 = real part of last data point, f_{N-1}
X(2N) = element 2 = imaginary part of last data point

where N is the total number of data samples. For real input data only, the second array element of each data point is identically zero.

The output data array contains the Fourier transform at N values of frequency in what is called *wraparound order*. Like the input array, this array also contains real and imaginary parts of the transform which alternate. The output array starts with zero frequency and works up to the most positive frequency in the data. These elements are followed by the negative frequency components, from the second–most negative up to the frequency which is just below zero. For example, the order in terms of frequencies is as follows:

$$f = 0,\ f = \frac{1}{N\Delta},\ \ldots,\ f = \frac{\frac{N}{2} - 1}{N\Delta},\ f = \pm\frac{1}{2\Delta},\ \text{etc.}$$

where Δ is the sampling interval.

Fast Fourier

Subroutine PRTFFT1

Purpose:

The subroutine PRTFFT1 provides a formatted screen display of the data generated by the subroutine FFT1.

Syntax:

CALL PRTFFT1 (NN, X(), XSAVED(), ISIGN)

Where:

NN = number of data points [input]

X() = array of data points [input] (1 dimensional array; 2 * NN rows by 1 column)

XSAVED() = array of saved data points [input] (1 dimensional array; 2 * NN rows by 1 column)

ISIGN = type of FFT transform [input]

+ 1 = forward FFT transform
– 1 = inverse FFT transform

Comments:

This subroutine will display the results computed by the subroutine FFT1 for both the forward and inverse Fast Fourier transform. This information will be displayed in *wraparound order* which is described on page 121.

When ISIGN = 1 the XSAVED array contains the original data points, and when ISIGN = – 1, this array contains the transformed data points. For each type of transform, this data will be displayed by columns, with both real and imaginary components.

Fast Fourier

Subroutine FFT2

Purpose:

The subroutine FFT2 performs the forward and inverse Fast Fourier transform of two–dimensional real and complex data.

Syntax:

CALL FFT2 (NN(), X(), ISIGN)

Where:

NN(1) = number of X data points [input]

NN(2) = number of Y data points [input]

X() = input as the array of data points and output as the FFT transform (1 dimensional array; 2 * NN(1) * NN(2) rows by 1 column)

ISIGN = type of FFT transform [input]

+ 1 = forward FFT transform
– 1 = inverse FFT transform

Comments:

This subroutine computes the forward and inverse Fast Fourier Transform (FFT) of two–dimensional real or complex data using the Danielson–Lanczos or bit reversal method.

This algorithm is a QuickBASIC implementation of the N–dimensional method described in Chapter 12 of Reference 1. A discussion about the one–dimensional form of this algorithm can be found under the description for the subroutine FFT1 and provides additional information which may be helpful.

The companion demo program illustrates how to use subroutine FFT2 to calculate both the forward and inverse FFT of the two–dimensional function defined by the equation

$$f(x_j, y_k) = 4 - .02\left[(x_j - 7.5)^2 + (y_k - 7.5)^2\right]$$

for $j = 1, \ldots, M$ and $k = 1, \ldots, N$

where M and N must be integer powers of 2.

Because QuickBASIC does not directly support complex variables and arrays, both the input and output data are stored in real, one–dimensional arrays of length 2 * NX * NY, where NX and NY are the number of X and Y data points, respectively. Each of these arrays can visualized as *rows* of data which consist of sets of two elements where the first array element is the real part and the second element is the imaginary component of the data. Each row consists of 2 * NY real numbers.

For example, the input data is stored as follows:

X(1) = real part of first data point, f_0
X(2) = imaginary part of first data point
.
.
X(2NX*NY – 1) = real part of last data point, f_{N-1}
X(2NX*NY) = imaginary part of last data point

where NX and NY are the number of X and Y data points, respectively. For real input data only, the second array element of each data point is identically zero.

This type of data storage is called *wraparound order*. Please see the description for subroutine FFT1 for a discussion about wraparound order and the frequency and time domain characteristics of the input and output data.

Fast Fourier

Subroutine PRTFFT2

Purpose:

The subroutine PRTFFT2 provides a formatted display of the data generated by subroutine FFT2.

Syntax:

CALL PRTFFT2 (NN(), X(), XSAVED(), ISIGN)

Where:

NN(1)	= number of X data points [input]
NN(2)	= number of Y data points [input]
X()	= array of data points [input] (1 dimensional array; 2 * NN(1) * NN(2) rows by 1 column)
XSAVED()	= array of saved data points [input] (1 dimensional array; 2 * NN(1) * NN(2) rows by 1 column)
ISIGN	= type of FFT transform [input] + 1 = forward FFT transform – 1 = inverse FFT transform

Comments:

This subroutine will display the results computed by the subroutine FFT2 for both the forward and inverse Fast Fourier transform. This information will be displayed in *wraparound order* which is described on page 121.

When ISIGN = 1 the XSAVED array contains the original data points, and when ISIGN = – 1, this array contains the transformed data points.

This subroutine will display both the input and output data by individual rows and their corresponding columns.

For the example problem for subroutine FFT2, the following is a typical screen display for the first row of the input data and its columns, and the eight columns of the first row of the transformed real and imaginary data, respectively.

Input Data

X = 1	f (real)	f (imag)
Y = 1	2.310000D+00	0.000000D+00
Y = 2	2.550000D+00	0.000000D+00
Y = 3	2.750000D+00	0.000000D+00
Y = 4	2.910000D+00	0.000000D+00
Y = 5	3.030000D+00	0.000000D+00
Y = 6	3.110000D+00	0.000000D+00
Y = 7	3.150000D+00	0.000000D+00
Y = 8	3.150000D+00	0.000000D+00

Transformed Data

Row = 1

Column

1	2.195200D+02 + 0.000000D+00i
2	–7.570193D+00 – 7.725483D+00i
3	–4.480000D+00 – 3.200000D+00i
4	–3.949807D+00 – 1.325483D+00i
5	–3.840000D+00 + 0.000000D+00i
6	–3.949807D+00 + 1.325483D+00i
7	–4.480000D+00 + 3.200000D+00i
8	–7.570193D+00 + 7.725483D+00i

Statistics

Subroutine CHI

Purpose:

The subroutine CHI provides information about the Chi–squared distribution of statistical data.

Syntax:

CALL CHI (IFLAG, N, X, PROB)

Where:

IFLAG = type of computation [input]
1 = given x, compute probability
2 = given probability, compute x

N = number of degrees of freedom [input] (N >= 1)

X = x value [input or output] (X >= 0)

PROB = probability [input or output] (0 < PROB <= 1)

Comments:

The subroutine CHI calculates either point x from a user–defined degree of freedom n of χ^2 distribution and a given probability to the right of x or the probability p to the right of point x from a given degree of freedom n and a given point x.

The type of computation performed by this subroutine is defined by the value of IFLAG.

This subroutine also requires the QuickBASIC normal distribution subroutine NORMAL which is described on page 129 of this manual.

Statistics

Subroutine FDIST

Purpose:

The subroutine FDIST calculates information about the F distribution of statistical data.

Syntax:

CALL FDIST (IFLAG, N1, N2, X, PROB)

Where:

IFLAG = type of computation [input]
1 = given x, compute probability
2 = given probability, compute x

N1 = number of degrees of freedom [input] (N1 >= 1)

N2 = number of degrees of freedom [input] (N2 >= 1)

X = x value [input or output] (X >= 0)

PROB = probability [input or output] (0 < PROB <= 1)

Comments:

The subroutine FDIST calculates either point x from user–defined degrees of freedom n_1 and n_2 of F distribution and a given probability to the right of x, or the probability p to the right of point x from given degrees of freedom n_1 and n_2 and a given point x. The type of computation performed by this subroutine is defined by the value of IFLAG.

This subroutine also requires the normal distribution subroutine which is described on page 129.

Statistics

Subroutine NORMAL

Purpose:

The subroutine NORMAL provides information about the normal distribution of statistical data.

Syntax:

CALL NORMAL (IFLAG, X, PROB)

Where:

IFLAG = type of computation [input]
1 = given x, compute probability
2 = given probability, compute x

X = x value [input or output]

PROB = probability [input or output] (0 < PROB < 1)

Comments:

The subroutine NORMAL calculates either point x of normal distribution and a given probability to the right of x, or the probability p to the right of point x from a given point x.

The type of computation performed by this subroutine is defined by the value of IFLAG.

Please note that probability must be specified in the range

$$0 < p < 1$$

The result computed by all QuickPak Scientific statistics algorithms is accurate to four significant digits.

Statistics

Subroutine TDIST

Purpose:

The subroutine TDIST provides information about T distribution of statistical data.

Syntax:

CALL TDIST (IFLAG, N, X, PROB)

Where:

IFLAG = type of computation [input]
1 = given x, compute probability
2 = given probability, compute x

N = number of degrees of freedom [input] (N >= 1)

X = x value [input or output]

PROB = probability [input or output] (0 < PROB < 1)

Comments:

The subroutine TDIST calculates either point *x* from a user–defined degree of freedom *n* of T distribution and a given probability to the right of *x* or the probability *p* to the right of point *x* from a given degree of freedom *n* and a given point *x*.

The type of computation performed by this subroutine is defined by the value of IFLAG.

This subroutine also requires the normal and F distribution subroutines. The main subroutine handles all communications with the other support subroutines.

Functions

Function GAMMA

Purpose:

The function GAMMA recursively computes the characteristics of the Gamma function.

