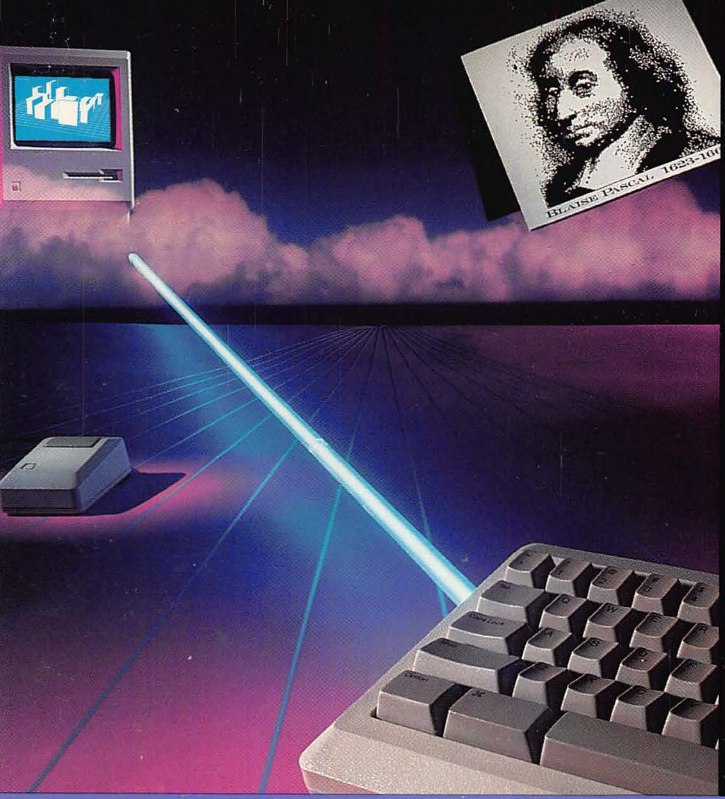
TURBO PASCAL TOOLBOX™

NUMERICAL METHODS

A complete collection of Turbo Pascal® routines and programs



MACINTOSH™

BORLAND

TURBO PASCAL Numerical Methods Toolbox™

First Edition
Printed in USA
9 8 7 6 5 4 3 2 1

READ ME FIRST

In order to provide you with the latest technical information on our products, announcements of future updates, and up-to-the-minute information on new products, please complete and return this registration form. Be sure to read the Borland No-Nonsense License Statement on the other side.

Technical Support—To receive telephone technical support, you must be the registered owner of the Borland product. Prompt technical support is available through the Borland Forums on CompuServe; just type GO BOR at any CompuServe prompt. If you need further assistance, write a letter or call Borland and be prepared to give the product name, version number, and the serial number found on the label of your master diskette.

The README File—If present on your master diskette, this file contains important information that may not be in the manual. To view this file, simply type README at the command prompt. Be sure to read this file before you call for technical support.

Thank you for completing this product registration card and returning it promptly. We want to keep you informed.

Name and address must be filled in by the person using the product for the registration form to be valid. (Please print legibly.)

Serial # ______________________ Date Purchased: ____/____/____
M D Y

Name: ______________________ Title: ______________
last first middle initial

Company Name: ______________________ Department: ______________

Address: ______________________ Mail Stop: ______________

City: ____________ State: ________ Zip: ________ Country: ________

Phone # (________) ______________-______________ ☐ Work ☐ Home

I have read and agree to the terms of the Borland No-Nonsense License Agreement:

Signature ______________________ Date: ____/____/____

In order to help us serve your needs, please complete the following:

Microcomputer used:
☐ IBM PC or compatible ☐ Macintosh ☐ other ______________

Where did you purchase this program?

☐ Borland direct	☐ other mail order	
☐ discount retailer	☐ full-service retailer	☐ other ________

Software was bought for:
☐ self ☐ company I work for ☐ company I own

Where will you use this program?
☐ at home ☐ at work ☐ both ☐ other ______________

Where did you hear about this program?

☐ ad in computer publication	☐ published review	
☐ ad in general interest publication	☐ retailer	
☐ other user	☐ trade show	☐ other ________

Nature of business:

☐ finance/real estate/insurance	☐ computer consulting	☐ transportation/communication/utilities
☐ retail/wholesale	☐ other consulting	☐ mining/construction
☐ legal	☐ software publishing	☐ governmemt
☐ health	☐ other publishing	☐ military
☐ professional services	☐ computers/electronics manufacturing	☐ education
☐ other services	☐ other manufacturing	☐ other ________

Nature of occupation:

☐ MIS/DP, systems analyst	☐ administration	☐ operations	☐ student
☐ programming	☐ finance/accounting	☐ consulting	☐ homemaker
☐ engineering/scientific	☐ sales/marketing	☐ teacher/trainer	☐ retired
☐ doctor/lawyer	☐ manufacturing/production	☐ clerical	☐ other ________
☐ other professional	☐ purchasing		

Number of employees at business: ☐ 1-24 ☐ 25-99 ☐ 100-499 ☐ 500-1999 ☐ 2000-9999 ☐ more than 9999
Number of microcomputers at business: ☐ 1-9 ☐ 10-49 ☐ 50-249 ☐ 250-999 ☐ more than 999

Other Borland products owned:

Programming languages		**Business Applications**	**Utility Programs:**
☐ Turbo Pascal	☐ Turbo Prolog	☐ Reflex	☐ SideKick
Turbo Pascal Toolboxes	☐ Turbo Prolog TB	☐ Reflex Workshop	☐ Traveling SideKick
☐ Tutor	☐ Turbo Basic	☐ Sprint	☐ SuperKey
☐ Database	☐ Turbo Basic Database TB	**For the Macintosh**	☐ Turbo Lightning
☐ Editor	☐ Turbo Basic Editor TB	☐ Turbo Pascal	☐ Lightning Word Wizard
☐ Graphix	☐ Turbo Basic Telecom TB	☐ Reflex	**Scientific & Engineering**
☐ GameWorks	☐ Turbo C	☐ SideKick	☐ Eureka
☐ Numerical Methods		☐ Eureka	☐ Other ________

What other software do you use:

☐ spreadsheet	☐ languages	☐ desktop publishing	☐ RAM-resident utilities
☐ database	☐ accounting	☐ business graphics	☐ games
☐ word processor	☐ communications	☐ CAD/CAM/CAE	
☐ project management	☐ network	☐ other ________	

What hardware peripherals do you use?

☐ modem	☐ hard disk	☐ EGA card
☐ laser printer	☐ plotter	☐ other printer ________
☐ mouse	☐ other peripheral	

BOR 0045F

Turbo Pascal Numerical Methods Toolbox

BOR 0420 Fold at dotted line. Tape closed. Drop in mail. No postage necessary. **27**

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 200 SANTA CRUZ, CA

POSTAGE WILL BE PAID BY ADDRESSEE

BORLAND
INTERNATIONAL

4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CALIFORNIA 95066-9987

Turbo Pascal Numerical Methods Toolbox™

For the Macintosh

BORLAND INTERNATIONAL, INC.
4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CALIFORNIA 95066

10 9 8 7 6 5 4 3 2 1

Table of Contents

Introduction

The Turbo Pascal Numerical Methods Toolbox is a reference manual for both the student of numerical analysis and the professional needing efficient routines. An elementary background in calculus and linear algebra is assumed, although many of the algorithms use only high-school-level mathematics. A general knowledge of Turbo Pascal® is also assumed. If you need to brush up on your knowledge of Pascal, we suggest looking at the *Turbo Pascal for the Macintosh Reference Manual.*

Before you begin using a particular routine, read through this brief introductory chapter and then refer to the chapter that interests you.

Toolbox Functions

The Turbo Pascal Numerical Methods Toolbox provides routines for

- Finding solutions to equations
- Interpolations
- Calculus
- Numerical derivatives and integrals
- Matrix operations: inversions, determinants, and eigenvalues

- Differential equations
- Least-squares approximations
- Fourier transforms

About this Manual

The major areas in numerical analysis are represented in this Toolbox, with each chapter focusing on a particular problem. Each routine begins with a general description of the implemented algorithm or numerical method. (References to numerical analysis texts are provided for each numerical procedure.) User-supplied types, functions, and input and output parameters are defined, and the syntax of the procedure call is provided. If appropriate, a "Comments" section is also provided.

Finally, every algorithm in the Toolbox is accompanied by a general-purpose program that handles all the necessary I/O, while allowing you to try each algorithm without building any code. Handily, these sample programs will often reduce the coding your own application may require.

As an example, let's say you want to find the roots to an equation in one variable. First, you would read the introduction to Chapter 2, "Roots to Equations in One Variable," and choose the numerical method best suited to your particular problem. Second, you would run the sample program for the desired numerical method to determine the necessary input and output. Third, you would write a Turbo Pascal function defining your equation, using the function already coded in the sample program as a guide. Fourth, you would run the sample program with your function substituted for the original one. Of course, if these algorithms are to be part of a larger program, you must build all the interfaces to the other parts of the system; but this should only be done after you gain experience with the particular numerical method.

Several books are referred to throughout the text; complete references are listed at the back of the book in the section entitled "References."

The body of this manual is printed in normal typeface; other typefaces serve to illustrate the following:

`Alternate`	This type displays program examples and procedure and function declarations.
Italics	This type emphasizes certain concepts, first-mentioned terms, and mathematical expressions.
Boldface	This type marks the reserved words of Turbo Pascal in text and in program examples.

On the Distribution Disks

The routines for this Toolbox are contained on two packed disks. Their contents and general installation instructions are covered in Chapter 1.

System Requirements

To use the Turbo Pascal Numerical Methods Toolbox you must have one of these Macintosh computers: 512K, Plus, SE or II; with one 800K or two 400K disk drives.

You will also need Turbo Pascal version 1.0 to run the routines.

Acknowledgements

We refer to the following products in this book.

- Turbo Pascal is a registered trademark and Turbo Pascal Numerical Methods Toolbox for the Macintosh is a trademark of Borland International, Inc.
- ImageWriter and LaserWriter are trademarks of Apple Computers, Inc.

C H A P T E R 1

Routine Beginnings

This chapter provides you with everything you need to start using the routines in this Toolbox. We'll discuss the files supplied on the disks. We also briefly discuss data types and defined constants used in the Toolbox, and the setting of compiler directives.

First, though, before we thrust you into the middle of numerical madness, let's take a look at one way to use this Toolbox.

Using the Toolbox: An Example

In late 1986 and early 1987, the America's Cup 12-meter yacht championship was held. The 12-meter yachts are just large sailboats, but the competition is so intense that the only way to be competitive is to use dozens of people, spend millions of dollars, design a special boat, and spend a couple of years training for the race. The race has become so sophisticated that many of the sailboats have on-board computers and other electronic equipment.

To keep stride with other challengers, one yacht's crew used personal computers, and of course, Borland software. They used Turbo Pascal to design the boat's hull. They used Reflex®: The Database Manager to maintain their databases and to display plots while the boat was sailing. And when it came time to do some mathematical modeling, again they turned to Borland for its inimitable software and chose the Turbo Pascal Numerical Methods Toolbox.

Simply speaking, the problem they had was one of "precision monitoring." It takes a crew of very highly skilled sailors to compete in America's Cup races, but even the best skippers cannot act with sufficient precision to win. A typical race lasts for several hours, and the winner usually wins by only a few feet.

The electronic equipment on a boat can sense with reasonable accuracy all of the crucial variables: boat velocity, wind velocity, boat direction, boat position, and so on. This data must then be made available to the skipper in a coherent form, and he/she must decide at what angle to place the rudder based on that information. The problem is too complex to rely on intuition alone.

Even just displaying the velocity is more complex than you might think at first. When sailing on the ocean, the waves are big enough that the velocity is in constant flux. Fortunately, the fluctuations due to the waves represents a steadily periodic force. By using *Fourier transforms* (Chapter 10), the crew was able to identify the periodic portion of the velocity and subtract it out. The result: the velocity as a function of time but with the wave fluctuations eliminated. The graph of this modified velocity is much smoother, and allows the skipper to tell much more quickly and accurately whether the boat is accelerating or decelerating.

To measure the acceleration quantitatively, the crew used the fact that the acceleration is the derivative of the velocity. They were able to do this easily with *differentiation* routines (Chapter 4). They were also able to directly measure the distance travelled by using *integration* routines (Chapter 5), and the fact that distance is the integral of the speed.

Perhaps the most difficult problem in navigating a sailboat is aiming the rudder. You can't just aim the boat in the direction that you want to go, rather you have to pick a direction that you can sail rapidly, depending on the wind direction. An experienced skipper can judge this pretty well, but not well enough. Every boat is a little different, and the best way to handle one boat is not necessarily the best way to handle another.

So, the team ran extensive trial races with the boat to gather data on how the boat performed in various circumstances. The data was collected automatically by electronic instruments on board, and stored digitally on floppy disks. They then used Reflex to manage the data and to display graphs. But they lacked the tools to relate their data to the data they would have under actual racing conditions.

In order to predict the behavior of their boat in an actual race, the team created a model from their collected data using *least-squares* routines (Chapter 9). With the least-squares routines, you can create a multiparameter model and then find the values of the parameters that make the model most accurately fit the data. With a mathematical model of the boat's behavior, the team was then able to predict how the boat would perform under circumstances similar but not identical to its practices.

This, of course, is just one of many possible applications of this Toolbox. Now, let's go on to the fundamentals.

The Distribution Disks

All of the Toolbox routines are contained on two disks. Each disk has folders corresponding to chapters in the manual.

The files for each chapter are self-contained and do not require any files from any other chapter, with these exceptions:

- All files require Turbo Pascal (not included).
- Most files require the IOSelection unit, located on Disk 2.
- The files for Chapter 11 require the compiled units from Chapters 9 and 10, as well as the TurboGraph unit from Chapter 11.

The numerical analysis routines are in the files with the .unit suffix. The files with the .pas suffix are demonstration programs. To run a demonstration program, get into Turbo Pascal and load the .pas file of your choice. The menus are self-explanatory. The .dat files contain input data for specific .pas files.

Contents of the distribution disks:

NMT Disk 1:

Read Me	Read Me program (double click on this)
Read.file	Text for the Read Me program
FFTComplex	Compiled Unit from Chapter 10
FFTDemo	Fast Fourier Transform Demo program
FFTDemo.pas	Source for Fast Fourier Transform Demo
FFTMenu.r	RMaker source for FFTMenu.rsrc
FFTMenu.rsrc	RMaker output for Fast Fourier Transform Demo
FFTReal	Compiled Unit from Chapter 10
FFTRoutines	Compiled Unit from Chapter 10
LeastSquares	Compiled Unit from Chapter 9
LeastSquaresDemo	Least Squares Demo program
LSQDemo.pas	Source for Least Squares Demo
LSQMenu.r	RMaker source for LSQMenu.rsrc
LSQMenu.rsrc	RMaker output for Least Squares Demo
Sample11A.dat	Data file for Least Squares Demo
Sample11B.dat	Data file for Fast Fourier Transform Demo
TurboGraph.unit	Source to the TurboGraph Unit

NMT Disk 2:

IO Selection	Packed source for IO Selection
Chapter 2	Packed source for Chapter 2
Chapter 3	Packed source for Chapter 3
Chapter 4	Packed source for Chapter 4
Chapter 5	Packed source for Chapter 5
Chapter 6	Packed source for Chapter 6
Chapter 7	Packed source for Chapter 7
Chapter 8	Packed source for Chapter 8
Chapter 9	Packed source for Chapter 9
Chapter 10	Packed source for Chapter 10
UnPack	The program to unpack the packed files

Installation

The files Chap2 through Chap10 on your disk are packed source for the corresponding chapters in this manual. In order to use these files, you must first unpack them with the UnPack program.

How to use the UnPack program:

1. Double-click on the icon for the UnPack program. You will be asked to name the Packed file to UnPack.
2. Using the Standard File Dialog, select the Packed file to UnPack. You will be asked for the Volume/Folder to save all of the source files to.
3. Using the Standard File Dialog, select the Volume/Folder to hold the source files in that Packed file.

And now you are ready to begin.

Files on Distribution Disks

Note: These files are not copy protected. All files are ordinary text files.

Contents of the folders.

IO Selection Routines common to all chapters

IO Selection	IO Selection.rsrc	IO Selection.unit
IO Selection.r		

Chap2 "Roots to Equations in One Variable"

Bisect.pas	Newtdefl.pas	Roots of Equat
Laguerre.pas	Raphson.pas	Roots of Equat.unit
Muller.pas	Raphson2.pas	Secant.pas

Chap3 "Interpolation"

Cube_cla.pas	Lagrange.pas	Sample3E.dat
Cube_fre.pas	Sample3A.dat	Sample3F.dat
Divdif.pas	Sample3B.dat	Sample3G.dat
Interpolation	Sample3C.dat	Sample3H.dat
Interpolation.unit	Sample3D.dat	Sample3I.dat

Chap4 "Numerical Differentiation"

Deriv.pas	Deriv2fn.pas	Interdrv.pas
Deriv2.pas	Differentiation	Sample4A.dat
Derivfn.pas	Differentiation.unit	Sample4B.dat

Chap5 "Numerical Integration"

Adapgaus.pas	Integration.unit	Trapzoid.pas
Adapsimp.pas	Romberg.pas	
Integration	Simpson.pas	

Chap6 "Matrix Routines"

Det.pas	Inverse.pas	Sample6A.dat
Dirfact.pas	MatrixRoutines	Sample6B.dat
Gauselim.pas	MatrixRoutines.unit	Sample6C.dat
Gaussidl.pas	Partpivt.pas	Sample6D.dat

Chap7 "Eigenvalues and Eigenvectors"

EigenRoutines	Jacobi.pas	Wielandt.pas
EigenRoutines.unit	Power.pas	
Invpower.pas	Sample7A.dat	

Chap8 "Initial Value and Boundary Value Methods"

Adams_1.pas	Runge_1.pas	Runge_s2.pas
DifferentialEquat.unit	Runge_2.pas	Shoot2.pas
Linshot2.pas	Runge_N.pas	
RKF_1.pas	Runge_s1.pas	

Chap9 "Least-Squares Approximations"

Least.pas	LeastSquares.unit
LeastSquares	Sample9A.dat

Chap10 "Fast Fourier Transform Routines"

FFTComplex	FFTReal.unit	Sample10B.dat
FFTComplex.unit	FFTRoutines	Sample10C.dat
FFTProgs.pas	FFTRoutines.unit	
FFTReal	Sample10A.dat	

All sample programs use the IO Selection unit from the disk. This file includes procedures that are common to all sample programs. When copying any of the sample programs to a disk, be sure to also copy the files IO Selection and IO Selection.rsrc to that disk or the sample programs will not compile.