Syntax:

Y = GAMMA (X)

Where:

X = function argument [input]
Y = function value [output]

Comments:

The Gamma function is defined by the following integral:

$$\Gamma(z) = \int_0^{\infty} t^{z-1} e^{-t} dt$$

This subroutine calculates the value of the Gamma function for any function argument.

Functions

Function BESSEL

Purpose:

The function BESSEL recursively calculates the Bessel functions of integer order, $J_n(x)$.

Syntax:

Y = BESSEL (NORD, X)

Where:

X = function argument [input]

NORD = integer order [input]

Y = value of Bessel function $J_n(x)$ [output]

Comments:

This function computes the Bessel functions of integer order for values of x >= –3.

This function is recursive and also requires six other support functions called ABESSEL, BBESSEL, FBESSEL, GBESSEL, TBESSEL and UBESSEL. These functions are included as part of the BESSEL file on the QuickPak Scientific disk. Function BESSEL handles all communication with these other functions.

Functions

Function ERF

Purpose:

The function ERF calculates values of the error function.

Syntax:

Y = ERF (X)

Where:

X = function argument [input]
Y = function value [output]

Comments:

The Gaussian error function, erf, is defined by the following integral equation:

$$\text{erf}\ (z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2}\, dt$$

This subroutine provides a infinite series expansion for function values of this integral.

The convergence tolerance for this series is *hardwired* within the ERF function to a value of .00000001. It has the QuickBASIC name TOL and can be changed to another value if desired.

Functions

Function BETA

Purpose:

The function BETA calculates values of the Beta function.

Syntax:

Y = BETA (Z, W)

Where:

Z, W = function arguments [input]
Y = function value [output]

Comments:

The Beta function is defined as:

$$\beta\ (z,w) = \frac{\Gamma(z)\ \Gamma(w)}{\Gamma(z\ +\ w)}$$

where Γ represents the Gamma function.

This subroutine also requires the QuickPak Scientific Gamma function. All communications with the Gamma function is performed by the Beta function.

Complex Numbers

For all QuickPak Scientific complex number subroutines, each complex number is represented by a one dimensional array consisting of two elements. The first element of this array holds the real part of the complex number, and the imaginary part is in the second array element. For example, the complex number A() is represented by A(1) + A(2) *i*.

Subroutine CMPXADD

Purpose:

The subroutine CMPXADD returns the result of adding two complex numbers.

Syntax:

CALL CMPXADD (A(), B(), C())

Where:

A() = first complex number [input]

B() = second complex number [input]

C() = A() + B() [output]

Comments:

As implemented in QuickPak Scientific, complex addition and other complex number operations are much like:

C(1) = A(1) + B(1)
C(2) = A(2) + B(2)

where C(1) is the real part of the addition and C(2) is the imaginary part of this calculation.

Complex Numbers

Subroutine CMPXDIV

Purpose:

The subroutine CMPXDIV returns the result produced by dividing two complex numbers.

Syntax:

```
CALL CMPXDIV (A(), B(), C())
```

Where:

A() = first complex number [input]
B() = second complex number [input]
C() = A() / B() [output]

Subroutine CMPXMULT

Purpose:

The subroutine CMPXMULT returns the result of multiplying two complex numbers together.

Syntax:

```
CALL CMPXMULT (A(), B(), C())
```

Where:

A() = first complex number [input]
B() = second complex number [input]
C() = A() * B() [output]

Complex Numbers

Subroutine CMPXRECP

Purpose:

The subroutine CMPXRECP calculates the reciprocal of a complex number.

Syntax:

CALL CMPXRECP (A(), B())

Where:

A() = complex number [input]
B() = 1 / A [output]

Subroutine CMPXPOWR

Purpose:

The subroutine CMPXPOWR returns the result of raising a complex number to an integer power.

Syntax:

CALL CMPXPOWR (N, A(), B())

Where:

N = power [input]
A() = complex number [input]
B() = A^N [output]

Complex Numbers

Subroutine CMPXROOT

Purpose:

The subroutine CMPXROOT calculates the Nth root of a complex number.

Syntax:

CALL CMPXROOT (N, A(), B())

Where:

N = desired root [input]

A() = complex number [input]

B() = $\sqrt[N]{A}$ [output]

Subroutine CMPXSQRT

Purpose:

The subroutine CMPXSQRT calculates the square root of a complex number.

Syntax:

CALL CMPXSQRT (A(), B())

Where:

A() = complex number [input]

B() = $\sqrt{A}$ [output]

Complex Numbers

Subroutine CMPXSUB

Purpose:

The subroutine CMPXSUB returns the result of subtracting two complex numbers.

Syntax:

```
CALL CMPXSUB (A(), B(), C())
```

Where:

A() = first complex number [input]
B() = second complex number [input]
C() = A() – B() [output]

Comments:

As implemented in QuickPak Scientific, complex subtraction and other complex number operations are much like:

C(1) = A(1) – B(1)

C(2) = A(2) – B(2)

where C(1) represents the real part of the subtraction and C(2) is the imaginary part of this calculation.

Trigonometry

Function ACOS

Purpose:

The function ACOS computes the inverse cosine.

Syntax:

Y = ACOS (X)

Where:

X = function argument [input]
Y = function value [output]

Comments:

This function calculates the inverse cosine of an angle in radians. The equation used is as follows:

$$ACOS\ (X) = \frac{\pi}{2} - \sin^{-1}(X)$$

Function ACOSH

Purpose:

The function ACOSH computes the inverse hyperbolic cosine.

Syntax:

$$Y = ACOSH(X) = log_e\left[\ x + \sqrt{x^2 - 1}\ \right]$$

Where:

X = function argument [input]
Y = function value [output]

Trigonometry

Function ASIN

Purpose:

The function ASIN computes the inverse sine.

Syntax:

Y = ASIN (X)

Where:

X = function argument [input]
Y = function value [output]

Comments:

This function calculates the inverse sine of an angle in radians. The equation used is as follows:

$$\text{ASIN}(X) = \tan^{-1}\left[\frac{x}{\sqrt{1 - x^2}}\right]$$

Function ASINH

Purpose:

The function ASINH computes the inverse hyperbolic sine.

Syntax:

$$Y = \text{ASINH}(X) = \log_e\left[x + \sqrt{x^2 + 1}\right]$$

Where:

X = function argument [input]
Y = function value [output]

Trigonometry

Function ATAN3

Purpose:

The function ATAN3 computes the four–quadrant inverse tangent.

Syntax:

Y = ATAN3 (A, B)

Where:

A	=	sine of angle θ [input]
B	=	cosine of angle θ [input]
Y	=	function value [output]

Comments:

This is a four quadrant inverse tangent function. It examines the sign of the sine and cosine of an angle and returns the value of the angle (in radians) such that $0 \leq \theta \leq 2\pi$.

The pseudocode for this algorithm is as follows:

IF (sin θ < ε) THEN

$$\text{ATAN3} = (1 - \text{sign}(\sin\theta)) * \frac{\pi}{2}$$

ELSE IF (cos θ < ε) THEN

$$\text{ATAN3} = (2 - \text{sign}(\sin\theta)) * \frac{\pi}{2}$$

ELSE

$$\text{ATAN3} = (2 - \text{sign}(\sin\theta)) * \frac{\pi}{2} + \text{sign}(\sin\theta) * \text{sign}(\cos\theta) * \left|\tan^{-1}(\sin\theta / \cos\theta)\right| - \frac{\pi}{2}$$

where ε = .00000001 and π = 3.14159265.

Trigonometry

Function ATANH

Purpose:

The function ATANH computes the inverse hyperbolic tangent.

Syntax:

$$Y = \text{ATANH}(X) = \frac{1}{2} \log_e \left[\frac{1 + x}{1 - x} \right]$$

Where:

X = function argument [input]
Y = function value [output]

Function COSH

Purpose:

The function COSH computes the hyperbolic cosine.

Syntax:

$$Y = \text{COSH}(X) = \frac{1}{2} \left[e^{x} + e^{-x} \right]$$

Where:

X = function argument [input]
Y = function value [output]

Trigonometry

Function SINH

Purpose:

The function SINH computes the hyperbolic sine.

Syntax:

$$Y = SINH\ (X) = \frac{1}{2}\left[e^{x} - e^{-x} \right]$$

Where:

X = function argument [input]
Y = function value [output]

Function TANH

Purpose:

The function TANH computes the hyperbolic tangent.

Syntax:

$$Y = TANH\ (X) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Where:

X = function argument [input]
Y = function value [output]

Matrices

Subroutine INVERSE

Purpose:

The subroutine INVERSE solves for the inverse of an N by N square real matrix.

Syntax:

CALL INVERSE (N, A(), AINV(), IER)

Where:

N	= number of equations [input]
A()	= matrix of coefficients [input] (2 dimensional array; N rows by N columns)
AINV()	= inverse of the matrix [A] [output] (2 dimensional array; N rows by N columns)
IER	= error flag [output] (0 = no error, 1 = singular matrix)

Comments:

The demo program asks the user to input the elements of the N by N matrix [A] by *rows*.

The subroutine INVERSE will check the matrix [A] for singularity. If the matrix is singular or cannot be factored, the error flag IER will be set to 1. Otherwise this flag is set to the number 0.

Please note that the original matrix of coefficients, [A], is modified by the software.

This algorithm is described in Reference 1, Chapter 2.

Matrices

Subroutine DETERMIN

Purpose:

The subroutine DETERMIN solves for the determinant of an N by N square matrix.