We have made the sample programs general and easy to use. For example, numerical input can originate from the keyboard (where improper input is trapped) or from a text file; output can be sent to the printer, screen, or text file; other refinements are also included. Since, to a beginner, the supporting code may obscure the simplicity of calling the procedure, we have included a minimal sample program for *Newton-Raphson*'s method of root-finding (Raphson2.pas).

The Graphics Demos

Because graphic displays are often an essential part of numerical analysis, we have included two demonstration programs that involve display of numerical results. These programs rely on graphics routines contained in the unit library TurboGraph supplied on the distribution disk.

The demonstration programs are on Disk 1. For instructions about how to run or recompile them, see Chapter 11.

Data Types and Defined Constants

Data types that might be confused with those in the calling program have been prefixed with the letters TN (for Turbo Numerical); for example, *TNmatrix* or *TNvector*. All Toolbox-type declarations are contained in the particular Toolbox unit you are using in your program. Therefore, you must recompile the unit if you want to modify one of the type declarations. (You might want to do this to dimension arrays based on your particular needs.) For example, the *Lagrange* procedure requires the definition

```
type TNvector = array[0..TNArraySize] of Extended;
```

The identifier *TNArraySize* should be optimized by the user, although we have set a default value in each of the Toolbox units. It may be replaced with an integer or byte constant.

Compiler Directives

Aside from the usual default values of the compiler directives in standard Turbo Pascal, we have set the compiler directive to {$R+} in all units that use arrays, and to {$I−} in all sample programs. The first directive checks to see that all array-indexing operations are within the defined bounds and all assignments to scalar and subrange variables are within range. The latter directive disables I/O error-checking. All the sample programs have their own I/O error-checking procedures (contained in the unit library IO Selection), so the {$I−} directive must remain disabled in the sample programs. The array checker {$R+} should always be active, since the performance penalty is slight and the advantages are significant.

C H A P T E R **2**

Roots to Equations in One Variable

The routines in this chapter are for finding the roots of a single equation in one real variable. A typical problem is to solve

$$x * \exp(x) - 10 = 0$$

In general, the routines find a value of x, where x is a scalar real variable, satisfying

$$f(x) = 0.0$$

where f is a real-valued function that you program in Pascal.

All of the methods are *approximate* methods, meaning that they find an approximate value of x that makes $f(x)$ close to zero. Because of round-off error, it is usually not possible to find the exact value of x. Furthermore, they are all *iterative* methods, meaning that you specify some initial guess that is some value for x, which you think is reasonably close to the solution. The routine repeats some calculations that replace the guess x with a more accurate guess until the required level of accuracy is achieved.

The *bisection* method returns an approximation to a root of a real continuous function of the real variable x. This method always converges (as long as the function changes signs at a root), but may do so relatively slowly.

The *Newton-Raphson* method also returns an approximation to a root of a real function f of the real variable x. When this algorithm converges, it is usually faster than the bisection method. If more than one root of a polynomial equation is desired, then use *Newton-Horner's* method.

The *secant* method is similar to the Newton-Raphson method, but doesn't require knowledge of the first derivative of the function. Consequently, it is more flexible than the Newton-Raphson method, though somewhat slower.

Newton-Horner's method applies Newton's method to real polynomials. It also uses *deflation* techniques to attempt to approximate all the real roots of a real polynomial. Both the Newton-Horner and Newton-Raphson methods are faster than the bisection and secant methods, but are undefined if $|f'(x)| <= $ *TNNearlyZero*.

The Newton-Horner and Newton-Raphson methods both converge around multiple roots, although convergence is slow. These algorithms depend upon an initial approximation of the root. If the initial approximation is not sufficiently close to the root, the Newton methods may not converge. In some instances, an initial choice may lead to successive iterations that oscillate indefinitely about a value of x usually associated with a relative minimum or relative maximum of f. In either case, the bisection method could be used to determine the root or to determine a close approximation to the root that can be employed as an initial approximation in the Newton-Raphson or Newton-Horner methods.

Müller's method returns an approximation to a root (possibly complex) of a complex function of the complex variable x. Although Müller's method can approximate the roots of polynomials, we recommend that you use Newton-Horner's method, the secant method, or (in the case of complex polynomials) *Laguerre*'s method to find the roots of polynomials.

Laguerre's method attempts to approximate all the real and complex roots of a real or complex polynomial. Laguerre's method is very reliable and quick, even when converging to a multiple root. This is the best general method to use with polynomials.

A caution when solving polynomial equations: Polynomials can be ill-conditioned, in the sense that small changes in the coefficients may lead to large changes in the roots.

Stopping Criteria

All the root-finding routines use the function *TestForRoot* to determine if a root has been found.

```
function TestForRoot(X, OldX, Y, Tol : Real) : Boolean;

{---------------------------------------------------------------}
{ Here are four stopping criteria. If you wish to               }
{ change the active criteria, simply comment off the current    }
{ criteria (including the appropriate or) and remove the comment }
{ brackets from the criteria (including the appropriate or) you  }
{ wish to be active.                                             }
{---------------------------------------------------------------}

begin
  TestForRoot :=                        {-------------------------}
    (ABS(Y) <= TNNearlyZero)            { Y=0                     }
                                        {                         }
          or                            {                         }
                                        {                         }
    (ABS(X - OldX) < ABS(OldX*Tol))     { relative change in X    }
                                        {                         }
                                        {                         }
(*         or                        *) {                         }
(*                                   *) {                         }
(*        (ABS(X - OldX) < Tol)      *) {  absolute change in X   }
(*                                   *) {                         }
(*         or                        *) {                         }
(*                                   *) {                         }
(*        (ABS(Y) <= Tol)            *) {  absolute change in Y   }
                                        {-------------------------}
end; { procedure TestForRoot }
```

The four separate tests provided by function *TestForRoot* may be used in any combination. The default criteria tests the absolute value of *Y* and the relative change in *X*. If you wish to change the active criteria, simply comment off the current criteria (including the appropriate **or**) and remove the comment brackets from the criteria (including the appropriate **or**) you wish to be active.

The first criterion simply checks to see if *Y* is zero (*TNNearlyZero* is defined at the beginning of the procedure). This criterion should usually be kept active.

The second criterion examines the relative change in *X* between iterations. To avoid division by zero errors, *OldX* has been multiplied through the inequality.

The third criterion checks the absolute change in *X* between iterations.

The fourth criterion determines the absolute difference between *Y* and the allowable tolerance. **Note:** The parameter *Tol*(erance) means something different in each test. Be sure you know which tests are active when you input a value for *Tol*.

Root of a Function Using the Bisection Method (Bisect.pas)

Description

This method (Burden and Faires 1985, 28 ff.) provides a procedure for finding a root of a real continuous function *f*, specified by the user on a user-supplied real interval [*a*,*b*]. The functions *f*(*a*) and *f*(*b*) must be of opposite signs. The algorithm successively bisects the interval and converges to the root of the function. You must also specify the desired accuracy to which the root should be approximated.

User-Defined Function

```
function TNTargetF(x : Extended) : Extended;
```

The procedure *Bisect* determines the roots of this function.

Input Parameters

`LeftEndpoint:Extended;`	Left end of the interval
`RightEndpoint:Extended;`	Right end of the interval
`Tol:Extended;`	Indicates accuracy of solution
`MaxIter:Extended;`	Maximum number of iterations permitted

The preceding parameters must satisfy the following conditions:

1. *LeftEndpoint* < *RightEndpoint.*
2. *TNTargetF(LeftEndpoint)* $*$ *TNTargetF(RightEndpoint)* < 0; the endpoints must have opposite signs.
3. *Tol* > 0.
4. *MaxIter* $\geq$ 0.

Output Parameters

`Answer:Extended;`	An approximate root of *TNTargetF*
`fAnswer:Extended;`	The value of the function at the value *Answer*
`Iter:Integer;`	Number of iterations to find answer
`Error:Byte;`	0: No error 1: *Iter* > *MaxIter* 2: Endpoints are of the same sign 3: *LeftEndpoint* > *RightEndpoint* 4: *Tol* ≤ 0 5: *MaxIter* < 0

If *Error* = 1 (maximum number of iterations exceeded), *Answer* is set to the last x value tested and *fAnswer* is set to *TNTargetF*(*Answer*). If *Error* > 1, then the other output parameters are not defined.

Syntax of the Procedure Call

```
Bisect(LeftEndpoint, RightEndpoint, Tol, MaxIter, Answer, yAnswer, Iter,
      Error,@TNTargetF);
```

The procedure *Bisect* determines the roots of function *TNTargetF*.

Comments

If a root occurs at a relative maximum or relative minimum, the bisection method will be unable to locate that value of p if p does not occur as an endpoint of a subinterval.

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter.

Sample Program

The sample program Bisect.pas provides I/O functions that demonstrate the bisection algorithm. To modify this program for your own function, simply change the definition of function *TNTargetF*. Note that the address of *TNTargetF* is passed into the *Bisect* procedure.

Example

Problem. Determine the solution to the equation cos(x) = x.

1. Write the following code for function *TNTargetF* into Bisect.pas:

```
{---------- HERE IS THE FUNCTION -----------}

function TNTargetF(x : Extended) : Extended;
  begin
    TNTargetF := Cos(x) - x;
  end;                { function TNTargetF }

{------------------------------------------}
```

2. Run Bisect.pas:

```
Left endpoint:    0
Right endpoint: 100

Tolerance (> 0): 1E-6

Maximum number of iterations (> 0): 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
               Left endpoint:   0.00000000000000e+0
              Right endpoint:   1.00000000000000e+2
                   Tolerance:   1.00000000000000e-6
Maximum number of iterations: 100

   Number of iterations:  28
        Calculated root:   7.39085301756859e-1
  Value of the function
 at the calculated root:  -2.82073423997129e-7
```

Root of a Function Using the Newton-Raphson Method (Raphson.pas)

Description

This example uses Newton-Raphson's algorithm (Burden and Faires 1985, 42 ff.) to find a root of a real user-specified function when the derivative of the function and an initial guess are given. The algorithm constructs the tangent line at each iterate approximation of the root. The intersection of the tangent line with the x-axis provides the next iterate value of the root. You must specify the desired tolerance to which the root should be approximated.

User-Defined Functions

```
function TNTargetF(x : Extended) : Extended;

function TNDerivF(x : Extended) : Extended;
```

The procedure *Newton Raphson* determines the roots of the function *TNTargetF*.

The function *TNDerivF* must be the first derivative of function *TNTargetF*.

Input Parameters

`InitGuess:Extended;`	User's initial approximation to the root
`Tol:Extended;`	Tolerance in answer (see "Comments")
`MaxIter:Integer;`	Maximum number of iterations permitted

The preceding parameters must satisfy the following conditions:

1. *Tol* > 0
2. *MaxIter* > 0

Output Parameters

`Root:Extended;`	Approximate root.
`Value:Extended;`	Value of the function at the approximate root.
`Deriv:Extended;`	Value of the derivative at the approximated root.
`Iter:Integer;`	Number of iterations needed to find the root.
`Error:Byte;`	0: No error. 1: *Iter* < *MaxIter.* 2: The slope is zero (see "Comments"). 3: *Tol* ≤ 0. 4: *MaxIter* < 0.

If a root is found, it is returned along with the value of the function at the root (which, of course, should be close to zero) and the value of the derivative at the root. If *Error* ≤ 2, the data from the last iteration is returned.

Syntax of the Procedure Call

```
Newton_Raphson(InitGuess, Tol, MaxIter, Root, Value, Deriv, Iter, Error, @TNTargetF,
               @TNDerivF);
```

Comments

Newton's method involves division by the value of the derivative of the function. Should the algorithm attempt to do any calculations at a point where the derivative is less than *TNNearlyZero*, the routine will stop and return an error message (Error = 2).

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter.

Sample Program

The sample program Raphson.pas provides I/O functions that demonstrate the Newton-Raphson algorithm. Note that the addresses of *TNTargetF* and *TNDerivF* are passed to the *Newton_Raphson* procedure.

The program Raphson2.pas also provides I/O functions that demonstrate the Newton-Raphson method. It is an extremely bare-bones program and is provided for the newcomer to Turbo Pascal who wants to see a simple, straightforward application of a Toolbox routine.

Example

Problem. Determine the solution to the equation cos(x) = x.

1. Code the following two functions into Raphson.pas (or Raphson2.pas):

```
{--------- HERE IS THE FUNCTION ------------}

function TNTargetF(x : Extended) : Extended;
  begin
    TNTargetF := Cos(x) - x;
  end;                { function TNTargetF }

{------------------------------------------}


{------- HERE IS THE DERIVATIVE ------------}

function TNDerivF(x : Extended) : Extended;
  begin
    TNDerivF := -Sin(x) - 1;
  end;                 { function TNDerivF }

{------------------------------------------}
```

2. Run Raphson.pas:

```
Initial approximation to the root: 0

Tolerance (> 0): 1E-6

Maximum number of iterations (>= 0): 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
       Initial approximation:   0.00000000000000e+0
                   Tolerance:   1.00000000000000e-6
Maximum number of iterations: 100


   Number of iterations:   5
        Calculated root:   7.39085133215161e-1
  Value of the function
 at the calculated root:   0.00000000000000e+0
Value of the derivative
 of the function at the
        calculated root:  -1.67361202918321e+0
```

Here is the Raphson2.pas version of the same function:

```
Initial approximation to the root:   0
                    Tolerance(>0):   1E-6
Maximum number of iterations(>=0): 100

Error = 0

             Number of iterations:   5
                  Calculated root:   7.39085133215161e-1
                     Value of the
             function at the root:   0.00000000000000e+0
   Value of the derivative of the
             function at the root:  -1.67361202918321e+0
```

Root of a Function Using the Secant Method (Secant.pas)

Description

This example uses the secant method (Gerald and Wheatley 1984, 11-13) to find a root of a user-specified real function given two initial real approximations to the root. The secant method constructs a secant through the two points specified by the initial approximations. The intersection of this line and the x-axis is used as the next best approximation to the root. The approximation to the root and its predecessor are used to construct the next secant line. The process continues until a root is approximated with specified accuracy or until a specified number of iterations have been exceeded.

User-Defined Function

```
function TNTargetF(x : Extended) : Extended;
```

The procedure *Secant* will determine the roots of this function.

Input Parameters

`InitGuess1:Extended;`	User's first approximation to the root
`InitGuess2:Extended;`	User's second approximation to the root
`Tol:Extended;`	Indicates accuracy in solution
`MaxIter:Integer;`	Maximum number of iterations permitted

The preceding parameters must satisfy the following conditions:

1. $Tol > 0$
2. $MaxIter \geq 0$

Output Parameters

`Root:Extended;`	Approximate root.
`Value:Extended;`	Value of the function at the approximate root.
`Iter:Integer;`	Number of iterations needed to find the root.
`Error:Byte;`	0: No error. 1: *Iter* > *MaxIter*. 2: The slope is zero (see "Comments"). 3: *Tol* ≤ 0. 4: *MaxIter* < 0.

If a root is found, it is returned with the value of the function at the root (which, of course, should be nearly zero). If *Error* ≤ 2, then the data from the last iteration is returned.

Syntax of the Procedure Call

```
Secant(InitGuess1, InitGuess2, Tol, MaxIter, Root, Value, Iter, Error, @TNTargetF);
```

The procedure *Secant* determines the roots of the function *TNTargetF*.

Comments

The secant algorithm constructs a line through two points and finds the intersection of that line with the x-axis. If the line has a slope whose absolute values are less than *TNNearlyZero* (that is, the two points have the same y-value), then it has no intersection with the x-axis (or infinitely many if it lies on the x-axis) and the algorithm will no longer continue. If this happens, Error 2 is returned. Error 2 will also be returned if the absolute difference of the two initial approximations (*Guess1* and *Guess2*) is less than *TNNearlyZero*.

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter.

Sample Program

The sample program Secant.pas provides I/O functions that demonstrate the secant algorithm. Note that the address of *TNTargetF* is passed to the secant procedure.

Example

Problem. Determine the solution to the equation $\cos(x) = x$.

1. Write the following code for procedure *TNTargetF* into Secant.pas:

```
{---------- HERE IS THE FUNCTION ------------}

function TNTargetF(x : Extended) : Extended;
  begin
    TNTargetF := Cos(x) - x;
  end;                 { function TNTargetF }

{------------------------------------------}
```

2. Run Secant.pas:

```
First initial approximation to the root:  0

Second initial approximation to the root: 1

Tolerance (> 0): 1E-8

Maximum number of iterations (> 0): 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
 First initial approximation:   0.00000000000000e+0
Second initial approximation:   1.00000000000000e+0
                   Tolerance:   1.00000000000000e-8
Maximum number of iterations: 100

        Number of iterations:   6
             Calculated root:   7.39085133215161e-1
       Value of the function
      at the calculated root:   0.00000000000000e+0
```

Real Roots of a Real Polynomial Equation Using the Newton-Horner Method with Deflation (Newtdefl.pas)

Description

This example uses Newton-Horner's algorithm and deflation. Newton-Horner is the Newton-Raphson method applied to polynomials (Burden and Faires 1985, 42 ff). *Deflation* is used to find several roots of a user-specified real polynomial given an initial guess specified by the user. This procedure approximates a real root and then removes the corresponding linear factor from the given polynomial. The newly obtained (deflated) polynomial is then analyzed for a real root. This process continues until a quadratic remains, the remaining roots are complex, or the algorithm is unable to approximate the remaining real roots. Should the polynomial contain two complex roots, they may be determined using the quadratic formula. You must specify (at most) the tolerance to which the roots should be approximated.

User-Defined Types

```
TNvector = array[0..TNArraySize] of Extended;

TNIntVector = array[0..TNArraySize] of Integer;
```

Input Parameters

`InitDegree:Integer;`	Degree of user-defined polynomial
`InitPoly:TNvector;`	Coefficients of user-defined polynomial
`Guess:Extended;`	User's initial approximation
`Tol:Extended;`	Indicates accuracy in solution
`MaxIter:Integer;`	Maximum number of iterations permitted

The preceding parameters must satisfy the following conditions:

1. *InitDegree* > 0
2. *Tol* > 0
3. *MaxIter* ≥ 0
4. *InitDegree* $\leq$ *TNArraySize*

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 4. If condition 4 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`Degree:Integer;`	Degree of the deflated polynomial (>2 if some of the roots are not approximated).
`NumRoots:Integer;`	Number of roots found.
`Poly:TNvector;`	Coefficients of the deflated polynomial.
`Root:TNvector;`	Real part of all roots found.
`Imag:TNvector;`	Imaginary part of all roots found (nonzero for 2 at most).
`Value:TNvector;`	Value of the polynomial at each approximate root.
`Deriv:TNvector;`	Value of the derivative at each found root.
`Iter:TNIntVector;`	Number of iterations required to find each root.
`Error:Byte;`	0: No error. 1: Maximum number of iterations exceeded. 2: The slope is zero (see "Comments"). 3: *Degree* ≤ 0. 4: *Tol* ≤ 0. 5: *MaxIter* < 0.

If a root is found, it is returned with the value of the polynomial at that root (which should be close to zero) and with the value of the derivative at that root. If the last two roots are complex (only two can be complex, since they are evaluated by the quadratic formula), then the value and derivative at those points are arbitrarily set to zero. If all the roots have not been found, then the unsolved deflated polynomial is also returned.

Syntax of the Procedure Call

```
Newt_Horn_Defl(InitDegree, InitPoly, InitGuess, Tol, MaxIter, Degree,
               NumRoots, Poly, Root, Imag, Value, Deriv, Iter, Error);
```

Comments

Newton's method involves division by the derivative of the function. Should the algorithm attempt to do any calculations at a point where the absolute values of the derivative are less than *TNNearlyZero*, the routine stops and returns an error message (Error = 2).

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter.

Sample Program

The sample program Newtdefl.pas provides I/O functions that demonstrate the Newton-deflation algorithm.

Input Files

It is possible to input the coefficients from a text file. The format for the text file is as follows:

1. The degree of the polynomial
2. The coefficients in descending order, beginning with the leading coefficient and decreasing to the constant term

Spaces or carriage returns can be used to separate the data. It does not matter whether the file ends with a carriage return; for example, the polynomial

$F(x) = x^3 - 2x$

could be entered in a text file as

3 1 0 −2 0

Example

Problem. Determine the roots to the 7th degree polynomial:

$x^6 + x^5 - 49x^4 + 69x^3 + 120x^2 + 98x - 240$

Run Newtdefl.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Degree of the polynomial (<= 30)? 6

Input the coefficients of the polynomial
where Poly[n] is the coefficient of x^n

Poly[6] =    1
Poly[5] =    1
Poly[4] =  -49
Poly[3] =   69
Poly[2] =  120
Poly[1] =   98
Poly[0] = -240

Initial approximation to the root: 0

Tolerance (> 0): 1E-8

Maximum number of iterations (>= 0): 100
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Initial Polynomial:
Poly[6]:  1.00000000000000e+0
Poly[5]:  1.00000000000000e+0
Poly[4]: -4.90000000000000e+1
Poly[3]:  6.90000000000000e+1
Poly[2]:  1.20000000000000e+2
Poly[1]:  9.80000000000000e+1
Poly[0]: -2.40000000000000e+2

       Initial approximation:  0.00000000000000e+0
                   Tolerance:  1.00000000000000e-8
Maximum number of iterations: 100


Number of calculated roots: 6
```

```
Root 1
    Number of iterations:  7
         Calculated root:  3.00000000000000e+0
   Value of the function
  at the calculated root:  3.33066907387547e-16
 Value of the derivative
      of the function at
     the calculated root: -7.48000000000000e+2


Root 2
    Number of iterations:  6
         Calculated root:  1.00000000000000e+0
   Value of the function
  at the calculated root:  3.46944695195361e-16
 Value of the derivative
      of the function at
     the calculated root:  3.60000000000000e+2


Root 3
    Number of iterations: 32
         Calculated root: -8.00000000000000e+0
   Value of the function
  at the calculated root:  0.00000000000000e+0
 Value of the derivative
      of the function at
     the calculated root: -6.43500000000000e+4


Root 4
    Number of iterations: 25
         Calculated root:  5.00000000000000e+0
   Value of the function
  at the calculated root:  0.00000000000000e+0
 Value of the derivative
      of the function at
     the calculated root:  3.84800000000000e+3


Root 5
    Number of iterations:  0
         Calculated root: -1.00000000000000e+0    +-1.00000000000000e+0   i
   Value of the function
  at the calculated root:  0.00000000000000e+0
 Value of the derivative
      of the function at
     the calculated root:  0.00000000000000e+0


Root 6
    Number of iterations:  0
         Calculated root: -1.00000000000000e+0    + 1.00000000000000e+0   i
   Value of the function
  at the calculated root:  0.00000000000000e+0
 Value of the derivative
      of the function at
     the calculated root:  0.00000000000000e+0
```

Complex Roots of a Complex Function Using Müller's Method (Muller.pas)

Description

This example uses Müller's method (Burden and Faires 1985, 71–75) to find a possibly complex root of a user-defined complex function. The algorithm finds a root of a parabola defined by three distinct points of the given function. This approximation to the root and its two predecessors are used to construct the next parabola. This is repeated until the convergence criteria is satisfied. Müller's method has the advantage of nearly always converging; however, it is slow because it uses complex arithmetic. You must create a complex function, input an initial guess (which need not be very accurate), the tolerance in the answer, and the maximum number of iterations.

User-Defined Types

```
TNcomplex = record
              Re, Im:Extended;
            end;
```

User-Defined Procedure

```
procedure TNTargetF(x:TNcomplex; var y:TNcomplex);
```

The *Muller* procedure approximates a complex root of this function.

Input Parameters

`Guess:TNcomplex;`	An initial guess
`Tol:Extended;`	Indicates accuracy in solution
`MaxIter:Integer;`	Maximum number of iterations

The preceding parameters must satisfy the following conditions:

1. *Tol* > 0
2. *MaxIter* ≥ 0

Output Parameters

`Answer:TNcomplex;`	An approximate root of the function
`yAnswer:TNcomplex;`	Value of the function at the approximate root
`Iter:Integer;`	Number of iterations required to find the root
`Error:Byte;`	0: No error 1: *Iter* > *MaxIter* 2: Parabola could not be formed (see "Comments") 3: *Tol* ≤ 0 4: *MaxIter* < 0

If *Error* ≤ 2, then the information from the last iteration is output.

Syntax of the Procedure Call

```
Muller(Guess, Tol, MaxIter, Answer, yAnswer, Iter, Error, @TNTargetF);
```

The procedure *Muller* approximates a complex root of function *TNTargetF*.

Comments

Müller's method involves constructing a parabola from three points. If they all lie on a line whose slope in absolute value is less than *TNNearlyZero*, then a parabola that intersects the x-axis cannot be constructed. Such a construction will halt the algorithm and return Error = 2. Fortunately, this does not commonly occur.

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter. Complex arithmetic is used.

Sample Program

The sample program Muller.pas provides I/O functions that demonstrate Müller's method.

The user-defined function is contained in the procedure *TNTargetF*. It is necessary to separately define the real and complex parts of the function. To define the complex function $F(x)$, you must code the following definitions:

$y.Re := Re[F(x.Re + ix.Im)];$
$y.Im := Im[F(x.Re + ix.Im)];$

where i is the square root of -1.

For example, the complex function $F(x) := \exp(x)$ would be coded like this:

$y.Re := \exp(x.Re) * \cos(x.Im);$
$y.Im := \exp(x.Re) * \sin(X.Im);$

Note that the address of *TNTargetF* is passed to the *Muller* procedure.

Example

Problem. Find a solution to the complex equation $\cos(x) = x$.

1. First, code the following procedure *TNTargetF* into Muller.pas:

```
{------------ HERE IS THE FUNCTION -----------------}

procedure TNTargetF(x : TNcomplex; var y : TNcomplex);

  begin { this is the complex function y = Cos(x) - x }
    y.Re := Cos(x.Re)*(Exp(-x.Im) + Exp(x.Im))/2 - x.Re;
    y.Im := Sin(x.Re)*(Exp(-x.Im) - Exp(x.Im))/2 - x.Im;
  end;                         { procedure TNTargetF }

{--------------------------------------------------}
```

2. Run Muller.pas:

```
Initial approximation to the root:
Re(Approximation)= -4
Im(Approximation)=  4

Tolerance (> 0): 1E-6

Maximum number of iterations (> 0): 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
       Initial approximation: -4.00000000000000e+0  + 4.00000000000000e+0   i
                   Tolerance:  1.00000000000000e-6
Maximum number of iterations: 100


    Number of iterations: 18
         Calculated root: -9.10998745393294e+0    + 2.95017086170180e+0   i
   Value of the function
  at the calculated root: -1.42544604592176e-11   + 3.75013236610100e-11  i
```

Complex Roots of a Complex Polynomial Using Laguerre's Method and Deflation (Laguerre.pas)

Description

This example uses Laguerre's method (Ralston and Rabinowitz 1978, 380–383) and linear deflation to find the possibly complex roots of a complex (or real) polynomial. You must input the coefficients of the polynomial, an initial guess, the tolerance with which to find the answer, and the maximum number of iterations.

User-Defined Types

```
TNcomplex = record
             Re, Im:Extended;
           end;

TNIntVector = array[0..TNArraySize] of Integer;

TNCompVector = array[0..TNArraySize] of TNcomplex;
```

Input Parameters

`Degree:Integer;`	Degree of the user's polynomial
`Poly:TNvector;`	Coefficients of the user's polynomial
`InitGuess:TNcomplex;`	Initial guess of the root
`Tol:Extended;`	Indicates accuracy in solution
`MaxIter:Integer;`	Maximum number of iterations

The preceding parameters must satisfy the following conditions:

1. *degree* > 0
2. *Tol* > 0
3. *MaxIter* ≥ 0
4. *degree* $\leq$ *TNArraySize*

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 4. If condition 4 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is enabled).

Output Parameters

`Degree:Integer;`	Degree of the deflated polynomial
`Poly:Integer;`	Coefficients of deflated polynomial
`NumRoots:Integer;`	Number of approximate roots
`Roots:TNCompVector;`	Approximate roots
`yRoots:TNCompVector;`	Value of the polynomial at the approximate root
`Iter:TNIntVector;`	Number of iterations required to find each root
`Error:Byte;`	0: No error 1: *Iter* $\geq$ *MaxIter* 2: *Degree* $\leq$ 0 3: *Tol* $\leq$ 0 4: *MaxIter* $<$ 0

Syntax of the Procedure Call

```
Laguerre(Degree, Poly, Guess, Tol, MaxIter, NumRoots,
        Answer, yAnswer, Iter, Error);
```

Comments

For some polynomials, certain starting values (*Guess*) will not yield convergence. If the routine does not converge to a solution, try a different starting value. Note that convergence is slower around multiple roots than around single roots.

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter.

Sample Program

The sample program Laguerre.pas provides I/O routines that demonstrate Laguerre's method.

Input Files

It is possible to input the coefficients from a text file. The format for the text file is as follows:

1. The degree of the polynomial
2. The real and imaginary parts of the coefficients in descending order, beginning with the leading coefficient and descending to the constant term

Spaces or carriage returns can be used to separate the data. It does not matter whether the file ends with a carriage return; for example, the polynomial

$$F(x) = x^4 - (2 + 2i)x^3 + 4ix^2 + (2 - 2i)x - 1$$

where i represents the square root of -1, could be entered in a text file like this:

```
4 1 0 −2 −2 0 4 2 −2 −1 0
```

Example

Problem. Find all the roots to the complex polynomial

$$F(x) = x^4 - (2 + 2i)x^3 + 4ix^2 + (2 - 2i)x - 1$$

where i is the square root of -1.

Run Laguerre.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Degree of the polynomial (<= 30)? 4

Input the complex coefficients of the polynomial
where Poly[n] is the coefficients of x^n

Re(Poly[4]) =  1
Im(Poly[4]) =  0

Re(Poly[3]) = -2
Im(Poly[3]) = -2

Re(Poly[2]) =  0
Im(Poly[2]) =  4
```

```
Re(Poly[1]) =  2
Im(Poly[1]) = -2

Re(Poly[0]) = -1
Re(Poly[0]) =  0


Initial approximation to the root:
Re(Approximation) = 1
Im(Approximation) = 0

Tolerance (> 0): 1E-6
Maximum number of iterations (> 0): 100
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Initial polynomial:

InitPoly[4]: 1.00000000000000e+0    + 0.00000000000000e+0   i
InitPoly[3]:-2.00000000000000e+0    +-2.00000000000000e+0   i
InitPoly[2]: 0.00000000000000e+0    + 4.00000000000000e+0   i
InitPoly[1]: 2.00000000000000e+0    +-2.00000000000000e+0   i
InitPoly[0]:-1.00000000000000e+0    + 0.00000000000000e+0   i


       Initial approximation:  1.00000000000000e+0    + 0.00000000000000e+0   i
                   Tolerance:  1.00000000000000e-6
Maximum number of iterations:  100



Root 1
   Number of iterations:  2
        Calculated root:  1.00000000000000e+0    + 0.00000000000000e+0   i
Value of the function at
    the calculated root:  0.00000000000000e+0    + 0.00000000000000e+0   i

Root 2
   Number of iterations:  2
        Calculated root:  1.00000000000000e+0    + 0.00000000000000e+0   i
Value of the function at
    the calculated root:  0.00000000000000e+0    + 0.00000000000000e+0   i

Root 3
   Number of iterations:  2
        Calculated root:  1.34424834689770e-10   + 9.99999999865575e-1   i
Value of the function at
    the calculated root: -1.08420217248550e-19   + 1.44222068471458e-19  i

Root 4
   Number of iterations:  2
        Calculated root:  6.71338828512027e-11   + 1.00000000013411e+0   i
Value of the function at
    the calculated root:  0.00000000000000e+0    + 3.80353570607857e-20  i
```

C H A P T E R 3

Interpolation

Interpolation is useful when some values of a function are known but others are required. For example, suppose values are known for a function $f(x)$ at $x = 2.3, 2.4, 2.5, 2.6, 2.7, 2.8$, and the value of $f(x)$ is desired at $x = 2.415$. The routines in this chapter provide the means to model to given values of $f(x)$ with an appropriate function, so that the function can be evaluated at other arbitrary points.

The goal of interpolation is to approximate the value of the function at a specified value of x, given N values of the function at N distinct points. This approximation will be a polynomial determined from the input data. The value of the polynomial at x will be returned as the approximation to the value of $f(x)$.

The *Lagrange* method accepts points in any order. The x-values need not be equally spaced. An interpolating polynomial is explicitly calculated. Although an interpolating polynomial can be useful for computing derivatives (and more), the Lagrange method is a lengthy process. Furthermore, high-degree polynomials may cause significant round-off error in some interpolations.

Newton's general divided-difference algorithm does not require input to have equally spaced x-values, nor is it necessary that the x-values be in either ascending or descending order. For large amounts of data, the divided-difference routine is more accurate than Lagrangian interpolation.

If there are many input points, the Lagrange and the divided-difference methods may result in high-degree polynomials whose oscillatory nature can produce an inaccurate approximation. This is especially true if the interpolation occurs at a value near the midpoint between adjacent input x-values. In such cases, the *cubic spline* methods are preferable.

The cubic spline methods require that the x-values be entered in ascending order. The *clamped cubic spline* method may yield more accurate results than the *free cubic spline* method but requires knowledge of the first derivative of the function at the endpoints of the input data. When this information is not available, the free cubic spline routine should be used.

The values at which interpolation is to occur should lie in the closed interval bounded by the extreme values of the input x-values. The preceding methods will not give accurate approximations to values outside this interval (extrapolation).

Polynomial Interpolation Using Lagrange's Method (Lagrange.pas)

Description

This example provides an interpolation algorithm (Burden and Faires 1985, 84 ff; Horowitz and Sahni 1984, 429-430). Given a set of N data points (x,y), the routine uses Lagrange polynomials to construct a polynomial to fit the data points. The degree of the polynomial is at most $N - 1$.