Syntax:

CALL DETERMIN (N, A(), DM, IER)

Where:

N	= number of equations [input]
A()	= matrix of coefficients [input] (2 dimensional array; N rows by N columns)
DM	= determinant of matrix [A] [input]
IER	= error flag [output] (0 = no error, 1 = singular matrix)

Comments:

The demo program requests the user to input the elements of matrix [A] by *rows*.

The subroutine DETERMIN will check the matrix [A] for singularity. If matrix A() is singular or cannot be factored, the error flag IER will be set to 1. Otherwise this flag is set to the number 0.

Please note that the original matrix of coefficients, [A], is modified by the software.

Please consult Chapter 2 of Reference 1 for a description of this algorithm.

Matrices

Subroutine EIGEN1

Purpose:

The subroutine EIGEN1 computes the real eigenvalues and eigenvectors of a real square matrix using the power method of solution.

Syntax:

```
CALL EIGEN1 (A(), N, VZERO(), MMAX, MFREQ, EPS,
                          XLAMBDA(), U(), IFLAG)
```

Where:

A()	= matrix [input] (2 dimensional array; N rows by N columns)
N	= number of rows and columns [input]
VZERO()	= vector of initial eigenvalues [input] (1 dimensional array; N rows by 1 column)
MMAX	= maximum number of iterations [input]
MFREQ	= iterations between updates [input]
EPS	= convergence tolerance [input]
XLAMBDA()	= vector of eigenvalues [output] (1 dimensional array; N rows by 1 column)
U()	= matrix of eigenvectors [output] (2 dimensional array; N rows by N columns)
IFLAG	= error flag [output] 0 = no error + = convergence error)

Comments:

An eigenvector is the nonzero vector $\bar{X}$ which satisfies the following matrix/vector/scalar relationship:

$$\left[A \right] \bar{X} = \lambda \bar{X}$$

The scalar quantity λ in this equation is called the eigenvalue (proper or characteristic value). The eigenvectors are also called proper or characteristic vectors.

The companion demo program for this subroutine will prompt the user for the matrix by *rows*. It will also ask for a vector of initial guesses for the eigenvalues. All but one of these guesses could be input as 0 although a value of 1 is recommended by the software. The program will also ask you for the maximum number of iterations; a value of 50 is recommended. The number of iterations between updates and a convergence criteria are also requested. The update value should be 1 or larger, and a criteria of 1D–8 is advised.

The eigenvalues will be displayed as a column vector and the companion eigenvectors will be displayed in a two–dimensional array. Each *column* of this array represents the eigenvector which corresponds to the same numbered *row* of the eigenvalue vector.

If this algorithm is successful, the error flag IFLAG will be returned with the value 0. However, if there is a convergence problem, this flag will be set to the actual number of iterations performed. This algorithm works best with real symmetric matrices, which always have real eigenvalues and eigenvectors.

The power method for calculating the real eigenvalues and eigenvectors of a real matrix is described in Chapter 4 of Reference 13.

Matrices

Subroutine EIGEN2

Purpose:

The subroutine EIGEN2 computes the real and complex eigenvalues of a real square matrix using the QR iterative method of solution.

Syntax:

CALL EIGEN2 (N, A(), ITMAX, XREAL(), XIMAG(), IER)

Where:

N	= size of matrix A() [input]
A()	= matrix [input] (2 dimensional array; N + 1 rows by N + 1 columns)
ITMAX	= maximum number of iterations [input]
XREAL()	= real components of eigenvalues [output] (1 dimensional array; N rows by 1 column)
XIMAG()	= imaginary components of eigenvalues [output] (1 dimensional array; N rows by 1 column)
IER	= error flag [output] (0 = no error, 1 = convergence error)

Comments:

This subroutine uses the QR iteration method to compute the real and complex eigenvalues of a real, square matrix.

Although matrix [A] is of size N, it must be dimensioned N + 1, N + 1 in the program which calls this subroutine.

A value of 50 for the maximum number of iterations should be adequate for most problems.

Matrices

Subroutine IMATRIX

Purpose:

Subroutine IMATRIX calculates an identity matrix whose size is specified by the user.

Syntax:

CALL IMATRIX(N, A())

Where:

N	= dimension of matrix [input]
A()	= identity matrix [output] (2 dimensional array; N rows by N columns)

Comments:

This subroutine returns an identity matrix [A] with N rows and N columns. This is a square matrix with all diagonal elements equal to 1, and all other elements equal to 0.

For example, if N = 3 we have:

$$\left[A \right] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Matrices

Subroutine MATADD

Purpose:

The subroutine MATADD performs matrix addition.

Syntax:

```
CALL MATADD (A(), B(), C(), N, M)
```

Where:

A()	= first matrix [input] (2 dimensional array; N rows by M columns)
B()	= second matrix [input] (2 dimensional array; N rows by M columns)
C()	= matrix addition of A() and B() [output] (2 dimensional array: N rows by M columns)
N	= number of rows of each matrix [input]
M	= number of columns of each matrix [input]

Comments:

Matrix addition consists of the following calculations:

```
FOR I = 1 TO N
    FOR J = 1 TO M
    C(I, J) = A(I, J) + B(I, J)
    NEXT J
NEXT I
```

Matrices

Subroutine MATSUB

Purpose:

The subroutine MATSUB performs matrix subtraction.

Syntax:

```
CALL MATSUB (A(), B(), C(), N, M)
```

Where:

N	= number of rows in each matrix [input]
M	= number of columns in each matrix [input]
A()	= matrix [A] [input] (2 dimensional array; N rows by M columns)
B()	= matrix [B] [input] (2 dimensional array; N rows by M columns)
C()	= matrix [C] = [A] + [B] [output] (2 dimensional array; N rows by M columns)

Comments:

Matrix subtraction consists of the following calculations:

```
FOR I = 1 TO N
    FOR J = 1 TO M
    C(I, J) = A(I, J) – B(I, J)
    NEXT J
NEXT I
```

Matrices

Subroutine MATXMAT

Purpose:

The subroutine MATXMAT performs matrix multiplication.

Syntax:

CALL MATXMAT (A(), B(), C(), L, M, N)

Where:

A() = matrix [A] [input]
(2 dimensional array; L rows by M columns)

B() = matrix [B] [input]
(2 dimensional array; M rows by N columns)

C() = matrix [C] = [A] * [B] [output]
(2 dimensional array; L rows by N columns)

Comments:

This subroutine performs a matrix multiplication such that:

$$\left[C \right] = \left[A \right] \left[B \right]$$

This can also be written as

$$C_{ij} = \sum_{i=1}^{L} \sum_{j=1}^{N} \sum_{k=1}^{M} A_{ik} B_{kj}$$

This algorithm assumes that the number of columns of the matrix [A] is equal to the number of rows of matrix [B]. The result is a matrix with L rows and N columns.

Matrices

Subroutine MATXVEC

Purpose:

The subroutine MATXVEC multiplies a column vector by a real matrix.

Syntax:

CALL MATXVEC (A(), B(), C(), L, M)

Where:

A() = matrix [A] [input]
(2 dimensional array; L rows by M columns)

B() = vector {B} [input]
(1 dimensional array; M rows by 1 column)

C() = vector {C} = [A] * {B} [output]
(1 dimensional array; L rows by 1 column)

Comments:

This subroutine performs a matrix and column vector multiplication such that:

$$\bar{C} = \left[A \right] \bar{B}$$

This can also be written as the following nested summation:

$$C_i = \sum_{i=1}^{L} \sum_{j=1}^{M} A_{ij} B_j$$

This operation is valid only if the number of columns of matrix [A] is equal to the number of rows of vector $\bar{B}$.

Matrices

Subroutine RANK

Purpose:

The subroutine RANK computes the rank of a real matrix.

Syntax:

CALL RANK (A(), N, M, K)

Where:

A()	= matrix [input] (2 dimensional array; N rows by M columns)
N	= number of rows of matrix [A] [input]
M	= number of columns of matrix [A] [input]
K	= rank of matrix [A] [output]

Comments:

The rank of any matrix is the order of the highest order *non-vanishing* determinant within the matrix.

The rank of a matrix is a positive integer number.

Furthermore, the rank of any matrix is unchanged if any multiple of the matrix elements of one row or column is added to the corresponding elements of another row or column, respectively.

Matrices

Subroutine TRACE

Purpose:

The subroutine TRACE calculates the trace of a square, real matrix.

Syntax:

CALL TRACE (N, A(), T)

Where:

N	= dimension of matrix [input]
A()	= matrix [A] [input] (2 dimensional array; N rows by N columns)
T	= trace of matrix [A] [output]

Comments:

This subroutine calculates the trace of an N by N square matrix. The trace of a matrix is given by:

$$\text{TRACE} = \sum_{i=1}^{N} A_{ii}$$

The trace of an N by N matrix is equal to the sum of all its diagonal elements. For an N by N matrix, this can be written as

$$\text{TRACE}\left[A \right] = A_{11} + A_{22} + A_{33} + \ldots + A_{NN}$$

Matrices

Subroutine TRANSPOSE

Purpose:

The subroutine TRANSPOSE calculates the transpose of a real matrix.

Syntax:

CALL TRANSPOSE (A(), B(), M, N)

Where:

M	= number of rows in matrix [A] [input]
N	= number of columns in matrix [A] [input]
A()	= matrix [A] [input] (2 dimensional array; M rows by N columns)
B()	= transpose of matrix [A] [output] (2 dimensional array; N rows by M columns)

Comments:

This subroutine calculates the transpose of a matrix which has M rows and N columns. The transpose is given by:

$$\left[B \right] = \left[A \right]^T \Rightarrow \left[B \right]_{ij} = \left[A \right]_{ji}$$

When computing the matrix transpose, the rows of matrix [B] take on the values of the columns of matrix [A].