Note: The nature of high-degree polynomials may cause significant error if the algorithm is used with large amounts of data (about $N > 25$). In such cases, Divdif.pas, Cube_Fre.pas, or Cube_Cla.pas should be used. You must supply the data points and the x-values at which interpolation will take place.

User-Defined Types

```
TNvector = array[0..TNArraySize] of Extended;
TNmatrix = array[0..TNArraySize] of TNvector;
```

Input Parameters

The parameters for Lagrange:

`NumPoints:Integer;`	Number of data points
`XData:TNvector;`	The x-coordinates of the data points
`YData:TNvector;`	The y-coordinates of the data points
`NumInter:Integer;`	Number of interpolations
`XInter:TNvector`	The x-coordinates at which interpolation is to take place

The preceding parameters must satisfy the following conditions:

1. The x-coordinates of the data points (*XInter*) must be unique.
2. *NumPoints*, *NumInter* $\leq$ *TNArraySize*.
3. *NumPoints* > 0.

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the type definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`YInter:TNvector;`	The interpolated values at *XInter*
`Poly:TNvector;`	The coefficients of the interpolating polynomial
`Error:Byte;`	0: No error 1: *X*-values of the data points not unique 2: *NumPoints* < 1

Syntax of the Procedure Call

```
Lagrange(NumPoints, XData, YData, NumInter, XInter, YInter, Poly, Error);
```

Sample Program

The sample program Lagrange.pas provides I/O functions that demonstrate the Lagrange interpolating algorithm.

Input Files

Data may be entered from a text file. The *x* and *y* coordinates should be separated by a space and followed by a carriage return. For example, data values of sqr(*x*) could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Example

Problem. Construct and use an interpolating polynomial for the cosine function between $x = 1$ degree and $x = 20$ degrees.

Run Lagrange.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3A.dat
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3B.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The Data :
 1.0000000             9.99847695156391e-1
 2.0000000             9.99390827019096e-1
 3.0000000             9.98629534754574e-1
 4.0000000             9.97564050259824e-1
 5.0000000             9.96194698091746e-1
 6.0000000             9.94521895368273e-1
 7.0000000             9.92546151641322e-1
 8.0000000             9.90268068741570e-1
 9.0000000             9.87688340595138e-1
10.0000000             9.84807753012208e-1
11.0000000             9.81627183447664e-1
12.0000000             9.78147600733806e-1
13.0000000             9.74370064785235e-1
14.0000000             9.70295726275996e-1
15.0000000             9.65925826289068e-1
16.0000000             9.61261695938319e-1
17.0000000             9.56304755963035e-1
18.0000000             9.51056516295154e-1
19.0000000             9.45518575599317e-1
20.0000000             9.39692620785908e-1
```

```
The polynomial :
Poly[19]=-1.72014247146006e-28
Poly[18]= 3.54986012534706e-26
Poly[17]=-3.41072664146385e-24
Poly[16]= 2.02588035084664e-22
Poly[15]=-8.33028761905346e-21
Poly[14]= 2.51630894794110e-19
Poly[13]=-5.78243038713688e-18
Poly[12]= 1.03284343326638e-16
Poly[11]=-1.45263304267538e-15
Poly[10]= 1.61970333747745e-14
Poly[ 9]=-1.43449305975914e-13
Poly[ 8]= 1.00656254399833e-12
Poly[ 7]=-5.55641265799623e-12
Poly[ 6]= 2.37976717179018e-11
Poly[ 5]=-7.79913921901990e-11
Poly[ 4]= 4.05555790625022e-9
Poly[ 3]=-3.26288947218059e-10
Poly[ 2]=-1.52308336619420e-4
Poly[ 1]=-2.49984780967393e-10
Poly[ 0]= 1.00000000007260e+0

     X              Interpolated Y value
   1.500            9.99657324975254e-1
   2.500            9.99048221581889e-1
   3.500            9.98134798421861e-1
   4.500            9.96917333733130e-1
   5.500            9.95396198367178e-1
   6.500            9.93571855676588e-1
   7.500            9.91444861373810e-1
   8.500            9.89015863361917e-1
   9.500            9.86285601537232e-1
  10.500            9.83254907563954e-1
  11.500            9.79924704620830e-1
  12.500            9.76296007119933e-1
  13.500            9.72369920397676e-1
  14.500            9.68147640378107e-1
  15.500            9.63630453208623e-1
  16.500            9.58819734868193e-1
  17.500            9.53716950748227e-1
  18.500            9.48323655206198e-1
  19.500            9.42641491092216e-1
  20.500            9.36672189246619e-1
```

The data is taken from a function of which the derivative could be computed exactly.

Interpolation Using Newton's Interpolary Divided-Difference Method (Divdif.pas)

Description

This example provides an interpolation algorithm. Given a set of data points (x,y), the routine uses Newton's interpolary divided-difference equation to interpolate between the points (Burden and Faires 1985, 100–102). The data points must have unique x-values, but these values need not be evenly spaced nor set in any particular order. You must supply the data points and the x-values at which interpolation is to take place.

User-Defined Types

```
TNvector = array[0..TNArraySize] of Extended;

TNmatrix = array[0..TNArraySize] of TNvector;
```

Input Parameters

`NumPoints:Integer;`	Number of data points
`XData:TNvector;`	The x-coordinates of the data points
`YData:TNvector;`	The y-coordinates of the data points
`NumInter:Integer;`	Number of interpolations
`XInter:TNvector`	The x-coordinates at which interpolation is to take place

The preceding parameters must satisfy the following conditions:

1. The x-coordinates of the data points (*XInter*) must be unique.
2. *NumPoints*, *NumInter* $\leq$ *TNArraySize*.
3. *NumPoints* > 0.

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`YInter:TNvector;`	The interpolated values at *XInter*
`Error:Byte;`	0: No error
	1: *X*-values of the data points not unique
	2: *NumPoints* < 1

Syntax of the Procedure Call

```
Divided_Difference(NumPoints, XData, YData, NumInter, XInter, YInter, Error);
```

Sample Program

The sample program Divdif.pas provides I/O functions that demonstrate Newton's interpolary divided-difference algorithm.

Input Files

Data may be entered from a text file. The x and y coordinates should be separated by a space and followed by a carriage return. For example, data values of sqr(x) could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Example

Problem. Interpolate the cosine function between $x = 1x$ and $x = 20x$.

Run Divdif.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3C.dat
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Number of points (0-50)?15

Point  1:  1.5
Point  2:  2.5
Point  3:  3.5
Point  4:  4.5
Point  5:  5.5
Point  6:  6.5
Point  7:  7.5
Point  8:  8.5
Point  9:  9.5
Point 10: 10.5
Point 11: 11.5
Point 12: 12.5
Point 13: 13.5
Point 14: 14.5
Point 15: 15.5
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
    X                    Y
12.000                0.9781476
 8.000                0.9902681
 1.000                0.9998477
10.000                0.9848078
 5.000                0.9961947
15.000                0.9659258
 4.000                0.9975641
 3.000                0.9986295
 7.000                0.9925462
14.000                0.9702957

    X                Interpolated Y value
 1.500               9.99656668284607e-1
 2.500               9.99047982204853e-1
 3.500               9.98134846782587e-1
 4.500               9.96917355869352e-1
 5.500               9.95396200633579e-1
 6.500               9.93571893532269e-1
 7.500               9.91444906399794e-1
 8.500               9.89015879894104e-1
 9.500               9.86285623948171e-1
10.500               9.83254980952454e-1
11.500               9.79924765142406e-1
12.500               9.76295923083642e-1
13.500               9.72369781236267e-1
14.500               9.68147757339141e-1
15.500               9.63629212784400e-1
```

The data is taken from a function of which the derivative could be computed exactly.

Free Cubic Spline Interpolation (Cube_Fre.pas)

Description

This example constructs a smooth curve through a given set of data points. The curve is a cubic spline interpolant with the following properties:

1. It passes through every data point.
2. It is continuous.
3. Its first derivative is continuous.
4. Its second derivative is continuous.

The second derivative is assumed to be zero at both endpoints (thus the cubic spline is "free") of the interval determined by the data (Burden and Faires 1985, 117 ff). Cubics that join adjacent data points are of the following form:

$$S[i](x) = \mathrm{Coef0}[i] + \mathrm{Coef1}[i](x - x[i]) + \mathrm{Coef2}[i](x - x[i])^2 + \mathrm{Coef3}[i](x - x[i])^3$$

where i ranges between 1 and the number of data points minus 1, the $x[i]$'s are the x-coordinates of the input data, and $x[i] \le x < x[i+1]$. The interpolated values of $f(x)$ are found by evaluating the ith cubic polynomial at x, where

$$x[i] \le x \le x[i + 1].$$

User-Defined Types

```
TNvector = array[0..TNArraySize] of Extended;
```

Input Parameters

`NumPoints:Integer;`	Number of data points
`XData:TNvector;`	The x-coordinates of the data points
`YData:TNvector;`	The y-coordinates of the data points
`NumInter:Integer;`	Number of interpolations
`XInter:TNvector;`	X-coordinates of points at which to interpolate

The preceding parameters must satisfy the following conditions:

1. *X* data points must be unique.
2. *X* data points must be in ascending order.
3. *NumPoints*, *NumInter* $\leq$ *TNArraySize*.
4. *NumPoints* $>$ 1.

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 3. If condition 3 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`Coef0:TNvector;`	Coefficient of the constant term
`Coef1:TNvector;`	Coefficient of the linear term
`Coef2:TNvector;`	Coefficient of the squared term
`Coef3:TNvector;`	Coefficient of the cubed term
`YInter:TNvector;`	Interpolated values at *XInter*
`Error:Byte;`	0: No error 1: *X*-values of the data points not unique 2: *X*-values of the data points not in ascending order 3: *NumPoints* $<$ 2

Syntax of the Procedure Call

```
CubicSplineFree(NumPoints, XData, YData, NumInter, XInter,
                Coef0, Coef1, Coef2, Coef3, YInter, Error);
```

Sample Program

The sample program Cube_Fre.pas provides I/O functions that demonstrate the free cubic spline algorithm.

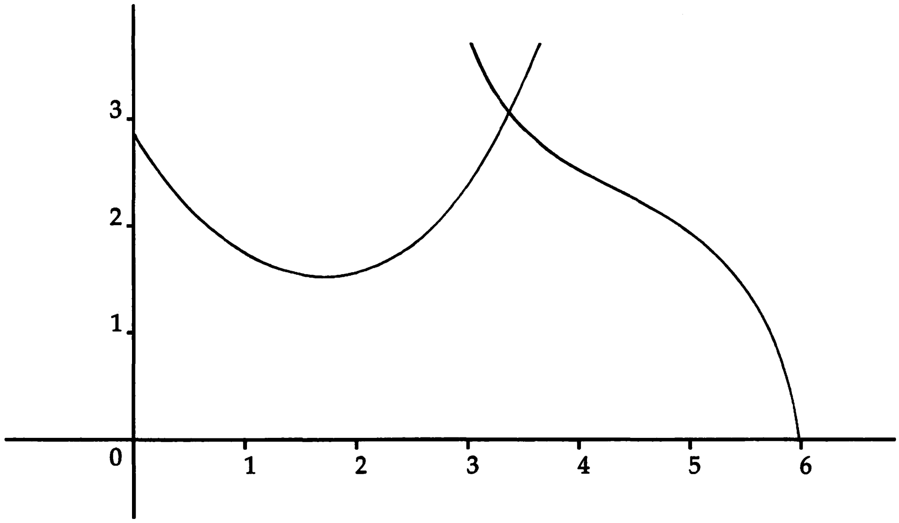
Input Files

Data may be entered from a text file. The x and y coordinates should be separated by a space and followed by a carriage return. For example, data values of sqr(x) could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Example

Problem. Construct an interpolating spline for the following figure:



Because a cusp occurs at $x = 3.55$, we will construct two splines, one for each side of the cusp.

Run Cube_Fre.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3D.dat
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3E.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Data :           X                 Y
  1:        0.0000000000      2.8000000000
  2:        0.1000000000      2.7000000000
  3:        0.2000000000      2.6000000000
  4:        0.6000000000      2.2000000000
  5:        1.0000000000      1.8000000000
  6:        1.4000000000      1.6000000000
  7:        1.8000000000      1.4000000000
  8:        2.0000000000      1.4200000000
  9:        2.2000000000      1.4000000000
 10:        2.6000000000      1.5000000000
 11:        3.0000000000      1.8000000000
 12:        3.4000000000      2.4000000000
 13:        3.4500000000      2.6000000000
 14:        3.5000000000      2.8000000000
 15:        3.5500000000      2.9000000000

Splines:       Coef0           Coef1           Coef2            Coef3
    1:     2.8000000000   -0.9988332302    0.0000000000    -0.1166769808
    2:     2.7000000000   -1.0023335396   -0.0350030942     0.5833849040
    3:     2.6000000000   -0.9918326113    0.1400123770    -0.4010771215
    4:     2.2000000000   -1.0723397281   -0.3412801689     1.3053237227
    5:     1.8000000000   -0.7188084763    1.2251082984    -1.6952177695
    6:     1.6000000000   -0.5524263669   -0.8091530249     2.3505473551
    7:     1.4000000000   -0.0714860563    2.0115038012    -5.7703675978
    8:     1.4200000000    0.0406713524   -1.4507167575     3.7367999767
    9:     1.4000000000   -0.0911993534    0.7913632286     0.1540878869
   10:     1.5000000000    0.6158534153    0.9762686929    -1.6022555777
   11:     1.8000000000    0.6277856923   -0.9464380003     7.8174344240
   12:     2.4000000000    3.6230038155    8.4344833084   -17.8911923822
   13:     2.6000000000    4.3322682035    5.7508044511  -247.9233704257
   14:     2.8000000000    3.0479233704  -31.4377011128   209.5846740851

Interpolated Points:   X                 Y
        1:        0.3000000000      2.5018157855
        2:        0.5000000000      2.3042222482
        3:        1.2000000000      1.6916808945
        4:        1.6000000000      1.4759529845
        5:        2.1000000000      1.4132967676
        6:        2.3000000000      1.3989477848
        7:        2.5000000000      1.4480232575
        8:        2.7000000000      1.5697457729
        9:        2.9000000000      1.7293593063
       10:        3.2000000000      1.9502390938
       11:        3.3000000000      2.1142270171
```

Second half of the figure:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3F.dat
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3G.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Data :           X                  Y
  1:        3.5500000000       2.9000000000
  2:        3.6000000000       2.8000000000
  3:        3.6500000000       2.6500000000
  4:        3.8000000000       2.5000000000
  5:        4.0000000000       2.3500000000
  6:        4.3000000000       2.2000000000
  7:        4.8000000000       1.9500000000
  8:        5.3000000000       1.6000000000
  9:        5.6000000000       1.3000000000
 10:        5.8000000000       1.2000000000
 11:        6.0000000000       0.0000000000

Splines:      Coef0           Coef1            Coef2              Coef3
    1:     2.9000000000    -1.6719664279     0.0000000000    -131.2134288293
    2:     2.8000000000    -2.6560671441   -19.6820143244     256.0671441466
    3:     2.6500000000    -2.7037649955    18.7280572976     -49.1308266290
    4:     2.5000000000    -0.4016786037    -3.3808146854       8.1960385189
    5:     2.3500000000    -0.7704798556     1.5368084259      -2.1173630243
    6:     2.2000000000    -0.4200828166    -0.3688182960       0.4179678583
    7:     1.9500000000    -0.4754252188     0.2581334916      -1.4145661079
    8:     1.6000000000    -1.2782163082    -1.8637156703       9.3036778805
    9:     1.3000000000     0.1155473174     6.5095944222     -47.9366550462
   10:     1.2000000000    -3.0330135193   -22.2523986055      37.0873310092

Interpolated Points:    X                  Y
          1:       3.7000000000       2.5554905401
          2:       3.9000000000       2.4342200313
          3:       4.1000000000       2.2862027357
          4:       4.2000000000       2.2404374617
          5:       4.5000000000       2.1045744477
          6:       4.6000000000       2.0520666406
          7:       5.0000000000       1.8539237670
          8:       5.2000000000       1.7105990402
          9:       5.5000000000       1.3442375346
         10:       5.7000000000       1.3287140209
         11:       5.9000000000       0.7112619930
```

Clamped Cubic Spline Interpolation (Cube_Cla.pas)

Description

This example constructs a smooth curve through a given set of data points. The curve is a cubic spline interpolant with the following properties:

1. It passes through every data point.
2. It is continuous.
3. Its first derivative is continuous.
4. Its second derivative is continuous.

The first derivative at the endpoints of the interval determined by the input data is defined by the user (Burden and Faires 1985, 122 ff.). (This is what makes the cubic spline "clamped.") The cubics that join adjacent data points are of the following form:

$$S[i](x) = \text{Coef0}[i] + \text{Coef1}[i](x - x[i]) + \text{Coef2}[i](x - x[i])^2 + \text{Coef3}[i](x - x[i])^3$$

where i ranges between 1 and the number of data points minus 1, the $x[i]$'s are the x-coordinates of the input data, and $x[i] \leq x < x[i + 1]$. The interpolated values of $f(x)$ are found by evaluating the ith cubic polynomial at x, where $x[i] \leq x \leq x[i + 1]$.

User-Defined Types

```
TNvector = array[0..TNArraySize] of Extended;
```

Input Parameters

`NumPoints:Integer;`	Number of data points
`XData:TNvector;`	The x-coordinates of the data points
`YData:TNvector;`	The y-coordinates of the data points
`DerivLE:Extended;`	Derivative of the function at the left endpoint
`DerivRE:Extended;`	Derivative of the function at the right endpoint

`NumInter:Integer;`	Number of interpolations
`XInter:TNvector;`	*X*-coordinates of points at which to interpolate

The preceding parameters must satisfy the following conditions:

1. *X* data points must be unique.
2. *X* data points must be in ascending order.
3. *NumPoints, NumInter* $\leq$ *TNArraySize.*
4. *NumPoints* $>$ *1.*

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the ***type*** definition of *TNvector. TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 3. If condition 3 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`Coef0:TNvector;`	Coefficient of the constant term
`Coef1:TNvector;`	Coefficient of the linear term
`Coef2:TNvector;`	Coefficient of the squared term
`Coef3:TNvector;`	Coefficient of the cubed term
`YInter:TNvector;`	Interpolated values at *XInter*
`Error:Byte;`	0: No error 1: *X*-values of the data points not unique 2: *X*-values of the data points not in ascending order 3: *NumPoints* < 2

Syntax of the Procedure Call

```
CubicSplineClamped(NumPoints, XData, YData, DerivLE, DerivRE, NumInter,
                   XInter, Coef0, Coef1, Coef2, Coef3, YInter, Error);
```

Sample Program

The sample program Cube_Cla.pas provides I/O functions that demonstrate the clamped cubic spline interpolation algorithm.

Input Files

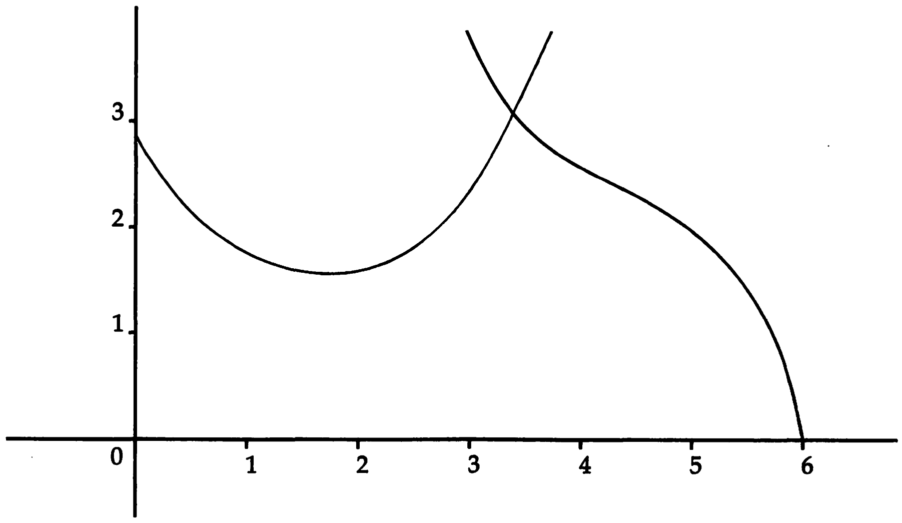
Data may be entered from a text file. The x- and y-coordinates should be separated by a space and followed by a carriage return. The last two values in the file must be the derivatives of the function at the endpoints. For example, data values of sqr(x) could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
2 10
```

Note that the last two values are the derivatives of sqr(x) at the endpoints $x = 1$ and $x = 5$.

Example

Problem. Construct an interpolating spline for the following figure:



Because a cusp occurs at $x = 3.55$, we will construct two splines, one for each side of the cusp.

Run Cube_Cla.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3H.dat
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3E.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Data :            X                 Y
   1:        0.0000000000      2.8000000000
   2:        0.1000000000      2.7000000000
   3:        0.2000000000      2.6000000000
   4:        0.6000000000      2.2000000000
   5:        1.0000000000      1.8000000000
   6:        1.4000000000      1.6000000000
   7:        1.8000000000      1.4000000000
   8:        2.0000000000      1.4200000000
   9:        2.2000000000      1.4000000000
  10:        2.6000000000      1.5000000000
  11:        3.0000000000      1.8000000000
  12:        3.4000000000      2.4000000000
  13:        3.4500000000      2.6000000000
  14:        3.5000000000      2.8000000000
  15:        3.5500000000      2.9000000000

Derivative at X= 0.00000000000000e+0     :  -1.33333333333333e+0
Derivative at X= 3.55000000000000e+0     :   3.00000000000000e+0

Splines:         Coef0            Coef1             Coef2            Coef3
     1:      2.8000000000    -1.3333333333     5.7579845570   -24.2465122365
     2:      2.7000000000    -0.9091317890    -1.5159691140     6.0728700429
     3:      2.6000000000    -1.0301395105     0.3058918989    -0.5763578064
     4:      2.2000000000    -1.0620777385    -0.3857374687     1.3523295373
     5:      1.8000000000    -0.7215495356     1.2370579761    -1.7079603429
     6:      1.6000000000    -0.5517241193    -0.8124944355     2.3545118344
     7:      1.4000000000    -0.0715539872     2.0129197658    -5.7757491499
     8:      1.4200000000     0.0405240212    -1.4525297241     3.7495480911
     9:      1.4000000000    -0.0905420975     0.7971991306     0.1353902832
    10:      1.5000000000     0.6122045428     0.9596674704    -1.5379470688
    11:      1.8000000000     0.6417239262    -0.8858690121     7.5788979919
    12:      2.4000000000     3.5708997526     8.2088085781     7.4639274157
    13:      2.6000000000     4.4477600660     9.3283976905  -365.6719802043
    14:      2.8000000000     2.6380599835   -45.5223993401   655.2239934014
```

```
Interpolated Points:    X                 Y
        1:        0.3000000000      2.4994686101
        2:        0.5000000000      2.3029267570
        3:        1.2000000000      1.6915087292
        4:        1.6000000000      1.4759914934
        5:        2.1000000000      1.4132766530
        6:        2.3000000000      1.3990531718
        7:        2.5000000000      1.4482408301
        8:        2.7000000000      1.5692791819
        9:        2.9000000000      1.7285068643
       10:        3.2000000000      1.9535412087
       11:        3.3000000000      2.1174192125
```

Second half of figure:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3I.dat
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample3G.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Data :          X                Y
  1:      3.5500000000      2.9000000000
  2:      3.6000000000      2.8000000000
  3:      3.6500000000      2.6500000000
  4:      3.8000000000      2.5000000000
  5:      4.0000000000      2.3500000000
  6:      4.3000000000      2.2000000000
  7:      4.8000000000      1.9500000000
  8:      5.3000000000      1.6000000000
  9:      5.6000000000      1.3000000000
 10:      5.8000000000      1.2000000000
 11:      6.0000000000      0.0000000000

Derivative at X= 3.55000000000000e+0    :  -4.00000000000000e+0
Derivative at X= 6.00000000000000e+0    :  -1.70000000000000e+1

Splines:       Coef0           Coef1            Coef2            Coef3
    1:    2.9000000000    -4.0000000000    80.2233303937   -804.4666078741
    2:    2.8000000000    -2.0111665197   -40.4466607874    413.3998236224
    3:    2.6500000000    -2.9553339213    21.5633127559    -56.8516885392
    4:    2.5000000000    -0.3238290709    -4.0199470867      9.4454622054
    5:    2.3500000000    -0.7983524409     1.6473302365     -2.1760736673
    6:    2.2000000000    -0.3974941891    -0.3111360640      0.2122488846
    7:    1.9500000000    -0.5494435897     0.0072372629     -0.6167001671
    8:    1.6000000000    -1.0047314521    -0.9178129877      3.1119483153
    9:    1.3000000000    -0.7151931996     1.8829404961     -4.0348724916
   10:    1.2000000000    -0.4462017001    -0.5379829989   -136.1550425028
```

```
Interpolated Points:    X                 Y
        1:         3.7000000000      2.5490351248
        2:         3.9000000000      2.4368630843
        3:         4.1000000000      2.2844619846
        4:         4.2000000000      2.2388141319
        5:         4.5000000000      2.1097537107
        6:         4.6000000000      2.0584802174
        7:         5.0000000000      1.8354671712
        8:         5.2000000000      1.6919117155
        9:         5.5000000000      1.3872367766
       10:         5.7000000000      1.2432752125
       11:         5.9000000000      1.0138449575
```

CHAPTER 4

Numerical Differentiation

Differentiation is a process used in calculus to quantify the rate of change of a given function. The ***derivative*** of a real-valued function of a real variable is another real-valued function of a real variable. For example, suppose you are driving down the freeway in your car and $f(t)$ gives the distance traveled at time t. Typical values might be

t	f(t)
1.0	45.0
1.1	49.2
1.2	54.5
1.3	59.8
1.4	65.1
1.5	70.4

The units are in hours and miles, and the data refers to a trip that started at noon. $f(1.0) = 45.0$, so the distance traveled by one o'clock is 45.0 miles, and $f(1.5) = 70.4$, so by half past one you will be 70.4 miles from where you were at noon.

The derivative of this distance function gives the velocity function. The car's velocity at one o'clock is the value of the derivative at $t = 1.0$. From the previous data, it is impossible to compute the derivative exactly, but it is possible to approximate the derivative. The car traveled $49.2 - 45.0 = 4.2$ miles in the six minutes after one o'clock ($1.1 - 1.0 = 0.1$ hours = 6 minutes). Thus, the average velocity of the car during those six minutes is $4.2 / 0.1 = 42$ miles per hour. This gives an approximation to the velocity at one o'clock.

Each method described in this chapter approximates derivatives of a real function of one real variable.

The routines Deriv.pas, Deriv2.pas, and Interdrv.pas compute derivatives of a function that is represented by tabular data. Consequently, their accuracy depends heavily upon the precision and spacing of the data points.

The routines Derivfn.pas and Deriv2fn.pas compute derivatives of a user-defined function. Consequently, the accuracy of the values calculated with these routines is limited by the precision of the computer.

Differentiation consists of subtracting two very close numbers and dividing by a very small number; hence, it is extremely sensitive to round-off error. The accuracy of the first derivative is approximately the square root of the precision with which real numbers are represented; the accuracy of the second derivative is approximately equal to the fourth root.

The first derivative of a function that is represented by a table of values can be approximated in Deriv.pas via a two-point formula, a three-point formula, or a five-point formula. The accuracy of the formula increases with the number of points used in the formula. In order to use the five-point formula, however, the domain values of the data points (that is, the x-coordinates) must be equally spaced. This is not required for the two-point and three-point formulas. Derivatives can only be approximated at data points.

The second derivative of a function that is represented by a table of values can be approximated in Deriv2.pas via a three-point formula or a five-point formula. The domain values of the data points must be equally spaced (regardless of whether the three-point formula or five-point formula is used). Second derivatives can only be approximated at data points.

The routine Interdrv.pas approximates a function by constructing a *free cubic spline* to a set of data points. Cubic splines avoid the undesirable oscillatory behavior of other interpolating polynomials. The derivative of the cubic spline at a given domain value, which may be different from the input data values, will then approximate the corresponding derivative of the function.

The first derivative of a user-supplied function is approximated in Derivfn.pas via a three-point formula. The approximation is refined with *Richardson extrapolation*. The derivative can be approximated at any point within the domain of the function.

The second derivative of a user-supplied function is approximated in Deriv2fn.pas via a three-point formula. The approximation is refined with Richardson extrapolation. The second derivative can be approximated at any point within the domain of the function.

First Differentiation Using Two-Point, Three-Point, or Five-Point Formulas (Deriv.pas)

Description

This example contains several algorithms for approximating the derivative of a function $f(x)$, given several data points $(x, f(x))$. The user must specify whether a two-point, three-point, or five-point formula should be used. Two points are used in the two-point formula, three in the three-point formula, and five in the five-point formula. The user must supply the data points $(x, f(x))$ and the x-values of the data points at which to approximate the derivative. **Note:** Derivatives can only be approximated at x-values corresponding to input data points.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

Input Parameters

`NumPoints : Integer;`	Number of data points
`XData : TNvector;`	X-coordinates of data points
`YData : TNvector;`	Y-coordinates of data points
`Point : Byte;`	Two-point, three-point, or five-point differentiation
`NumDeriv : Integer;`	Number of points at which the derivative is to be approximated
`XDeriv : TNvector;`	X-coordinates of data points at which the derivative is to be approximated

The preceding parameters must satisfy the following conditions:

1. *XData* points must be unique.
2. *XData* points must be entered in ascending order.
3. At least two points are needed for two-point differentiation, three for three-point differentiation, and five for five-point differentiation.
4. *Point* must equal two, three, or five.

5. *XData* points must be equally spaced for five-point differentiation.
6. *XDeriv* points must be a subset of the *XData* points.
7. *NumPoints*, *NumDeriv* $\leq$ *TNArraySize*.

TNArraySize represents the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 7. If condition 7 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`YDeriv : TNvector;`	Approximation to the first derivative at the points in *XDeriv*
`Error : Byte;`	0: No errors 1: WARNING! Not all the derivatives were computed (see "Comments") 2: *X*-values not unique 3: *X*-values not in ascending order 4: Not enough data 5: *Point* not equal to 2, 3, or 5 6: *X*-values not equally spaced for the five-point formula

Syntax of the Procedure Call

```
First_Derivative(NumPoints, XData, YData, Point, NumDeriv, XDeriv, YDeriv, Error);
```

Comments

If an x-value at which the derivative is to be approximated is not among the data points, the value $-9.999999999E35$ is arbitrarily assigned to the derivative at that point and Error = 1 is returned. When using five-point differentiation with only five points, there is not enough information to approximate the derivative at the first, second, fourth, or fifth points. Likewise, if only six points are input, there is insufficient information for approximating the derivative at the second and fifth data points. Should an attempt be made to approximate the derivative at any of these points, the value of 9.999999999E35 is arbitrarily assigned the derivative at that point and Error = 1 is returned.

Sample Program

The sample program Deriv.pas provides I/O functions that demonstrate differentiation with two-point, three-point, and five-point formulas.

Input Files

Data points may be entered from a text file. The x- and y-coordinates should be separated by a space and followed by a carriage return. For example, data values of sqr(x) could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Derivative points may also be entered from a text file. Every derivative point must be followed by a carriage return. For example, to determine the derivatives of the preceding points, create the following file of derivative points:

```
1
2
3
4
5
```

Example

Problem. Approximate the first derivative of $f(x)$ = sqr(x) * cos(x) at several points between one and two radians. The output from three runs is given. Actual values of the derivatives to eight significant figures are also given.

Run Deriv.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample4A.dat
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Number of X values (0-100)? 5

Point 1: 1.1
Point 2: 1.3
Point 3: 1.5
Point 4: 2.0
Point 5: 2.2

2-, 3-, or 5-point differentiation ? 2
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Input Data:
   X                      Y
  1.0000000          5.40302305868140e-1
  1.1000000          5.48851306924949e-1
  1.2000000          5.21795166446410e-1
  1.3000000          4.52073020375553e-1
  1.4000000          3.33135600084472e-1
  1.5000000          1.59158703752332e-1
  1.6000000         -7.47507770912994e-2
  1.7000000         -3.72360588514066e-1
  1.8000000         -7.36134786805602e-1
  1.9000000         -1.16707533637725e+0
  2.0000000         -1.66458734618857e+0

                <* -------------------------- *>
                <*          WARNING           *>
                <* -------------------------- *>

Using 2-point differentiation:

   X              Derivative at X
  1.100         8.54900105680900e-2
  1.300        -6.97221460708570e-1
  1.500        -1.73976896332140e+0
  2.000        -4.97512009811320e+0
  2.200        No derivative calculated

Using 3-point differentiation:

   X              Derivative at X
  1.100        -9.25356971086500e-2
  1.300        -9.43297831809690e-1
  1.500        -2.03943188587886e+0
  2.000        -5.30797739931156e+0
  2.200        No derivative calculated
```

```
Using 5-point differentiation:

     X                Derivative at X
   1.100          -8.08749392678308e-2
   1.300          -9.32986606435739e-1
   1.500          -2.03221450709713e+0
   2.000          -5.30200229054730e+0
   2.200          No derivative calculated
```

The data is taken from a function of which a derivative could be computed exactly.

The warning signal indicates that some derivatives were not calculated.

The derivative is not approximated for $x = 2.2$ in any of the examples because $x = 2.2$ is not among the data points.

Second Differentiation Using Three-Point or Five-Point Formulas (Deriv2.pas)

Description

This example contains two algorithms that approximate the second derivative of a function $f(x)$ when several data points $(x, f(x))$ are specified. You decide whether to use a three-point or five-point formula (Gerald and Wheatley 1984, 236–237); three points are used in the three-point formula, and five in the five-point formula. You must supply the data points $(x, f(x))$ and the x-values of the data points at which the second derivative is to be approximated. The second derivative may only be approximated at x-values that were input as data points.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

Input Parameters

`NumPoints : Integer;`	Number of data points
`XData : TNvector;`	X-coordinates of the data points
`YData : TNvector;`	Y-coordinates of the data points
`Point : Byte;`	Three-point or five-point differentiation
`NumDeriv : Integer;`	Number of points at which the derivative is to be approximated
`XDeriv : TNvector;`	X-coordinates of points at which the derivative is to be approximated

The preceding parameters must satisfy the following conditions:

1. *XData* points must be unique.
2. *XData* points must be entered in ascending order.
3. At least three points for three-point differentiation and five points for five-point differentiation.
4. *Point* must equal 3 or 5.

5. *XData* points must be equally spaced.
6. *XDeriv* points must be a subset of the *XData* points.
7. *NumPoints*, *NumDeriv* $\leq$ *TNArraySize*.