Vectors

Subroutine UVECTOR

Purpose:

Subroutine UVECTOR calculates the unit vector of 3–component, real column vector.

Syntax:

CALL UVECTOR (A(), B())

Where:

A() = column vector [input]
(1 dimensional array; 3 rows by 1 column)

B() = unit vector [output]
(1 dimensional array; 3 rows by 1 column)

Comments:

This subroutine returns the unit vector of a 3 by 1 column vector. The result is also a 3 rows by 1 column vector with scalar magnitude equal to 1.

A unit vector can be computed from the following expression:

$$\hat{B} = \frac{\bar{A}}{|A|} = \begin{Bmatrix} A_1 / A \\ A_2 / A \\ A_3 / A \end{Bmatrix}$$

where A_1, A_2, and A_3 are the three components of the original vector $\bar{A}$ and $A = \sqrt{A_1^2 + A_2^2 + A_3^2}$ is the vector magnitude.

Vectors

Subroutine VCROSS

Purpose:

The subroutine VCROSS determines the cross product of two vectors.

Syntax:

CALL VCROSS (A(), B(), C())

Where:

A() = column vector [input]
(1 dimensional array; 3 rows by 1 column)

B() = column vector [input]
(1 dimensional array; 3 rows by 1 column)

C() = column vector [output]
(1 dimensional array; 3 rows by 1 column)

Comments:

This subroutine calculates the cross product of two 3 rows by 1 column vectors such that:

$$\bar{C} = \bar{A} \times \bar{B} = \begin{Bmatrix} A_2B_3 - A_3B_2 \\ A_3B_1 - A_1B_3 \\ A_1B_2 - A_2B_1 \end{Bmatrix}$$

where A_1, A_2, A_3, B_1, B_2, and B_3 are the components of the $\bar{A}$ and $\bar{B}$ vectors, respectively.

The result is also a column vector with 3 rows and 1 column.

Vectors

Subroutine VDOT

Purpose:

The subroutine VCROSS determines the dot product of two vectors.

Syntax:

CALL VDOT (N, A(), B(), C)

Where:

N	= number of rows in vectors {A} and {B} [input]
A()	= column vector {A} [input] (1 dimensional array; N rows by 1 column)
B()	= column vector {B} [input] (1 dimensional array; N rows by 1 column)
C	= dot product of {A} and {B} [output]

Comments:

This subroutine calculates the vector dot product of two N by 1 column vectors such that:

$$C = \bar{A} \bullet \bar{B}$$

The vector dot product can also be written as

$$C = A_1B_1 + A_2B_2 + \ldots + A_NB_N$$

where A_1, A_2, . . . , A_N, and B_1, B_2, . . . , B_N are the N components of the $\bar{A}$ and $\bar{B}$ vectors, respectively. The result of the dot product is a scalar.

Vectors

Subroutine VECADD

Purpose:

The subroutine VECADD performs vector addition.

Syntax:

CALL VECADD (A(), B(), C(), N)

Where:

N	= number of columns in each vector [input]
A()	= first vector {A} [input] (1 dimensional array; N rows by 1 column)
B()	= second vector {B} [input] (1 dimensional array; N rows by 1 column)
C()	= vector sum of {A} and {B} [output]

Comments:

Vector addition consists of the following straight–forward operation:

$$\overline{C} = \overline{A} + \overline{B}$$

which on an element by element basis can also be expressed as

$$C_i = A_i + B_i \qquad i = 1, 2, \ldots, N$$

where N is the number of rows in each vector.

Vectors

Subroutine VECMAG

Purpose:

The subroutine VECMAG calculates the scalar magnitude of a real three–component vector.

Syntax:

CALL VECMAG (A(), A)

Where:

A() = column vector {A} [input]
(1 dimensional array; 3 rows by 1 column)

A = magnitude of A [output]

Comments:

This subroutine calculates the scalar magnitude of a 3 rows by 1 column vector such that:

$$A = | \bar{A} |$$

The scalar magnitude of a vector can also be written as

$$A = \sqrt{A_1^2 + A_2^2 + A_3^2}$$

where A_1, A_2, and A_3 are the three components of the original $\bar{A}$ vector.

The magnitude of a vector is a positive scalar number.

Vectors

Subroutine VECSTP

Purpose:

The subroutine VECSTP calculates the scalar triple product.

Syntax:

CALL VECSTP (A(), B(), C(), VSTP)

Where:

A()	= first column vector {A} [input] (1 dimensional array; 3 rows by 1 column)
B()	= second column vector {B} [input] (1 dimensional array; 3 rows by 1 column)
C()	= third column vector {C} [input] (1 dimensional array; 3 rows by 1 column)
VSTP	= scalar triple product [output]

Comments:

The scalar triple product of any three, 3–element column vectors is defined by the following combination dot and cross product calculation:

$$VSTP = \bar{A} \bullet \left[\bar{B} \times \bar{C} \right]$$

The result of the scalar triple product operation is a scalar number.

Vectors

Subroutine VECSUB

Purpose:

The subroutine VECSUB performs vector subtraction.

Syntax:

CALL VECSUB (A(), B(), C(), N)

Where:

N	= number of columns in each vector [input]
A()	= first column vector {A} [input] (1 dimensional array; N rows by 1 column)
B()	= second column vector {B} [input] (1 dimensional array; N rows by 1 column)
C()	= vector equal to {A} – {B} [output]

Comments:

Vector subtraction consists of the following straight–forward operation:

$$\bar{C} = \bar{A} - \bar{B}$$

which on an element by element basis can also be expressed as

$$C_i = A_i - B_i \qquad i = 1, 2, \ldots, N$$

where N is the number of rows in each vector.

Vectors

Subroutine VECVTP

Purpose:

The subroutine VECVTP calculates the vector triple product.

Syntax:

CALL VECVTP (A(), B(), C(), D())

Where:

A() = first column vector {A} [input]
(1 dimensional array; 3 rows by 1 column)

B() = second column vector {B} [input]
(1 dimensional array; 3 rows by 1 column)

C() = third column vector {C} [input]
(1 dimensional array; 3 rows by 1 column)

D() = vector triple product [output]
(1 dimensional array; 3 rows by 1 column)

Comments:

The vector triple product of any three, 3–element column vectors is defined by the following cross product calculations:

$$\bar{D} = \bar{A} \times \left[\bar{B} \times \bar{C} \right]$$

The result of the vector triple product calculation is also a column vector with 3 rows and 1 column.

Utility

Subroutine CDATE

Purpose:

The subroutine CDATE calculates the calendar date from the Julian date.

Syntax:

CALL CDATE (XJDATE, XMONTH, DAY, YEAR)

Where:

XJDATE = Julian date [input]
XMONTH = calendar month [output] (1<= XMONTH <=12)
DAY = calendar day [output] (1 <= DAY <= 31)
YEAR = calendar year [output]

Comments:

This subroutine will compute the calendar or Gregorian date for any valid Julian date. The Julian date must be a positive number. Please note that B.C. years are negative and A.D. years are positive. For example, 1 B.C. is the year 0, year –1 = 2 B.C. and so forth.

The Gregorian calendar was introduced by Pope Gregory XII in the year 1582 to replace the Julian calendar. In this system, every year that is evenly divisible by the number four is a leap year with the exception of centurial years. Centurial years are leap years only if they are evenly divisible by 400.

Depending on the actual Julian date, the calendar day may also have a fractional part.

Utility

Subroutine JULIAN

Purpose:

The subroutine JULIAN computes the Julian date for any valid calendar date.

Syntax:

CALL JULIAN (XMONTH, DAY, YEAR, XJD, VDATE$)

Where:

XMONTH = calendar month [input] (1<= XMONTH <=12)
DAY = calendar day [input] (1 <= DAY <= 31)
YEAR = calendar year [input]
XJD = Julian date [output]
VDATE$ = valid date flag [output] (Y = yes, N = no)

Comments:

The Julian date is a continuous count of days and fractions thereof since Greenwich noon on 1 January 4713 B.C. Note that the Julian date begins at noon, so at 0 hour civil time, the fractional part of the Julian date is .5.

It is important to pass this subroutine all the digits of the calendar year. For example, the calendar year 1991 must be passed as *1991* and not *91*. Note also that B.C. years are negative and A.D. years are positive. For example, 1 B.C. is the year 0, year –1 = 2 B.C. and so forth.

The JULIAN subroutine will check the calendar dates for validity and return VDATE$ = N for any invalid date. In the Gregorian calendar reform, the day following October 4, 1582 is October 15, 1582. Therefore, calendar dates between October 5 and 14, 1582 are invalid. If an invalid date is encountered, the demo program DEMODATE will display the following message:

This date does not exist !!

and the user will be asked to input another date.

The companion demo program for these subroutines also illustrates how to use the QuickPak Scientific date algorithms to compute other types of date information. These options are as follows:

- the number of days between two dates
- the day of the week from a calendar date
- the day of the year from a calendar date
- the calendar date from the day of the year

For example, the number of days between any two calendar dates is simply the difference between their Julian dates. Other date calculations are variations of the basic CDATE and JULIAN algorithms. For example, the day of the week, DOW, can be found with the following calculation:

```
DOW = FIX(7 * (XJD / 7# - FIX(XJD / 7#)) + .5#)
```

where

```
XJD = XJD0 + 1
```

and

XJD0 is the Julian date at 0 hours Universal Time.