TNArraySize represents the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 7. If condition 7 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`YDeriv : TNvector;`	Approximation to the second derivative at the *XDeriv* points
`Error : Byte;`	0: No errors 1: WARNING! At least one derivative was not approximated (see "Comments") 2: *X*-values not unique 3: *X*-values not in increasing order 4: Not enough data 5: *Point* not equal to 3 or 5 6: *X*-value points not equally spaced

Syntax of the Procedure Call

```
Second_Derivative(NumPoints, XData, YData, Point, NumDeriv, XDeriv, YDeriv, Error);
```

Comments

If an x-value at which the second derivative is approximated is not among the data points, the value $-9.9999999E35$ is arbitrarily assigned to the derivative at that point and Error = 1 is returned. When using five-point second differentiation with only five data points, there is insufficient information for approximating the second derivative at the second and fourth data points. Should an attempt be made to approximate the second derivative at these points, the value 9.9999999E35 is arbitrarily assigned to the second derivative at that point and Error = 1 is returned.

Sample Program

The sample program Deriv2.pas provides I/O functions that demonstrate second-order differentiation with three-point and five-point formulas.

Input Files

Data points may be entered from a text file. The x- and y-coordinates should be separated by a space and followed by a carriage return. For example, data values of sqr(x) could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Derivative points may also be entered from a text file. Every derivative point must be followed by a carriage return. For example, to determine the second derivatives of the preceding points, create the following file of derivative points:

```
1
2
3
4
5
```

Example

Problem. Approximate the second derivative of $f(x)$ = sqr(x) * cos(x) at several points between $x = 1$ and $x = 2$ radians. The output from two runs is given. Actual values of the second derivatives to eight significant figures are also given.

Run Deriv2.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample4A.dat
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Number of X values (0-100)?5

Point 1: 1.1
Point 2: 1.3
Point 3: 1.5
Point 4: 2.0
Point 5: 2.2

3- or 5-point second differentiation ? 3
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Input Data:
    X                   Y
  1.0000000         5.40302305868140e-1
  1.1000000         5.48851306924949e-1
  1.2000000         5.21795166446410e-1
  1.3000000         4.52073020375553e-1
  1.4000000         3.33135600084472e-1
  1.5000000         1.59158703752332e-1
  1.6000000        -7.47507770912994e-2
  1.7000000        -3.72360588514066e-1
  1.8000000        -7.36134786805602e-1
  1.9000000        -1.16707533637725e+0
  2.0000000        -1.66458734618857e+0

               <* --------------------------- *>
               <*           WARNING           *>
               <* --------------------------- *>

Using 3-point second differentiation:

    X           Second Derivative at X
  1.100         -3.56051415353480e+0
  1.300         -4.92152742202240e+0
  1.500         -5.99325845114914e+0
  2.000         -6.65714602396720e+0
  2.200         No 2nd derivative calculated.

Using 5-point second differentiation:

    X           Second Derivative at X
  1.100         -3.61167369644120e+0
  1.300         -4.92756964541466e+0
  1.500         -6.00263647117238e+0
  2.000         -6.59765691992320e+0
  2.200         No 2nd derivative calculated.
```

The data is taken from a function of which the derivative could be computed exactly.

The warning signal indicates that some second derivatives were not calculated.

The second derivative is not approximated at $x = 2.2$ for either run because $x = 2.2$ is not among the input x-value points.

Differentiation with a Cubic Spline Interpolant (Interdrv.pas)

Description

This example contains an algorithm for approximating the first and second derivatives of a function given several data points $(x, f(x))$. The algorithm assumes that a free cubic spline interpolant (Burden and Faires 1985, 117–122) is an adequate approximation to the function $f(x)$, so that the slope of the interpolant at any value x_i is an adequate approximation to $f'(x_i)$. See Chapter 3 (Cube_Fre.pas) for more information on free cubic splines. The user must supply the data points $(x, f(x))$ and the x-values at which to approximate the derivatives. Derivatives may be approximated at any x-value contained in the closed interval determined by the data points. This routine will likely give significant errors if interpolation (Gerald and Wheatley 1984, 227–231) is attempted outside the range of x-values (extrapolation).

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

Input Parameters

`NumPoints : Integer;`	Number of data points
`XData : TNvector;`	X-coordinates of data points
`YData : TNvector;`	Y-coordinates of data points
`NumDeriv : Integer;`	Number of points at which the derivative is to be approximated
`XDeriv : TNvector;`	X-coordinates of points at which the derivative is to be approximated

The preceding parameters must satisfy the following conditions:

1. *XData* points must be unique.
2. *XData* points must be in ascending order.
3. *NumPoints* ≥ 2.
4. *NumPoints*, *NumDeriv* $\leq$ *TNArraySize*.

TNArraySize represents the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 4. If condition 4 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`YInter : TNvector;`	Interpolated y-values at the *XDeriv* points
`YDeriv : TNvector;`	Approximation to the first derivative at the x-values in *XDeriv*
`YDeriv2 : TNvector;`	Approximation to the second derivative at the x-values in *XDeriv*
`Error : Byte;`	0: No errors 1: X-values not unique 2: X-values not in ascending order 3: *NumPoints* < 2

Syntax of the Procedure Call

```
Interpolate_Derivative(NumPoints, XData, YData, NumDeriv,
                       XDeriv, YInter, YDeriv, YDeriv2, Error);
```

Sample Program

The sample program Interdrv.pas provides I/O functions that demonstrate differentiation with a cubic spline interpolant.

Input Files

Data points may be entered from a text file. The x- and y-coordinates should be separated by a space and followed by a carriage return. For example, data values of sqr(x) could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Derivative points may also be entered from a text file. Every derivative point must be followed by a carriage return. For example, to determine the derivatives of the preceding points, create the following file of derivative points:

```
1
2
3
4
5
```

Example

Problem. Determine the first and second derivative of $f(x)$ = sqr(x) $*$ cos(x) at several points between one and two radians. Actual values of the derivatives to eight significant figures are given here.

Run Interdrv.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample4B.dat
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Number of derivative points (0-100)?5

Point 1: 1.1
Point 2: 1.3
Point 3: 1.55
Point 4: 1.95
Point 5: 2.20
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Input Data:
    X                  Y
  1.000            0.5403023
  1.100            0.5488513
  1.200            0.5217952
  1.300            0.4520730
  1.400            0.3331356
  1.500            0.1591587
  1.600           -0.0747508
  1.700           -0.3723606
  1.800           -0.7361348
  1.900           -1.1670753
  2.000           -1.6645873

Using free cubic spline interpolation:

 X             Value at X            1st Deriv at X          2nd Deriv at X
 1.100          5.48851300000000e-1  -5.86015666816464e-2    -4.32274700
 1.300          4.52073000000000e-1  -9.31377366861403e-1    -4.98862501
 1.550          4.99429267146237e-2  -2.33770918101853e+0    -6.19118137
 1.950         -1.41057141673716e+0  -5.01018588841894e+0    -4.20790661
 2.200         -2.57545316779455e+0  -3.43222090956673e+0    16.83162644
```

The data is taken from a function of which the derivative could be computed exactly. The actual values are shown here:

X	Value at X	1st Deriv at X	2nd Deriv at X
1.1	0.5488513	−0.0804494	−3.5629715
1.3	0.4520730	−0.9329164	−4.9275779
1.55	0.0499596	−2.3375165	−6.2070293
1.95	−1.4076126	−4.9760746	−6.5786348
2.20	−2.8483454	−6.5025275	−5.4434252

Note the poor results obtained at values outside the range of input data ($x = 2.2$). Also note the large error in the second derivatives near the endpoints of the interval determined by the data.

Differentiation of a User-Defined Function (Derivfn.pas)

Description

Given a user-defined function $f(x)$, this example will approximate the first derivative of the function at a set of x values. The formula

$$f'(x) = [f(x + \Delta X) - f(x - \Delta X)]/2*\Delta X$$

gives a first approximation to the derivative. Richardson extrapolation is then used to refine the approximation (Burden and Faires 1985, 137–152).

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

User-Defined Functions

```
function TNTargetF(X : Extended) : Extended;
```

Input Parameters

`NumDeriv : Integer;`	Number of points at which the derivative is to be approximated
`XDeriv : TNvector;`	X-coordinates of points at which the derivative is to be approximated
`Tolerance : Extended;`	Indicates accuracy of solution

The preceding parameters must satisfy the following conditions:

1. *NumDeriv* $\leq$ *TNArraySize*
2. *Tolerance* $>$ *TNNearlyZero*

TNArraySize represents the number of elements in each vector. It is used in the type definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 1. If condition 1 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`YDeriv : TNvector;`	Approximation to the first derivative at the x-values in *XDeriv*
`Error : Byte;`	0: No errors 1: *Tolerance* < *TNNearlyZero*

Syntax of the Procedure Call

```
FirstDerivative(NumDeriv, XDeriv, YDeriv, Tolerance, Error, @TNTargetF);
```

The procedure *FirstDerivative* approximates the first derivative of function *TNTargetF*.

Comments

Note that the address of *TNTargetF* is passed into the *FirstDerivative* procedure.

Sample Program

The sample program Derivfn.pas provides I/O functions that find the first derivative of a function at a set of points.

Input Files

Derivative points may be entered from a text file. Every derivative point must be followed by a carriage return. For example, to determine the derivatives at x-values 1 through 5, create the following file of derivative points:

```
1
2
3
4
5
```

Example

Problem. Determine the first derivative of $f(x) = \mathrm{sqr}(x) * \cos(x)$ at several points between 1 and 2.2. Actual values of the derivatives to eight significant figures are given here.

First, write the function into the Derivfn.pas program:

```
{ ----- here is the function to differentiate -------------------- }

function TNTargetF(X : Extended) : Extended;

  begin
    TNTargetF := Sqr(X)*Cos(X);
  end;                                    { function TNTargetF }

{ --------------------------------------------------------------- }
```

Run Derivfn.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Number of points (0-100)? 5

Point 1: 1.1
Point 2: 1.3
Point 3: 1.55
Point 4: 1.95
Point 5: 2.2


Tolerance (> 0)? 1E-4
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Tolerance =  1.00000000000000e-4

     X              Derivative at X
   1.100          -8.04494385380667e-2
   1.300          -9.32916380187814e-1
   1.550          -2.33751652942971e+0
   1.950          -4.97607456093026e+0
   2.200          -6.50252751007340e+0
```

The data is taken from a function of which the derivative could be calculated exactly.

Second Differentiation of a User-Defined Function (Deriv2fn.pas)

Description

Given a user-defined function $f(x)$, this example will approximate the second derivative of the function at a set of x values. The three-point formula

$$f''(x) = [f(x + \Delta X) - 2f(x) + f(x - \Delta X)]/\Delta X^2$$

gives a first approximation to the second derivative. Richardson extrapolation is then used to refine the approximation (Burden and Faires 1985, 142–152).

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

User-Defined Function

```
function TNTargetF(X : Extended) : Extended;
```

Input Parameters

`NumDeriv : Integer;`	Number of points at which the derivative is to be approximated
`XDeriv : TNvector;`	*X*-coordinates of points at which the derivative is to be approximated
`Tolerance : Extended;`	Indicates accuracy in solution

The preceding parameters must satisfy the following conditions:

1. *NumDeriv* $\leq$ *TNArraySize*
2. *Tolerance* $\geq$ *TNNearlyZero*

TNArraySize represents the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 1. If condition 1 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`YDeriv : TNvector;`	Approximation to the second derivative at the x-values in *XDeriv*
`Error : Byte;`	0: No errors
	1: *Tolerance* < *TNNearlyZero*

Syntax of the Procedure Call

```
SecondDerivative(NumDeriv, XDeriv, YDeriv, Tolerance, Error, @TNTargetF);
```

SecondDerivative approximates the derivative of function *TNTargetF*.

Comments

Note that the address of *TNTargetF* is passed into the *SecondDerivative* procedure.

Sample Program

The sample program Deriv2fn.pas provides I/O functions that find the second derivative of a function at a set of points.

Input Files

Derivative points may be entered from a text file. Every derivative point must be followed by a carriage return. For example, to determine the second derivatives at x-values 1 through 5, create the following file of derivative points:

```
1
2
3
4
5
```

Example

Problem. Determine the second derivative of $f(x) = \text{sqr}(x)^2 * \cos(x)$ at several points between 1 and 2.2. Actual values of the derivatives to eight significant figures are given here.

First, write the function into the Deriv2fn.pas program:

```
{ ----- here is the function to differentiate ------------------- }

function TNTargetF(X : Extended) : Extended;

  begin
    TNTargetF := Sqr(X)*Cos(X);
  end;                                    { function TNTargetF }

{ ------------------------------------------------------------- }
```

Run Deriv2fn.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Number of points (0-100)? 5

Point 1: 1.1
Point 2: 1.3
Point 3: 1.55
Point 4: 1.95
Point 5: 2.2

Tolerance (> 0)? 1E-4
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Tolerance =  1.00000000000000e-4

     X          Second Derivative at X
   1.100        -3.56297144833630e+0
   1.300        -4.92757792729853e+0
   1.550        -6.20702925616294e+0
   1.950        -6.57863482851564e+0
   2.200        -5.44342518062510e+0
```

The data is taken from a function of which the derivative could be calculated exactly.

CHAPTER 5

Numerical Integration

Integration is another concept used in calculus. It is just the opposite of differentiation, for which routines are provided in Chapter 4. Differentiation tells you the changes in a function, where integration tells you how to add those changes to get the original function.

Integration is most easily understood in terms of areas under curves. Given a function $f(x)$ and real numbers a and b with $a < b$, the area under the curve $y = f(x)$ and above the x-axis between $x = a$ and $x = b$ is given by the integral of $f(x)$ from a to b.

As with derivatives, the laws of calculus are required to compute integrals exactly. The routines in this chapter provide very accurate approximations.

Several methods are described here that approximate the value of a definite integral of a real function of one real variable. Both limits of integration must be finite.

The *trapezoid* method and *Simpson*'s method return an approximation of the integral when a number of equal length subintervals are specified. For a given number of subintervals, Simpson's method is preferred over the trapezoid method whenever the function being integrated is sufficiently smooth.

It is sometimes possible to approximate the definite integral to within a user-specified accuracy with fewer function evaluations using *adaptive schemes.* Adaptive schemes determine the length of each subinterval by the local behavior of the integrand. Simpson's method and the *Gaussian quadrature* method are used with adaptive schemes. The Gaussian quadrature method permits, in some instances, the integrand to possess a singularity at an endpoint of integration, since the function is evaluated at points that are not the endpoints of the interval of integration.

The *Romberg* method uses the trapezoid method and *Richardson extrapolation* to approximate the integral. It returns an approximation within a user-specified accuracy. Except for extremely oscillatory functions or functions that possess an endpoint singularity, this method is fastest and most accurate. If the function oscillates substantially or possesses an endpoint singularity, the adaptive Gaussian quadrature routine is preferred.

Integration Using Simpson's Composite Algorithm (Simpson.pas)

Description

This example uses Simpson's composite algorithm (Burden and Faires 1985, 156–167) to approximate the definite integral of a function $f(x)$ over an interval $[a, b]$. The interval is divided into N subintervals of equal length. The curve in each subinterval is approximated by a second-degree Lagrange polynomial. The integral of the resulting polynomial is then calculated. The sum of the integrals of the N Lagrange polynomials approximates the integral of the function f over the interval $[a, b]$. You must supply the function, the limits of integration, and the number of subintervals.

User-Defined Function

```
function TNTargetF(x : Extended) : Extended;
```

The procedure *Simpson* approximates the integral of this function.

Input Parameters

`LowerLimit : Extended;`	Lower limit of integration
`UpperLimit : Extended;`	Upper limit of integration
`NumIntervals : Integer;`	Number of subintervals over which to apply Simpson's rule

The preceding parameters must satisfy the following condition:

NumIntervals > 0

Output Parameters

`Integral : Extended;`	Approximation to the integral of the function
`Error : Byte;`	0: No errors
	1: *NumIntervals* ≤ 0

Syntax of the Procedure Call

```
Simpson(LowerLimit, UpperLimit, NumIntervals, Integral, Error, @TNTargetF);
```

Simpson approximates the integral of *TNTargetF*.

Sample Program

The sample program Simpson.pas provides I/O functions that demonstrate Simpson's composite algorithm.

Example

Problem. Approximate the integral exp(3*x*) + sqr(*x*)/3 from 0 to 5 using Simpson's composite algorithm.

1. Code function *TNTargetF*:

```
function TNTargetF(x : Extended) : Extended;

{---------------------------------------------------------------------}
{---          THIS IS THE FUNCTION TO INTEGRATE                    ---}
{---------------------------------------------------------------------}

begin
  TNTargetF := Exp(3*X) + Sqr(X)/3;
end;                          { function TNTargetF }
```

2. Run Simpson.pas:

```
Lower limit of integration? 0

Upper limit of integration? 5

Number of intervals (> 0): 100
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
          Lower limit:   0.0000000000000000e+0
          Upper limit:   5.0000000000000000e+0
 Number of intervals: 100

             Integral:   1.08968620446200e+6
```

To eight significant figures, the correct answer is 1,089,686.2.

Integration Using the Trapezoid Composite Rule (Trapzoid.pas)

Description

This example uses the trapezoid composite rule (Burden and Faires 1985, 154–167) to approximate the definite integral of a function $f(x)$ over an interval $[a, b]$. The interval is divided into N subintervals of equal length. In each subinterval the function is approximated by a straight line. The sum of the integrals of the resulting trapezoids approximates the integral of the function f over the interval $[a, b]$. You must supply the function, the limits of integration, and the number of subintervals.

User-Defined Function

```
function TNTargetF(x : Extended) : Extended;
```

The procedure *Trapezoid* approximates the integral of this function.

Input Parameters

`LowerLimit : Extended;`	Lower limit of integration
`UpperLimit : Extended;`	Upper limit of integration
`NumIntervals : Integer;`	Number of subintervals over which to apply the trapezoid rule

The preceding parameters must satisfy the following condition:

NumIntervals > *0*

Output Parameters

`Integral : Extended;`	Approximation to the integral of the function
`Error : Byte;`	0: No errors
	1: *NumIntervals* ≤ *0*

Syntax of the Procedure Call

```
Trapezoid(LowerLimit, UpperLimit, NumIntervals, Integral, Error, @TNTargetF);
```

Trapezoid approximates the integral of *TNTargetF.*

Sample Program

The sample program Trapzoid.pas provides I/O functions that demonstrate the trapezoid composite rule.

Example

Problem. Approximate the integral exp(3*x*) + sqr(*x*)/3 from 0 to 5 using the trapezoid composite rule.

1. Code function *TNTargetF:*

```
function TNTargetF(x : Extended) : Extended;

{------------------------------------------------------------------}
{---        THIS IS THE FUNCTION TO INTEGRATE                   ---}
{------------------------------------------------------------------}

begin
  TNTargetF := Exp(3*X) + Sqr(X)/3;
end;                          { function TNTargetF }
```

2. Run Trapzoid.pas:

```
Lower limit of integration? 0

Upper limit of integration? 5

Number of intervals (> 0)? 100
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
        Lower Limit: 0.00000000000000e+0
        Upper Limit: 5.00000000000000e+0
Number of intervals: 100

           Integral: 1.09172838320801e+6
```

To eight significant figures, the correct answer is 1,091,728.3.

Integration Using Adaptive Quadrature and Simpson's Rule (Adapsimp.pas)

Description

This example contains an algorithm for approximating the definite integral of a function $f(x)$ over an interval $[a,b]$ within a specified tolerance. By increasing the number of subintervals in regions of large functional variation (adaptive quadrature), the desired degree of accuracy can be reached (Burden and Faires 1985, 153–167). The integral within each subinterval is calculated with Simpson's rule. The adaptive quadrature approximates the integral over a subinterval twice: once over the whole subinterval, and again as the sum of the integral over each half of the subinterval. The algorithm halts when the fractional difference between these two approximations is less than the tolerance. You must supply the function, the limits of integration, and the tolerance with which to approximate the integral.

User-Defined Function

```
function TNTargetF(x : Extended) : Extended;
```

The procedure *Adaptive_Simpson* approximates the integral of this function.

Input Parameters

`LowerLimit : Extended;`	Lower limit of integration
`UpperLimit : Extended;`	Upper limit of integration
`Tolerance : Extended;`	Indicates accuracy in solution
`MaxIntervals : Integer;`	Maximum number of subintervals

The preceding parameters must satisfy the following conditions:

1. *Tolerance* > 0
2. *MaxIntervals* > 0

Output Parameters

`Integral : Extended;`	Approximation to the integral of the function
`NumIntervals : Integer;`	Number of subintervals used
`Error : Byte;`	0: No errors 1: *Tolerance* $\leq$ 0 2: *MaxIntervals* $\leq$ 0 3: *NumIntervals* $\geq$ *MaxIntervals*

Syntax of the Procedure Call

```
Adaptive_Simpson(LowerLimit, UpperLimit, Tolerance, MaxIntervals,
                 Integral, NumIntervals, Error, @TNTargetF);
```

Adaptive_Simpson approximates the integral of *TNTargetF*.

Comments

Adaptive quadrature is a recursive routine. In order to avoid recursive procedure calls (which slow down the execution), a stack is created on the heap to simulate recursion.

Sample Program

The sample program Adapsimp.pas provides I/O functions that demonstrate the adaptive quadrature method with Simpson's rule.

Example

Problem. Approximate the integral exp(3*x*) + sqr(*x*)/3 from 0 to 5 using adaptive quadrature and Simpson's rule.

1. Code function *TNTargetF:*

```
function TNTargetF(x : Extended) : Extended;

{-------------------------------------------------------------------}
{---          THIS IS THE FUNCTION TO INTEGRATE                  ---}
{-------------------------------------------------------------------}

begin
  TNTargetF := Exp(3*X) + Sqr(X)/3;
end;                            { function TNTargetF }
```

2. Run Adapsimp.pas:

```
Lower limit of integration? 0

Upper limit of integration? 5

Tolerance (> 0): 1E-8

Maximum number of subintervals (> 0): 1000
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
                   Lower limit:    0.0000000000000000e+0
                   Upper limit:    5.0000000000000000e+0
                     Tolerance:    1.0000000000000000e-8
Maximum number of subintervals: 1000
   Number of subintervals used:  511

                      Integral:    1.08968601332499e+6
```

To eight significant figures, the correct answer is 1,089,686.0.

Integration Using Adaptive Quadrature and Gaussian Quadrature (Adapgaus.pas)

Description

This example contains an algorithm for approximating the integral of a function $f(x)$ over an interval $[a,b]$ within a specified tolerance. By increasing the number of subintervals in regions of large functional variation (adaptive quadrature), the desired degree of accuracy can be reached. The integral within each subinterval is approximated by applying Gaussian quadrature (Burden and Faires 1985, 184-188) with a 16th degree Legendre polynomial. Adaptive quadrature (Burden and Faires 1985, 172-176) approximates the integral over a subinterval twice: once over the whole subinterval, and again as the sum of the integral over each half of the subinterval. The algorithm halts when the fractional difference between these two approximations is less than the tolerance. You must supply the function, the limits of integration, and the tolerance with which to approximate the integral.

User-Defined Function

```
function TNTargetF(x : Extended) : Extended;
```

The procedure *Adaptive_Gauss_Quadrature* approximates the integral of this function.

Input Parameters

`LowerLimit : Extended;`	Lower limit of integration
`UpperLimit : Extended;`	Upper limit of integration
`Tolerance : Extended;`	Indicates accuracy in solution
`MaxIntervals : Integer;`	Maximum number of subintervals

The preceding parameters must satisfy the following conditions:

1. *Tolerance* > 0
2. *MaxIntervals* > 0

Output Parameters

`Integral : Extended;`	Approximation to the integral of the function
`NumIntervals : Integer;`	Number of subintervals used
`Error : Byte;`	0: No errors 1: *Tolerance* ≤ *0* 2: *MaxIntervals* ≤ *0* 3: *NumIntervals* ≥ *MaxIntervals*

Syntax of the Procedure Call

```
Adaptive_Gauss_Quadrature(LowerLimit, UpperLimit, Tolerance, MaxIntervals,
                          Integral, NumIntervals, Error, @TNTargetF);
```

Adaptive_Gauss_Quadrature approximates the integral of *TNTargetF.*

Comments

Adaptive quadrature is a recursive routine. In order to avoid recursive procedure calls (which slow down execution), a stack is created on the heap to simulate recursion.

Gaussian quadrature uses orthogonal polynomials (in this case, Legendre polynomials) to approximate an integral. Generally, a higher degree polynomial will yield a more accurate result, but will take more time to compute. The 16th degree Legendre polynomial used in Adapgaus.pas is very efficient. The values of its zeros and weight factors follow (Abramowitz and Stegun 1972).

The following condition is satisfied by the numbers that follow it:

Integral from -1 to 1 of $f(x)\ dx$

equals

Sum from $i = 1$ to *NumLegendreTerms* of
Legendre[i].*Weight* $*$ f(*Legendre*[i].*Root*)

for an arbitrary function $f(x)$.

Legendre[1]	Root:	0.0950125098376370440185
	Weight:	0.189450610455068496285
Legendre[2]	Root:	0.281603550778258913230
	Weight:	0.182603415044923588867
Legendre[3]	Root:	0.458016777657227386342
	Weight:	0.169156519395002538189
Legendre[4]	Root:	0.617876244402643748447
	Weight:	0.149595988816576732081
Legendre[5]	Root:	0.755404408355003033895
	Weight:	0.124628971255533872052
Legendre[6]	Root:	0.865631202387831743880
	Weight:	0.095158511682492784810
Legendre[7]	Root:	0.944575023073232576078
	Weight:	0.062253523938647892863
Legendre[8]	Root:	0.989400934991649932596
	Weight:	0.027152459411754094852
Legendre[9]	Root:	−0.0950125098376370440185
	Weight:	0.189450610455068496285
Legendre[10]	Root:	−0.281603550778258913230
	Weight:	0.182603415044923588867
Legendre[11]	Root:	−0.458016777657227386342
	Weight:	0.169156519395002538189
Legendre[12]	Root:	−0.617876244402643748447
	Weight:	0.149595988816576732081
Legendre[13]	Root:	−0.755404408355003033895
	Weight:	0.124628971255533872052
Legendre[14]	Root:	−0.865631202387831743880
	Weight:	0.095158511682492784810
Legendre[15]	Root:	−0.944575023073232576078
	Weight:	0.062253523938647892863
Legendre[16]	Root:	−0.989400934991649932596
	Weight:	0.027152459411754094852

Sample Program

The sample program Adapgaus.pas provides I/O functions that demonstrate the adaptive quadrature method with Gaussian quadrature.

Example

Problem. Approximate the integral exp(3*x*) + sqr(*x*)/3 from 0 to 5 using adaptive quadrature with Gaussian quadrature algorithm.

1. Code function *TNTargetF*:

```
function TNTargetF(x : Extended) : Extended;

{------------------------------------------------------------------}
{---          THIS IS THE FUNCTION TO INTEGRATE                 ---}
{------------------------------------------------------------------}
begin
  TNTargetF := Exp(3*X) + Sqr(X)/3;
end;                              { function TNTargetF }
```

2. Run Adapgaus.pas:

```
Lower limit of integration? 0

Upper limit of integration? 5

Tolerance in answer: (> 0): 1E-8

Maximum number of subintervals (> 0): 1000
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
                   Lower limit:    0.000000000000000e+0
                   Upper limit:    5.000000000000000e+0
                     Tolerance:    1.000000000000000e-8
Maximum number of subintervals: 1000
   Number of subintervals used:    1

                      Integral:    1.089686013046090e+6
```

To eight significant figures, the correct answer is 1,089,686.0.

Integration Using the Romberg Algorithm (Romberg.pas)

Description

This example contains an algorithm (Burden and Faires 1985, 177–182) for approximating the integral of a function $f(x)$ over an interval $[a, b]$ within a specified tolerance. The trapezoid rule is used to generate a preliminary approximation, and Richardson extrapolation (Burden and Faires 1985, 148–152) is subsequently used to improve the approximation. Extrapolation continues until the fractional difference between successive approximations of the integral is less than the tolerance. You must supply the function, the limits of integration, and the tolerance with which to approximate the integral.

User-Defined Function

```
function TNTargetF(x : Extended) : Extended;
```

The procedure *Romberg* approximates the integral of this function.

Input Parameters

`LowerLimit : Extended;`	Lower limit of integration
`UpperLimit : Extended;`	Upper limit of integration
`Tolerance : Extended;`	Indicates accuracy in solution
`MaxIter : Integer;`	Maximum number of iterations allowed

The preceding parameters must satisfy the following conditions:

1. *Tolerance* > 0
2. *MaxIter* > 0

Output Parameters

`Integral : Extended;`	Approximation to the integral of the function
`Iter : Integer;`	Number of iterations
`Error : Byte;`	0: No errors 1: *Tolerance* ≤ 0 2: *MaxIter* ≤ 0 3: *Iter* $\geq$ *MaxIter*

Syntax of the Procedure Call

```
Romberg(LowerLimit, UpperLimit, Tolerance, MaxIter, Integral, Iter, Error,
        @TNTargetF);
```

Romberg approximates the integral of *TNTargetF.*

Sample Program

The sample program Romberg.pas provides I/O functions that demonstrate the Romberg algorithm.

Example

Problem. Approximate the integral exp(3*x*) + sqr(*x*)/3 from 0 to 5 using the Romberg algorithm.

1. Code function *TNTargetF:*

```
function TNTargetF(x : Extended) : Extended;

{------------------------------------------------------------------}
{---          THIS IS THE FUNCTION TO INTEGRATE                 ---}
{------------------------------------------------------------------}
begin
  TNTargetF := Exp(3*X) + Sqr(X)/3;
end;                             { function TNTargetF }
```

2. Run Romberg.pas:

```
Lower limit of integration? 0

Upper limit of integration? 5

Tolerance (> 0): 1E-8

Maximum number of iterations: (> 0): 100
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
                Lower limit:   0.00000000000000e+0
                Upper limit:   5.00000000000000e+0
                  Tolerance:   1.00000000000000e-8
Maximum number of iterations: 100
        Number of iterations:   7

                   Integral:   1.08968601696675e+6
```

To eight significant figures, the correct answer is 1,089,686.0.

C H A P T E R **6**

Matrix Routines

This chapter provides routines for dealing with systems of linear equations. An example of a system of linear equations is as follows:

$$\begin{aligned} 2X + Y + Z &= 7 \\ X - Y + Z &= 2 \\ X + Y - Z &= 0 \end{aligned}$$

Matrix algebra is a collection of notations that constitutes a technique for handling such systems. With matrix algebra, the preceding system would be written

$$A\,x = b$$

where

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \end{bmatrix} \qquad x = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \qquad b = \begin{bmatrix} 7 \\ 2 \\ 0 \end{bmatrix}$$

In Pascal, x and b are represented as one-dimensional arrays, and A is represented as a two-dimensional array. In matrix notation, the solution is given by

$$x = A^{-1}\,b$$

where A^{-1} is the *inverse* to A.

The determinant is an indicator of whether the matrix can be inverted. For example, the equations

$$\begin{aligned} 3X - 3Y &= 4 \\ -2X + 2Y &= 5 \end{aligned}$$

cannot be solved. Even for different values of the right-hand side, the equations can only be solved in certain exceptional cases. (If you change 4 and 5 to 3 and -2, then there are infinitely many solutions; but there are none if you change 4 and 5 to 3 and -3.0001.)

Following is a description of several routines that operate on matrices and systems of linear equations.

The determinant of a square matrix is found via Det.pas.

The inverse of a nonsingular matrix is found via Inverse.pas.

The direct techniques implemented to solve a system of N linear equations in N unknowns are *Gaussian elimination*, *Gaussian elimination with partial pivoting*, and *direct factorization*.

The *Gauss-Seidel* method, an iterative technique that converges to the solution, is seldom used for solving small systems, since the time required for sufficient accuracy exceeds that required for the preceding direct techniques.

In general, Gaussian elimination with partial pivoting is the fastest, most accurate algorithm. The following special cases may warrant the use of one of the other routines:

- If you are considering systems where round-off is minimal (that is, small systems whose coefficients are all of nearly the same magnitude), Gaussian elimination without pivoting may be used. It is somewhat faster than its pivoting counterpart.
- When considering sparse coefficient matrices, the Gaussian elimination routine with partial pivoting is the most efficient and accurate routine. If the matrix is small and the nonzero coefficients do not differ wildly from each other, regular Gaussian elimination can usually be used safely.
- For large, dense matrices, the iterative technique is the most efficient; it creates less round-off error than the direct methods. However, the Gauss-Seidel algorithm has its own weaknesses (see the section, "Solving a System of Linear Equations with the Iterative Gauss-Seidel Method," for more details).
- When it is necessary to solve several systems with the same coefficient matrix but a different vector of constant terms, the direct factorization method is the most efficient. This is because it does not require reduction of the coefficient matrix for each vector of constants.

Determinant of a Matrix (Det.pas)

Description

The determinant of an $N \times N$ matrix can be computed by the following algorithm (Gerald and Wheatley 1984, 110–111):

1. Use elementary row operations to make the matrix upper triangular (that is, all the elements below the main diagonal are zero).
2. Find the product of the main diagonal elements — this will be the determinant.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;

TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

`Dimen : Integer;` Dimension of the data matrix

`Data : TNmatrix;` The square matrix

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 0
2. *Dimen* $\leq$ *TNArraySize*

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the type definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Output Parameters

`Det : Extended;`	Determinant of the data matrix
`Error : Byte;`	0: No errors
	1: *Dimen* < 1

Syntax of the Procedure Call

```
Determinant(Dimen, Data, Det, Error);
```

Sample Program

The sample program Det.pas provides I/O functions that demonstrate how to find the determinant of a matrix.