Utility

Subroutine XYPLOT

Purpose:

The subroutine XYPLOT displays a simple X–Y plot of data input by the user. This subroutine supports the CGA, EGA, VGA, and Hercules graphics mode of the IBM–PC and true compatible computers.

Syntax:

```
CALL XYPLOT (MODE, NPTS, X(), Y(), XAXIS$,
                                   YAXIS$, TITLE$)
```

Where:

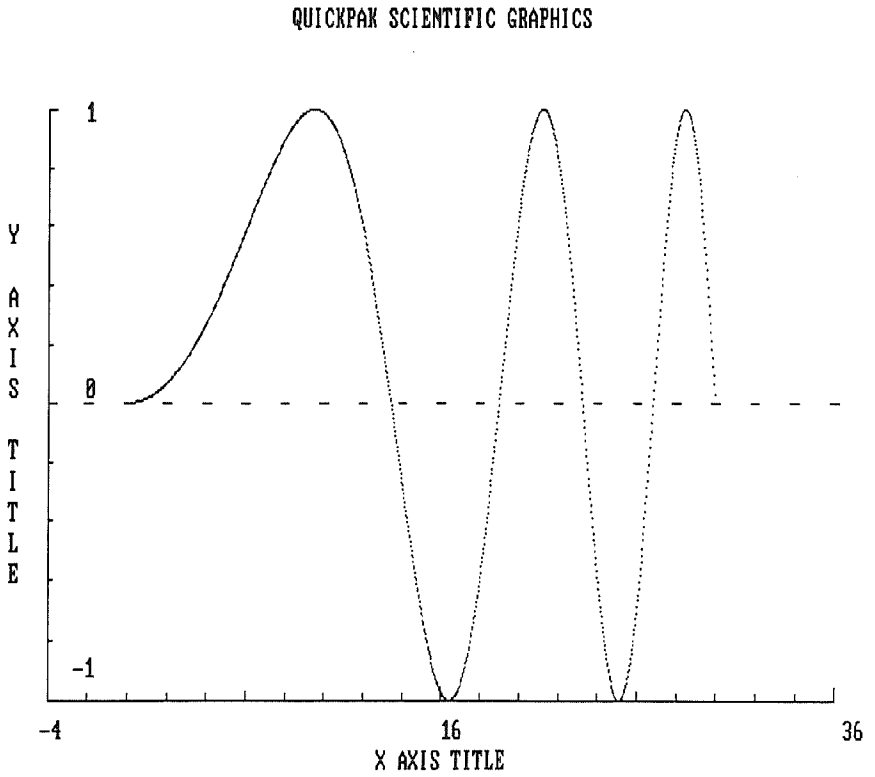
MODE	= graphics mode [input] (1 = CGA, 2 = EGA, 3 = VGA, 4 = Hercules)
NPTS	= number of X and Y data points [input]
X()	= array of X data points [input] (1 dimensional array; NPTS rows by 1 column)
Y()	= array of Y data points [input] (1 dimensional array; NPTS rows by 1 column)
XAXIS$	= X–axis title [input]
YAXIS$	= Y–axis title [input]
TITLE$	= plot title [input]

Comments:

This subroutine automatically sorts, scales, and plots a simple graph of X and Y data passed to it in one–dimensional QuickBASIC arrays.

The software is *hardwired* to use 20 subdivisions along the horizontal X axis and 10 subdivisions along the vertical Y axis. If the Y data array bounds zero, the software also plots a horizontal dashed line at Y = 0. The subroutine provides the user with the capability to pass ASCII names for both axes, and a title for the graphics plot.

The following is a typical graphics display generated by subroutine XYPLOT and its demo program. It illustrates a plot of the simple function $y = \sin x^2$.



Utility

Subroutine SCALE

Purpose:

The subroutine SCALE is a support routine for the subroutine XYPLOT. This subroutine takes unscaled minimum and maximum scale values for a graphics axis and returns new and easy to read scale values.

Syntax:

```
CALL SCALE (XMIN, XMAX, N, XMINP, XMAXP)
```

Where:

XMIN	= minimum unscaled axis value [input]
YMIN	= maximum unscaled axis value [input]
N	= number of axis subdivisions [input] (N >= 1)
XMINP	= minimum scaled axis value [output]
XMAXP	= maximum scaled axis value [output]

Comments:

This subroutine is based on ACM Algorithm # 463 which rescales axes data into a more readable form. According to the algorithm description, a readable linear scale is defined to be a scale with an interval size which is a product of an integer power of 1, 2, 5 or 10, and scale values which are integer multiples of the interval size. The interval size is directly related to the subroutine parameter N which specifies the number of axis subdivisions.

This subroutine is called by XYPLOT to rescale both the X and Y axes of the graphics plot.

Utility

Function ROUND

Purpose:

The function ROUND rounds a real, double–precision number to a user–specified number of decimal places.

Syntax:

```
Y = ROUND (X, N)
```

Where:

X = real number [input]
N = number of decimal places [input]
Y = X rounded to N decimal places [output]

Comments:

This QuickBASIC function will round a real, double–precision number to an integer number of decimal places specified by the user. The QuickBASIC source code for this calculation is:

```
A = 10# ^ N

ROUND = INT(X / A + .5#) * A
```

where X is the real number and N is the number of decimal places requested, respectively.

Bibliography and References

1. Press, W. H., Flannery, B. P., Teukolsly, S.A. and Vetterling, W. T. (1986). *Numerical Recipes*, Cambridge University Press.

2. Burden, R. L. and Faires, J. D. (1981). *Numerical Analysis*, PWS–Kent Publishing Company.

3. Chapra, S. C. and Canale, R. P. (1985). *Numerical Methods for Engineers*, McGraw–Hill Book Company.

4. Danby, J. M. A. (1988). *Fundamentals of Celestial Mechanics*, Willmann–Bell, Inc.

5. Brent, R. P. (1973). *Algorithms for Minimization without Derivatives*, Prentice–Hall.

6. Arora, J. S. (1989). *Introduction to Optimum Design*, McGraw–Hill Book Company.

7. Ruckdeschel, F. R. (1981). *BASIC Scientific Subroutines, Volumes I and II*, BYTE Publications, Inc.

8. Hull, D. G. and Williamson, W. E. (1979). "Numerical Derivatives for Parameter Optimization". *AIAA Journal of Guidance and Control*, Vol. 2, No. 2, March–April 1979, pp. 158–160.

9. Balch, S. J. and Thompson, G. T. (1989). "An Efficient Algorithm for Polynomial Surface Fitting". *Computers and Geosciences*, Vol. 15, No. 1, pp. 107–119.

10. McNamee, J. M. (1981). "A Program to Integrate a Function Tabulated at Unequal Intervals". *International Journal for Numerical Methods in Engineering*, Vol. 17, pp. 271–279.

BIBLIOGRAPHY and REFERENCES

11. Sewell, G. (1988). *The Numerical Solution of Ordinary and Partial Differential Equations*, Academic Press, Inc.

12. Kahaner, D., Moler, C., and Nash, S. (1989). *Numerical Methods and Software*, Prentice Hall, Inc.

13. Carnahan, B., Luther, H. A., and Wilkes, J. O. (1969). *Applied Numerical Methods*, John Wiley and Sons, Inc.

14. Shampine, L. F., and Allen, Jr., R. C. (1973). *Numerical Computing*, W. B. Saunders Company.

15. Danby, J. M. A. (1985). *Computing Applications to Differential Equations*, Reston Publishing Company, Inc.

16. Gill, P. E., Murray, W., and Wright, M. H. (1981). *Practical Optimization*, Academic Press.

17. Cuthbert, T. R. (1987). *Optimization Using Personal Computers*, John Wiley and Sons, Inc.

18. Nash, J. C. (1990). *Compact Numerical Methods for Computers*, Adam Hilger.

19. Nakamura, S. (1991). *Applied Numerical Methods With Software*, Prentice–Hall, Inc.

20. Abramowitz, M. and Stegun, I. (1965). *Handbook of Mathematical Functions*, Dover Publications, Inc.

21. Maron, M. J., and Lopez, R. J. (1991). *Numerical Analysis: A Practical Approach*, Wadsworth, Inc.

22. Atkinson, L. V., Harley, P. J., and Hudson, J. D. (1989). *Numerical Methods With FORTRAN 77*, Addison–Wesley Publishing Company, Inc.

Appendix A

ORDER REDUCTION OF DIFFERENTIAL EQUATIONS

In this appendix we demonstrate how to reduce one or more differential equations of any order to first–order equations which can be solved with the QuickPak Scientific algorithms.

Let's define a single *Nth–order* differential equation as

$$x^{(N)} = f\left[t,\ x,\ x',\ x'',\ \ldots,\ x^{(N-1)}\right]$$

with the initial conditions given by

$$x(t_0) = x_0,\quad x'(t_0) = x_0',\quad \ldots,\quad x^{(N-1)}(t_0) = x_0^{(N-1)}$$

To reduce this system to a first–order differential equation, we introduce N *state variables* $y_1, \ldots, y_N$ such that

$y_1 = x$	$\Rightarrow$	$y_1' = x' = y_2$	$y_1(t_0) = x_0$
$y_2 = x'$	$\Rightarrow$	$y_2' = x'' = y_3$	$y_2(t_0) = x_0'$
$y_3 = x''$	$\Rightarrow$	$y_3' = x''' = y_4$	$y_3(t_0) = x_0''$
.		.	.
.		.	.
$y_{N-1} = x^{(N-2)}$	$\Rightarrow$	$y_{N-1}' = x^{(N-1)} = y_N$	$y_{N-1}(t0) = x_0^{(N-1)}$
$y_N = x^{(N-1)}$	$\Rightarrow$	$y_N' = x^N = y_N$	$y_N(t_0) = x_0^{(N-1)}$

Column one of this arrangement denotes the state variable substitution, column two shows the equivalent system of first–order differential equations, and column three represents the new form of the initial conditions.

To illustrate the use of this method, we will reduce the system of second–order differential equations of unperturbed orbital motion defined by:

$$\ddot{x} = \frac{d^2x}{dt^2} = \frac{-\mu x}{R^3}, \qquad \ddot{y} = \frac{d^2y}{dt^2} = \frac{-\mu y}{R^3}, \qquad \ddot{z} = \frac{d^2z}{dt^2} = \frac{-\mu z}{R^3}$$

where x, y, and z represent the position vector coordinates of the spacecraft, and R is the scalar position magnitude. The initial conditions are:

$$x(t_0) = x_0, \qquad y(t_0) = y_0, \qquad z(t_0) = z_0 \qquad \text{(position)}$$

and

$$\dot{x}(t_0) = \dot{x}_0, \qquad \dot{y}(t_0) = \dot{y}_0, \qquad \dot{z}(t_0) = \dot{z}_0 \qquad \text{(velocity)}$$

Since these are second–order differential equations, we expect the reduced system to consist of two first–order equations for each coordinate.

With the state variable substitutions

$$y_1 = x, \quad y_2 = y, \quad y_3 = z, \quad y_4 = \dot{x}, \quad y_5 = \dot{y}, \quad y_6 = \dot{z},$$

we have the following system of first–order equations:

$$\dot{y}_1 = \dot{x} \qquad \dot{y}_2 = \dot{y} \qquad \dot{y}_3 = \dot{z}$$

$$\dot{y}_4 = \ddot{x} \qquad \dot{y}_5 = \ddot{y} \qquad \dot{y}_3 = \ddot{z}$$

and the equivalent initial conditions:

$$y_1(t_0) = x(t_0) \qquad y_2(t_0) = y(t_0) \qquad y_3(t_0) = z(t_0)$$

$$y_4(t_0) = \dot{x}(t_0) \qquad y_5(t_0) = \dot{y}(t_0) \qquad y_6(t_0) = \dot{z}(t_0)$$

Now let's examine what the QuickBASIC code for this new system of differential equations might look like. If we make the following QuickBASIC array variable definitions:

X(1) = x component of the satellite's position vector

X(2) = y component of the satellite's position vector

X(3) = z component of the satellite's position vector

X(4) = x component of the satellite's velocity vector

X(5) = y component of the satellite's velocity vector

X(6) = z component of the satellite's velocity vector

and define the 6 component array Y() to be the integration vector, we will have the following subroutine which is compatible with the QuickPak Scientific Runge–Kutta–Fehlberg algorithms.

```
SUB DERIVATIVE (T, X(), Y()) STATIC
    ' Equations of motion subroutine
    Y(1) = X(4)
    Y(2) = X(5)
    Y(3) = X(6)
    R = SQR(X(1) * X(1) + X(2) * X(2) + X(3) * X(3))
    R3 = R * R * R
    Y(4) = - XMU * X(1) / R3
    Y(5) = - XMU * X(2) / R3
    Y(6) = - XMU * X(3) / R3
END SUB
```

XMU is the gravitational constant and might be passed to this subroutine with a CONST statement, for example.

Appendix B

PRACTICAL APPLICATIONS

This appendix is a discussion about three stand–alone computer programs which use QuickPak Scientific algorithms to solve practical and interesting problems from the field of Celestial Mechanics.

Optimal Impulsive Orbital Transfer

The coplanar orbital transfer between two circular orbits was first discovered by Walter Hohmann in 1925. The transfer consists of a velocity impulse on an initial circular orbit, in the direction of motion, which propels the space vehicle into an elliptical transfer orbit. At a transfer angle of 180 degrees from the first impulse, a second velocity impulse or delta–V, also in the direction of motion, places the vehicle into a final circular orbit at the desired altitude.

The impulsive velocity assumption means that the velocity, but not the position, of the vehicle is changed instantaneously. This implies that there is a discontinuity in the speed of the spacecraft.

With a QuickBASIC program called IOTA and the QuickPak Scientific subroutine REALROOT, we will extend the classic Hohmann transfer for the case of non–coplanar orbital transfer. The user can interactively specify a plane change between the initial and final orbits, and this program will compute the "best" way to partition this plane change between the two impulses, and the total delta–V required for the orbit transfer. Best in this case means the minimum total delta–V for the orbit transfer.

The analysis presented here in also described in Chapter 3 of the classic text, *Methods of Astrodynamics* by P. R. Escobal.

For convenience, we will define three normalized radii which are functions only of the size of the two circular orbits:

$$H_1 = \sqrt{2 \frac{r_f}{r_i + r_f}} \qquad (1)$$

$$H_2 = \sqrt{\frac{r_i}{r_f}} \qquad (2)$$

$$H_3 = \sqrt{2 \frac{r_i}{r_i + r_f}} \qquad (3)$$

where

r_i = radius of the initial circular orbit

r_f = radius of the final circular orbit

The delta–velocity increment or impulse required for the first impulse is given by:

$$\Delta V_1 = V_{1c} \sqrt{1 + H_1^2 - 2H_1 \cos \theta_1} \qquad (4)$$

and the delta–velocity increment required for the second impulse is given by:

$$\Delta V_2 = V_{1c} \sqrt{H_2^2 + H_2^2 H_3^2 - 2H_2^2 H_3 \cos \theta_2} \qquad (5)$$

where θ_1 and θ_2 are the plane change angles at the first and second impulses, respectively.

The total delta–velocity required for the orbital transfer is:

$$\Delta V_t = \Delta V_1 + \Delta V_2 \qquad (6)$$

where

θ_1 = plane change angle at the initial orbit

θ_2 = plane change angle at the final orbit

θ_t = total plane change angle = $\theta_1 + \theta_2$

$V_{lc} = \sqrt{\frac{\mu}{r_i}}$ = *local circular velocity*

μ = gravitational constant

The local circular velocity is simply the orbiting speed of a spacecraft in the initial circular orbit.

We first need to derive the partial derivative of equation (6) with respect to either the first or second plane change angle.

The partial derivative of the normalized total delta velocity, $\Delta V = \Delta V_t / V_{lc}$ with respect to θ_1 is given by:

$$\frac{\partial \Delta V}{\partial \theta_1} = \frac{H_1 \sin \theta_1}{\sqrt{1 + H_1^2 - 2H_1 \cos \theta_1}} + \frac{H_2^2 H_3 \sin \theta_2}{\sqrt{H_2^2 + H_1^2 H_3^2 - 2H_2^2 H_3 \cos \theta_2}} \qquad (7)$$

If we use subroutine REALROOT to solve this equation for the positive value of θ_1 which makes this expression equal 0, we will have found the minimum ΔV required for the orbit transfer. We also know that the root of this equation is somewhere between 0 and θ_t degrees.

Once the value for θ_1 has been determined, the second plane change angle can be computed from:

$$\theta_2 = \theta_t - \theta_1 \qquad (8)$$

Finally, the magnitude of the two required velocity impulses can be determined from equations (4) and (5), and the total delta–V from equation (6).

The following example illustrates an orbit transfer from a typical Space Shuttle orbit at 28.5 degrees inclination to an equatorial geosynchronous orbit which has an inclination of zero degrees and a period of 24 hours.

Program IOTA

< Impulsive Orbital Transfer Analysis >

Initial orbit altitude	(nm)	150.000
Initial orbit radius	(nm)	3593.920
Final orbit altitude	(nm)	19323.000
Final orbit radius	(nm)	22766.920
Initial orbit delta–V	(ft/sec)	8056.707
Final orbit delta–V	(ft/sec)	5851.348
Total delta–V	(ft/sec)	13908.055
Initial orbit plane change	(degrees)	2.193
Final orbit plane change	(degrees)	26.307
Total plane change	(degrees)	28.500

A Graphics Display of Three-Body Motion

This application is a computer program called G3BODY which can graphically display orbital motion in the circular restricted three–body problem (CRTBP). The software supports the CGA, EGA, and Hercules graphics modes of the IBM–PC and true compatible computers.

This algorithm and program examples are based on the methods described in "Periodic Orbits in the Restricted Three–Body Problem with Earth–Moon Masses", by R. A. Broucke, JPL TR 32–1168, 1968. Additional information can also be found in the JPL Technical Report, "Periodic Orbits in the Elliptic Restricted Three–Body Problem", by R. A. Broucke, JPL TR 32–1360, 1969. This report contains a listing of a FORTRAN computer program which calculates periodic orbits in the elliptic restricted three–body problem.

Program G3BODY simulates the motion of a spacecraft in the Earth–Moon system. In the discussion which follows, subscript 1 is the Earth and subscript 2 represents the Moon.

The system of second order vector differential equations of motion of a point mass satellite in the CRTBP are

$$\frac{d^2x}{dt^2} - 2\frac{dy}{dt} - x = -(1-\mu)\frac{x - x_1}{r_1^3} - \mu\frac{x - x_2}{r_2^3} \qquad (1a)$$

$$\frac{d^2y}{dt^2} - 2\frac{dx}{dt} - y = -(1-\mu)\frac{y}{r_1^3} - \mu\frac{y}{r_2^3} \qquad (1b)$$

where

x = x component of position

y = y component of position

$x_1 = -\mu$

and

$$x_2 = 1 - \mu$$

$$\mu = \text{Earth–Moon mass ratio} = m_1 / m_2 \approx 1/81.27$$

$$r_1^2 = (x - x_1)^2 + y^2$$

$$r_2^2 = (x - x_2)^2 + y^2$$

The motion of the spacecraft is displayed in a coordinate system which is rotating about the center–of–mass or barycenter of the Earth–Moon system. The motion is confined to the x–y plane.

For convenience the problem is formulated in canonical units. The unit of length is taken to be the constant distance between the Earth and Moon, and the unit of time is chosen such that the Earth and Moon have an angular velocity ω about their barycenter equal to 1. Kepler's third law is then

$$\omega^2 | m_1 \; m_2 |^3 = g(m_1 + m_2) = 1 \qquad (2)$$

and the value for the universal gravitational constant g is 1.

The x and y coordinates of the Earth and Moon in this system are as follows:

$$x_1 = -\mu, \qquad y_1 = 0, \qquad x_2 = 1 - \mu, \qquad y_2 = 0.$$

In his technical reports, Professor Broucke calls these *synodical coordinates.*

Program G3BODY numerically integrates the first order form of the vector differential equations given by Eqs. (1a) and (1b) using the QuickPak Scientific fourth–order, variable step size Runge–Kutta–Fehlberg subroutine RKF45.

In his prize memoir of 1772, Joseph–Louis Lagrange proved that there are five equilibrium points in the restricted three–body problem. If we place a satellite at one of these points with zero initial velocity, it will stay there permanently. These special positions are called *libration points.*

For the Earth–Moon mass ratio value of $\mu = 0.012155099$, the x and y coordinates of these five equilibrium points L_i are:

L_1	x = + 0.836892919	y = 0
L_2	x = + 1.155699520	y = 0
L_3	x = – 1.005064520	y = 0
L_4	x = + 0.487844901	y = + 0.866025404
L_5	x = + 0.487844901	y = – 0.866025404

Three of the libration points are on the x–axis and the other two form *equilateral triangles* with the positions of the Earth and Moon.

Program G3BODY will begin by displaying the following types of orbit options:

< 1 > periodic orbit about L1

< 2 > periodic orbit about L2

< 3 > periodic orbit about L3

< 4 > user input of initial conditions

The first three menu options will display typical periodic orbits about the respective libration point. The initial conditions for each of these orbits are as follows:

(1) Periodic orbit about the L_1 libration point

X0 = 0.300000161	YDOT0 = –2.536145497
MU = 0.012155092	TF = 5.349501906

(2) Periodic orbit about the L_2 libration point

X0 = 2.840829343 YDOT0 = –2.747640074
MU = 0.012155085 TF = 11.933318588

(3) Periodic orbit about the L_3 libration point

X0 = –1.600000312 YDOT0 = 2.066174572
MU = 0.012155092 TF = 6.303856312

Notice that each orbit is defined by an initial x–component of position, X0, an initial y–component of velocity, XDOT0, a value of Earth–Moon mass ratio MU, and an orbital period TF. The initial y–component of position and x–component of velocity for each these orbits is equal to zero. The software will draw the location of the Earth, Moon and libration point on the graphics screen. Note that the size of the Earth and Moon are not drawn to scale, and the actual physical dimension in the x and y directions will probably be distorted due to the aspect ratio of your monitor screen.

If you would like to experiment with your own initial conditions, select option < 4 > from the main menu. The program will then prompt you for the initial x and y components of the position and velocity vector, and the value of the gravitational constant μ to use in the simulation. The software will also ask for an initial and final time to run the simulation, and an integration step size. The number .01 is a good value for step size.

The program will also ask for the graphics mode (CGA, EGA, or Hercules) available on your computer. If you select Hercules mode, be sure to run the program MSHERC.COM before trying to execute G3BODY. You will also be asked to define the window for drawing the graphics. This window is specified by x and y coordinate pairs for the lower left hand corner and the upper right hand corner of the screen.

These values are best determined by trial and error, and will depend on the actual orbit you are trying to plot. By changing the dimensions of the window, you can zoom in or out on the display screen. This allows you to see a better view of the motion as the spacecraft approaches the Earth or Moon, for example.

During the graphics display, the distance between successive trajectory points and the program execution speed are indications of the integration step size. The step size should decrease as the satellite approaches the Earth or Moon. Can you explain why this happens?

Here is a short list of initial conditions for several other periodic orbits which you may want to display with G3BODY.

(1) Retrograde periodic orbit about m_1

X0 = –2.499999883 YDOT0 = 2.100046263
MU = 0.012155092 TF = 11.99941766

(2) Direct periodic orbit about m_1

X0 = 0.952281734 YDOT0 = –.957747254
MU = 0.012155092 TF = 6.450768946

(3) Direct periodic orbit about m_1 and m_2

X0 = 3.147603117 YDOT0 = –3.07676285
MU = 0.012155092 TF = 12.567475674

(4) Direct periodic orbit about m_2

X0 = 1.399999991 YDOT0 = –.9298385561
MU = 0.012155092 TF = 13.775148738

Time of Apogee and Perigee of the Moon

Program APMOON demonstrates how to use several QuickPak Scientific algorithms to solve for the approximate calendar date and Universal Time of perigee and apogee of the Moon. Specifically, we will use subroutines BMINIMA and MINIMA2 to find the time of minimum geocentric distance of the Moon (perigee), and the time of maximum geocentric distance of the Moon (apogee).

The subroutine which computes the geocentric position of the Moon is taken from the paper, "Low–precision Formulae for Planetary Positions", by T.C. Van Flandern and K.F. Pulkkinen. This paper appeared on pages 391–411 of the November 1979 issue of *The Astrophysical Journal Supplement Series.* The algorithm for the Moon's geocentric distance consists of a trigonometric cosine series of 30 terms and is valid for the years 1600 A.D. to 2200 A.D.

The independent argument used in this algorithm is the number of days relative to the epoch January 1, 2000 at 12 hours Universal Time. This can be calculated as a function of the Julian date with the simple equation:

T = JDATE – 2451545

where JDATE is the Julian date and is calculated with the QuickPak Scientific JDATE subroutine.

The algorithm then calculates several fundamental arguments with the next four equations:

GM = 0.374897 + 0.03629164709T

FM = 0.259091 + 0.03674819520T

EM = 0.827362 + 0.03386319198T

GS = 0.993126 + 0.00273777850T

where

GM = mean anomaly of the Moon

FM = argument of latitude of the Moon

EM = mean elongation of the Moon from the Sun

GS = mean anomaly of the Sun

These arguments are in units of revolutions and are converted to radians with a QuickBASIC user–defined function given by:

```
R2R (X) = PI2 * (X – FIX(X))
```

where PI2 is a global constant and is equal to 2π.

All angles are adjusted between 0 and 2π radians by another QuickBASIC function called AMODULO.

Finally, the geocentric distance of the Moon, in the units of earth radii, is calculated with a trigonometric cosine series as follows:

```
RM =    60.36298
      -  0.57994  COS(GM-2EM)
      -  0.08904  COS(2GM)
      -  0.03237  COS(2EM-GS)
      -  0.02358  COS(GM-2EM+GS)
      +  0.01719  COS(EM)
      +  0.01247  COS(GM-2FM)
      +  0.00529  COS(2EM+GS)
      +  0.00398  COS(GM-2EM-GS)
      -  0.00295  COS(2GM-4EM)
      +  0.00249  COS(3GM-2EM)
      +  0.00185  COS(2FM-2EM)
      +  0.00147  COS(GM+2FM-2EM)
      +  0.00139  COS(2GM-2EM+GS)
      -  0.00116  COS(2GM+2EM)
      -  3.27746  COS(GM)
      -  0.46357  COS(2EM)
      +  0.03865  COS(2GM-2EM)
      -  0.02688  COS(GM+2EM)
      -  0.02030  COS(GM-GS)
      +  0.01671  COS(GM+GS)
      +  0.00704  COS(GS)
      -  0.00524  COS(GM-4EM)
      -  0.00366  COS(3GM)
      -  0.00263  COS(EM+GS)
      -  0.00221  COS(GM+2EM-GS)
      -  0.00161  COS(2*(EM-GS))
      -  0.00142  COS(4EM)
      -  0.00118  COS(GM-4EM+GS)
      -  0.00110  COS(2GM-GS)
```

The radius of the Earth is equal to 6378.14 kilometers.

Program APMOON will ask you for an initial calendar date on which to begin the search and a total search duration in days. Be sure to include all four digits of the calendar year.

With this information, the BMINIMA subroutine searches forward in time until it brackets the first lunar perigee. Then subroutine MINIMA2 is called to actually calculate the Julian date. The QuickPak Scientific subroutine CDATE is used to calculate the calendar date from this Julian date. The event time is incremented by one day, and this entire process is repeated, this time looking for a lunar apogee. This is controlled by a function multiplier which alternates between the values 1 and -1, forcing the algorithm to look for a maxima or minima, respectively.

Several bracketing and minimization control parameters such as convergence tolerance, step size multiplier, and the number of iterations are hardwired into the software.

The following is a typical display generated by this program.

Lunar Apogee

Calendar date	March 9, 1991
Universal time	1 hours 9 minutes
Julian Date	2448324.548
Geocentric distance (kilometers)	404292.211

Lunar Perigee

Calendar date	March 22, 1991
Universal time	4 hours 12 minutes
Julian Date	2448337.675
Geocentric distance (kilometers)	369912.814

An Offer to Subscribe to
CELESTIAL COMPUTING

ISSN 1044 - 9825

CELESTIAL COMPUTING

A Journal for Personal Computers and Celestial Mechanics

Volume 1 1988 - 1989

Feature Articles
Symbolic Computing - Predicting Lunar Eclipses - Uniform Extension of Gauss's Boundary Value Problem - Predicting Planetary Positions and Events

Fundamental Astronomy
Julian and Calendar Dates - Sidereal Time - A Computer Program for Precession Astronomical Coordinate Systems and Transformations

Applied Astrodynamics
Sun-synchronous, Repeating-groundtrack Orbits - Optimal Impulsive Orbital Transfer Shadow Conditions of Satellites - Visibility Conditions of Satellites

Symbolic Computing
Perigee and Apogee of the Moon - Geodetic Latitude and Altitude - Closest Approach Between Two Planets - Closest Approach between a Satellite and an Observer

Recreational Computing
A Computer Graphics Display of the Galilean Satellites

Numerical Methods
Matrix, Vector and Trigonometry Utility Functions and Subroutines - Linear Algebra Differential Equations - Computer Programs for Numerical Optimization

ISSN 1044 - 9825

CELESTIAL COMPUTING

A Journal for Personal Computers and Celestial Mechanics

Volume 2 1989 - 1990

Feature Articles

The Stumpff/Weiss Solution of the Four-Body Problem - Predicting Solar Eclipses
The Gauss Method of Orbit Determination - Predicting Lunar Occultations

Fundamental Astronomy

Predicting an Earth Ephemeris - Calculating the Apparent Position of a Star
Calculating a Lunar Ephemeris - The Classical Orbital Elements

Applied Astrodynamics

Mutual Visibility Between Two Earth Satellites - Real-time Orbit Simulation
Goodyear's Method of Orbit Propagation - Predicting Satellite Orbital Lifetime

Symbolic Computing

Symbolic Solutions of Kepler's Equation - Closest Approach between Satellites
Symbolic Computing with DERIVE and QUICK

Recreational Computing

Computer Graphics Display of Three-body Motion - Graphical Orbital Motion
Graphics Display of Orbital Events - Zero-velocity Contour Graphics

Numerical Methods

Solving Non-linear Equations - A Least-squares Curve Fit Computer Program
Real and Complex Roots of a Polynomial - Numerical Integration of Functions

WELCOME TO CELESTIAL COMPUTING

The use of personal computers in celestial mechanics seems only natural. The computer can help provide a vivid understanding of fundamental concepts of astronomy and celestial mechanics. We can also use the power of computers to predict unique celestial events and phenomena which we have yet to witness. Although celestial mechanics is the world's oldest science, it attracts our attention today perhaps even more than it did many, many centuries ago.

Celestial Computing is written for two types of computer users. It provides technical information for the programmer who is interested in the math and physics required to solve problems in celestial mechanics, and it will also be useful for the person who simply wants reliable and accurate astronomical software.

In *Celestial Computing*, we will focus on *computer applications* in the following areas of celestial mechanics:

- **Astronomy** — the observation, calculation and interpretation of the characteristics of celestial bodies and phenomena.
- **Astrometry** — the study of positions and motions of celestial bodies.
- **Astrodynamics** — the study of the motion and behavior of man–made spacecraft.

The purpose of *Celestial Computing* is to provide computer methods which will allow everyone to pursue these areas of celestial mechanics. Each computer application will include a discussion of the mathematics and physics of the problem. We will also examine books, technical publications, and other computer programs about celestial mechanics, and try our best to provide material which is easy to use and understand.

Each issue contains a feature article, and regular columns in the areas of fundamental astronomy, applied astrodynamics, symbolic computing, and numerical methods. There is also a recreational computing column which emphasizes computer graphics to illustrate many fundamental concepts of celestial mechanics and astronomy.

The following is a brief description of the regular columns presented in each issue of *Celestial Computing*.

Feature Article

Each issue of *Celestial Computing* will present a feature article of general interest. Some of the topics covered in Volumes 1 and 2 are as follows:

- The prediction of lunar eclipses
- The prediction of solar eclipses
- Tracking and observing earth satellites
- Lunar occultations of stars and planets
- Calculating planetary positions and unique events
- Real–time orbit simulation of a space telescope
- Computer methods for orbit determination

Fundamental Astronomy

In this column of *Celestial Computing*, we will present interactive QuickBASIC computer programs which can be used to study and understand fundamental concepts of astronomy. The following is a list of some of the topics we will address:

- Julian and calendar dates
- The accurate calculation of sidereal time
- Precession and nutation in astronomy
- Astronomical coordinate systems and transformations
- The calculation of classical orbital elements

Applied Astrodynamics

In this regular department of *Celestial Computing*, we will present computer solutions to classic and unique problems in the field of astrodynamics. This is an area where we can apply our knowledge of celestial mechanics to solve problems related to manned and unmanned spaceflight. In this column, we will present computer applications in the following areas:

- Spacecraft trajectory analysis
- The prediction of orbital events
- Methods of orbit design
- Interplanetary spaceflight

Symbolic Computing

This is a regular column of *Celestial Computing* where we will use different symbolic computing programs such as MathCAD, Eureka: The Solver, and Derive to solve unique problems in celestial mechanics. Typical topics which will be covered in this column include the following:

- Symbolic computing solutions of Kepler's equation
- Symbolic computing solutions for the geodetic latitude and altitude of an earth–orbiting spacecraft
- Symbolic computing solutions for the closest approach between a satellite and an observer on an oblate earth
- Symbolic computing solutions of lunar/planetary events

Recreational Computing

This column of *Celestial Computing* is dedicated to computer applications which are both fun and entertaining. Many of these programs emphasize graphics to help the user visualize different types of astronomical concepts and celestial motions.

Typical graphics applications which have appeared in this column include:

- A computer graphics display of the Galilean satellites
- A computer graphics display of the Three–Body problem
- Computer graphic displays of orbital events
- Computer graphic displays of orbital motion

Numerical Methods

In this regular column of *Celestial Computing*, we will present numerical methods and procedures which can be used to solve a variety of problems in celestial mechanics, astronomy and astrodynamics. Many of these methods will be QuickBASIC computer programs and subroutines which you can use as *modules* in your own programs and computer applications. Each of these modules also includes a short program which demonstrates how to use the software correctly.

We will present computer methods in such areas as linear algebra, numerical integration of differential equations, the solution of non–linear equations and numerical optimization.

Celestial Book Review

In this quarterly column books, technical reports, and other publications pertaining to celestial mechanics are reviewed. The following is a short list of several of the publications which have been reviewed:

- *Fundamentals of Celestial Mechanics* by J. M. A. Danby
- *Astronomical Formulae for Calculators* by Jean Meeus
- *Introduction to BASIC Astronomy with a PC* by J. L. Lawrence
- Publications of the U.S. Naval Observatory

Celestial Software Review

In this regular column of *Celestial Computing*, we review both public domain and commercial software which may be of interest to readers. Among the software packages reviewed:

- Program XonVu
- DERIVE: A Mathematical Assistant Program
- Program GEOCLOCK
- The Interactive Computer Ephemeris (ICE)
- Program COLPACK

Subscription and Disk Information

Celestial Computing is published four times per year, around the time of the solstices and equinoxes (March, June, September and December). Both journal and floppy disk subscriptions are available for the IBM–PC and true compatible computers. The written journal contains a technical discussion about each article and instructions which explain how to use the software. The floppy disk contains the actual computer programs.

The computer programs are available on both 5 1/4", 360K capacity floppy disks and 3 1/2", 720K capacity floppy disks. Please be sure to specify the format when ordering disk subscriptions. Both QuickBASIC *source* code and *executable* programs are provided on the disks. The Microsoft QuickBASIC compiler is *not* required to run these programs.

Please submit payment in the form of a personal check or money order (no credit cards please), in *U.S. dollars* and drawn on a *U.S. bank*, payable to *Science Software*. The costs to countries other than the United States are shown in parentheses. These prices include first class shipping within the United States and air mail shipping elsewhere.

Please send all subscription orders and correspondence to:

Science Software
7370 South Jay Street
Littleton, CO 80123-4661 USA
(303) 972-4020

□ *Celestial Computing* journal subscription
(1 year, 4 issues) $29.95 U.S. ($39.95 elsewhere)

□ *Celestial Computing* disk subscription
(1 year, 4 disks) $19.95 U.S. ($24.95 elsewhere)

□ *Celestial Computing* back issues
(Please indicate issue(s) by volume and number)
- Journal $6.95 U.S. ($8.95 elsewhere) _____
- Disk $4.95 U.S. ($6.95 elsewhere) _____

Please specify the floppy disk format
□ 5 1/4", 360K □ 3 1/2", 720K

□ Bound copy–Volume 1 $29.95 U.S. ($39.95 elsewhere)
□ Floppy disks–Volume 1 $19.95 U.S. ($29.95 elsewhere)

□ Bound copy–Volume 2 $29.95 U.S. ($39.95 elsewhere)
□ Floppy disks–Volume 2 $19.95 U.S. ($29.95 elsewhere)

Name _____

Street address _____

City, state, zip _____