Input File

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be like this:

1. The dimension of the matrix
2. The elements of the matrix in row order; that is,
   [1, 1], [1, 2] ... [1, *N*], [2, 1] ... [2, *N*] ... [*N*, *N*],
   where *N* is the dimension of the matrix

For example, a text file containing the matrix

$$\begin{bmatrix} 2 & 3 \\ -4 & 0 \end{bmatrix}$$

could look like this:

```
 2
 2  3
-4  0
```

Example

Problem. Find the determinant of the following matrix:

$$\begin{bmatrix} 1 & 2 & 0 & -1.0 \\ -1 & 4 & 3 & -0.5 \\ 2 & 2 & 1 & -3.0 \\ 0 & 0 & 3 & -4.0 \end{bmatrix}$$

Run Det.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File.** Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample6A.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The matrix:
 1.00000000  2.00000000  0.00000000 -1.00000000
-1.00000000  4.00000000  3.00000000 -0.50000000
 2.00000000  2.00000000  1.00000000 -3.00000000
 0.00000000  0.00000000  3.00000000 -4.00000000

Determinant =  -2.10000000000000e+1
```

Inverse of a Matrix (Inverse.pas)

Description

The inverse of an $N \times N$ matrix A is an $N \times N$ matrix A^{-1}, such that $A^{-1}A$ equals the identity matrix (Burden and Faires 1985, 306–316). Gauss-Jordan elimination (Gerald and Wheatley 1984, 96–98) is used to transform the original matrix into the identity matrix. The same elementary row operations that reduce A to the identity matrix transform the identity matrix into the inverse of the original matrix A. If one or more of the main diagonal elements of the transformed original matrix (that is, after Gauss-Jordan elimination) is zero, then the original matrix A is singular and its inverse does not exist.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;

TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

`Dimen : Integer;` Dimension of the data matrix

`Data : TNmatrix;` The elements of the square matrix

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 0
2. *Dimen* $\leq$ *TNArraySize*

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the type definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Output Parameters

`INV : TNmatrix;`	The inverse of the data matrix
`Error : Byte;`	0: No errors 1: *Dimen* < 1 2: No inverse exists

Syntax of the Procedure Call

```
Inverse(Dimen, Data, INV, Error);
```

Sample Program

The sample program Inverse.pas provides I/O functions that demonstrate how to find the inverse of a matrix.

Input Files

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. The dimension of the matrix
2. The elements of the matrix in row order; that is,
   [1, 1], [1, 2] ... [1, *N*], [2, 1] ... [2, *N*] ... [*N*, *N*],
   where *N* is the dimension of the matrix

For example, a text file containing the matrix

$$\begin{bmatrix} 2 & 3 \\ -4 & 0 \end{bmatrix}$$

could look like this:

```
 2
 2 3
-4 0
```

Example

Problem. Invert the following matrix:

$$\begin{array}{rrrr} 1 & 2 & 0 & -1.0 \\ -1 & 4 & 3 & -0.5 \\ 2 & 2 & 1 & -3.0 \\ 0 & 0 & 3 & -4.0 \end{array}$$

Run Inverse.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample6A.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The matrix:
 1.000000000  2.000000000  0.000000000 -1.000000000
-1.000000000  4.000000000  3.000000000 -0.500000000
 2.000000000  2.000000000  1.000000000 -3.000000000
 0.000000000  0.000000000  3.000000000 -4.000000000

Inverse:
-1.952380952  0.190476190  1.571428571 -0.714285714
 0.761904762  0.047619048 -0.357142857  0.071428571
-1.904761905  0.380952381  1.142857143 -0.428571429
-1.428571429  0.285714286  0.857142857 -0.571428571
```

To continue this example, reinvert the matrix just obtained:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample6B.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The matrix:
-1.952380952  0.190476190  1.571428571 -0.714285714
 0.761904762  0.047619048 -0.357142857  0.071428571
-1.904761905  0.380952381  1.142857143 -0.428571429
-1.428571429  0.285714286  0.857142857 -0.571428571

Inverse:
 1.000000000  2.000000000  0.000000000 -1.000000000
-1.000000000  4.000000000  3.000000000 -0.500000000
 2.000000000  2.000000000  1.000000000 -3.000000000
-0.000000000 -0.000000000  3.000000000 -4.000000000
```

Solving a System of Linear Equations with Gaussian Elimination (Gauselim.pas)

Description

The solution to a system of N linear equations, $AX = B$, in N unknowns may be found by simultaneously performing Gaussian elimination (Burden and Faires 1985, 291–304) on the matrix containing the coefficients of the equations (the coefficient matrix A) and the vector containing the constant terms of the equations (the constant vector B). First, elementary row operations are used to make A upper triangular (that is, all the elements below the main diagonal are zero). ***Backward substitution*** (whereby $X[N]$ is calculated and used to calculate $X[N-1]$, which is then used to calculate $X[N-2]$, and so on) is then used to compute the solution vector X. If one or more of the elements on the main diagonal of the upper triangular matrix is zero, then no unique solution to the system exists.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;

TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

`Dimen : Integer;`	Dimension of the coefficients matrix
`Coefficients : TNmatrix;`	The square matrix containing the coefficients of the equations
`Constants : TNvector;`	The constant terms of each equation

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 0
2. *Dimen* $\leq$ *TNArraySize*

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Output Parameters

`Solution : TNvector;`	Solution to the set of equations.
`Error : Byte;`	0: No errors.
	1: *Dimen* < 1.
	2: Coefficients matrix is singular; no unique solution exists.

Syntax of the Procedure Call

```
Gaussian_Elimination(Dimen, Coefficients, Constants, Solution, Error);
```

Sample Program

The sample program Gauselim.pas provides I/O functions that demonstrate how to solve a system of linear equations with Gaussian elimination.

Input File

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. The dimension of the coefficient matrix
2. The elements of the matrix in row order; that is,
   [1, 1], [1, 2], ..., [1, *N*], [2, 1], ..., [2, *N*], ..., [*N*, *N*],
   where *N* is the dimension of the matrix
3. The elements of the constant vector, in the order [1],...,[*N*]

For example, to solve the system

$$\begin{aligned} 2x + 3y &= 10 \\ -4x &= 10 \end{aligned}$$

a text file could be created to look like this:

```
  2
  2  3
 -4  0
 10
 10
```

Example

Problem. Solve the following linear system:

$$\begin{aligned} w + 2x + 0y - z &= 10.0 \\ -w + 4x + 3y - 0.5z &= 21.5 \\ 2w + 2x + y - 3z &= 26.0 \\ 3y - 4z &= 37.0 \end{aligned}$$

Run Gauselim.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample6A.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The coefficients:
 1.000000000  2.000000000  0.000000000 -1.000000000
-1.000000000  4.000000000  3.000000000 -0.500000000
 2.000000000  2.000000000  1.000000000 -3.000000000
 0.000000000  0.000000000  3.000000000 -4.000000000

The constants:
 1.00000000000000e+1
 2.15000000000000e+1
 2.60000000000000e+1
 3.70000000000000e+1

The solution:
-1.00000000000000e+0
 2.00000000000000e+0
 3.00000000000000e+0
-7.00000000000000e+0
```

Solving a System of Linear Equations with Gaussian Elimination and Partial Pivoting (Partpivt.pas)

Description

The solution to a system of N linear equations, $AX = B$, in N unknowns may be found by simultaneously performing Gaussian elimination (Burden and Faires 1985, 291–304) on the matrix containing the coefficients of the equations (the coefficient matrix A) and the vector containing the constant terms of the equations (the constant vector B). However, excessive round-off errors can occur when elements on the main diagonal are small compared to the elements below them in the same column. To avoid this, partial pivoting (maximal column pivoting) is performed (Burden and Faires 1985, 324–327); that is, row interchanges are performed so that each main diagonal element is greater than or equal to the elements below it in the same column.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;

TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

`Dimen : Integer;`	Dimension of the coefficients matrix
`Coefficients : TNmatrix;`	The square matrix containing the coefficients of the equations
`Constants : TNvector;`	The constant terms of each equation

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 0
2. *Dimen* $\leq$ *TNArraySize*

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the type definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Output Parameters

`Solution : TNvector;`	Solution to the set of equations.
`Error : Byte;`	0: No errors. 1: *Dimen* < 1. 2: Coefficients matrix is singular; no unique solution exists.

Syntax of the Procedure Call

```
Partial_Pivoting(Dimen, Coefficients, Constants, Solution, Error);
```

Sample Program

The sample program Partpivt.pas provides I/O functions that demonstrate how to solve a system of linear equation with Gaussian elimination and partial pivoting.

Input File

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. The dimension of the matrix
2. The elements of the matrix in row order; that is,
   [1, 1], [1, 2], ..., [1, *N*], [2, 1], ..., [2, *N*], ..., [*N*, *N*],
   where *N* is the dimension of the matrix
3. The elements of the constant vector, in the order [1],...,[*N*]

For example, to solve the system

$$\begin{aligned} 2x + 3y &= 10 \\ -4x &= 10 \end{aligned}$$

a text file could be created to look like this:

```
 2
 2  3
-4  0
10
10
```

Example

Problem. Solve the following linear system:

$$\begin{aligned} w + 2x + 0y - z &= 10 \\ -w + 4x + 3y - 0.5z &= 21.5 \\ 2w + 2x + y - 3z &= 26 \\ 3y - 4z &= 37 \end{aligned}$$

Run Partpivt.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample6A.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The coefficients:
 1.000000000  2.000000000  0.000000000 -1.000000000
-1.000000000  4.000000000  3.000000000 -0.500000000
 2.000000000  2.000000000  1.000000000 -3.000000000
 0.000000000  0.000000000  3.000000000 -4.000000000

The constants:
 1.00000000000000e+1
 2.15000000000000e+1
 2.60000000000000e+1
 3.70000000000000e+1

The solution:
-1.00000000000000e+0
 2.00000000000000e+0
 3.00000000000000e+0
-7.00000000000000e+0
```

Solving a System of Linear Equations with Direct Factoring (Dirfact.pas)

Description

The solution to a system of N linear equations, $AX = B$, in N unknowns can be computed by factoring the matrix containing the coefficients of the N equations (the coefficient matrix A) into an upper triangular matrix U (that is, all the elements below the main diagonal are zero) and a lower triangular matrix L (that is, all the elements above the main diagonal are zero) such that $A = LU$. Partial pivoting is used to reduce round-off error. A record of the pivoting permutations are recorded in a permutation matrix P, so that the equation is actually $A = PLU$. *Forward substitution* (analogous to backward substitution; see "Solving a System of Linear Equations with Gaussian Elimination") is used to solve the equations $LZ = B$ (actually $LZ = PB$, where P is the pivoting permutation matrix) and $UX = Z$ (where X is the solution to the N linear equations, and Z is an intermediate solution). If the coefficient matrix cannot be factored into nonsingular triangular matrices, then no unique solution exists.

This module includes two procedures to perform this algorithm. Procedure *LU_Decompose* performs the LU decomposition of a matrix, and procedure *LU_Solve* performs forward and backward substitution to solve the linear equations.

The most efficient way to calculate the solutions to several systems with the same coefficient matrix but different constant vectors is to first decompose the coefficient matrix A into L and U (Burden and Faires 1985, 342–349). Then perform backward substitution on this decomposed matrix and each of the constant vectors B. Thus, the coefficient matrix is decomposed only once.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;

TNmatrix = array[1..TNArraySize] of TNvector;
```

Procedure LU_Decompose Input Parameters

`Dimen : Integer;`	Dimension of the coefficients matrix
`Coefficients : TNmatrix;`	Square matrix containing the coefficients of the equations

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 0
2. *Dimen* ≤ *TNArraySize*

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Procedure LU_Decompose Output Parameters

`Decomp : TNmatrix;`	The LU decomposition of the coefficients matrix.
`Permute : TNmatrix;`	A permutation matrix that records the effects of pivoting.
`Error : Byte;`	0: No errors. 1: *Dimen* < 1. 2: The coefficients matrix is singular.

Syntax of the Procedure Call

```
LU_Decompose(Dimen, Coefficients, Decomp, Permute, Error);
```

Procedure LU_Solve Input Parameters

`Dimen : Integer;`	Dimension of the coefficients matrix
`Decomp : TNmatrix;`	The LU decomposition of the coefficients matrix
`Constants : TNmatrix;`	The constant terms of each equation
`Permute : TNmatrix;`	A permutation matrix that records the effects of pivoting

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 0
2. *Dimen* $\leq$ *TNArraySize*

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Procedure LU_Solve Output Parameters

`Solution : TNvector;`	Solution to each system of equations
`Error : Byte;`	0: No errors 1: *Dimen* < 1

Syntax of the Procedure Call

```
LU_Solve(Dimen, Decomp, Constants, Permute, Solution, Error);
```

Sample Program

The sample program Dirfact.pas provides I/O functions that demonstrate how to solve a system of linear equations with the method of direct factoring.

Input File

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. The dimension of the matrix
2. The elements of the matrix in row order; that is,
   [1, 1], [1, 2], ..., [1, N], [2, 1], ..., [2, N], ..., [N, N],
   where N is the dimension of the matrix

3. The elements of the first constant vector, in the order [1],...,[N], with each element followed by a carriage return
4. The elements of any additional constant vectors, in the order [1],...,[N], with each element followed by a carriage return

For example, to solve the systems

$$\begin{aligned} 2x + 3y &= 10 \\ -4x &= 10 \end{aligned} \qquad\qquad \begin{aligned} 2x + 3y &= 1 \\ -4x &= 2 \end{aligned}$$

a text file could be created to look like this:

```
 2
 2  3
-4  0
10
10
 1
 2
```

Example

Problem. Given the following set of coefficients:

$$\begin{array}{l} 2w + x + 5y - 8z \\ 7w + 6x + 2y + 2z \\ -1w - 3x - 10y + 4z \\ 2w + 2w + 2y + z \end{array}$$

compute solutions for each of the five constant vectors:

$$\begin{bmatrix} 0 & -15 & 14 & -13 & 5 \\ 17 & 50 & 1 & 84 & 30 \\ -10 & -5 & -12 & -51 & -15 \\ 7 & 17 & 1 & 37 & 10 \end{bmatrix}$$

Run Dirfact.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample6C.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The coefficients:
  2.000000000  1.000000000  5.000000000 -8.000000000
  7.000000000  6.000000000  2.000000000  2.000000000
 -1.000000000 -3.000000000-10.000000000  4.000000000
  2.000000000  2.000000000  2.000000000  1.000000000


The constants:
 0.00000000000000e+0
 1.70000000000000e+1
-1.00000000000000e+1
 7.00000000000000e+0

The solution:
 1.00000000000000e+0
 1.00000000000000e+0
 1.00000000000000e+0
 1.00000000000000e+0

The constants:
-1.50000000000000e+1
 5.00000000000000e+1
-5.00000000000000e+0
 1.70000000000000e+1

The solution:
 2.00000000000000e+0
 5.00000000000000e+0
-2.26268279475236e-19
 3.00000000000000e+0

The constants:
 1.40000000000000e+1
 1.00000000000000e+0
-1.20000000000000e+1
 1.00000000000000e+0

The solution:
 1.00000000000000e+0
-1.00000000000000e+0
 1.00000000000000e+0
-1.00000000000000e+0

The constants:
-1.30000000000000e+1
 8.40000000000000e+1
-5.10000000000000e+1
 3.70000000000000e+1

The solution:
 4.00000000000000e+0
 5.00000000000000e+0
 6.00000000000000e+0
 7.00000000000000e+0
```

```
The constants:
 5.00000000000000e+0
 3.00000000000000e+1
-1.50000000000000e+1
 1.00000000000000e+1

The solution:
 1.98254111540207e-18
 5.00000000000000e+0
 1.07686940416918e-18
 7.38863702730862e-19
```

Solving a System of Linear Equations with the Iterative Gauss-Seidel Method (Gaussidl.pas)

Description

The solution to a system of N linear equations, $AX = B$, in N unknowns can be approximated by the Gauss-Seidel iterative technique (Burden and Faires 1985, 424–432). The equation $AX = B$ is transformed into $X = TX + C$. Given an initial approximation X_o, the sequence $X_m = TX_{m-1} + C$ is generated with the following formula:

$$X_m[i] = \frac{-\sum_{j=1}^{i-1} A[i,j]\, X_m[j] - \sum_{j=i+1}^{N} (A[i,j]\, X_{m-1}[j]) + B[i]}{A[i,i]}$$

The algorithm halts when the fractional difference for each element of the vector X between two iterations is less than a specified tolerance.

If A is *diagonally dominant* (that is, each of the diagonal terms is greater than or equal to the sum of the off-diagonal terms in the same row), then the sequence will converge to the solution X. If the matrix A is not diagonally dominant, then the sequence may converge to the solution, but more likely it will not. You must supply the tolerance with which to approximate a solution.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;

TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

`Dimen : Integer;`	Dimension of the coefficients matrix
`Coefficients : TNmatrix;`	The square matrix containing the coefficients of the equations
`Constants : TNvector;`	The constant terms of the equation
`Tol : Extended;`	Indicates accuracy in solution
`MaxIter : Integer;`	Maximum number of iterations

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 0.
2. *Dimen* ≤ *TNArraySize*.
3. *Tol* > 0.
4. *MaxIter* ≥ 0.
5. The coefficients matrix may not contain a zero on the main diagonal.

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Output Parameters

`Solution : TNvector;`	Solution to the set of equations.
`Iter : Integer;`	The number of iterations required to find the solution.
`Error : Byte;`	0: No errors. 1: *Iter* > *MaxIter* and matrix is not diagonally dominant. 2: *Iter* > *MaxIter* and matrix is diagonally dominant. 3: *Dimen* < 1. 4: *Tol* ≤ 0. 5: *MaxIter* < 0. 6: Zero on the diagonal of the coefficients matrix. 7: Sequence is diverging.

If the coefficients matrix is diagonally dominant, then the Gauss-Seidel method will converge to a solution. If the coefficients matrix is not diagonally dominant, then the Gauss-Seidel may converge to a solution, but more likely it will not. Error 7 can only occur when the coefficients matrix is not diagonally dominant. If Error 1 is returned, it is likely that convergence is not possible; if Error 2 is returned, convergence is possible but will take more than *MaxIter* iterations.

If the diagonal of the coefficients matrix contains a zero (Error 6), then the Gauss-Seidel method may not be used to solve the system of equations.

If the system of equations is under-determined, the Gauss-Seidel method will still converge to a (nonunique) solution. The Gauss-Seidel method cannot distinguish between unique and nonunique solutions. If you suspect that your system of equations is under-determined, use one of the direct methods (for example, Gauselim.pas) to attempt a solution; Gaussian elimination will give an error if it is under-determined. Alternatively, you could use Det.pas to find the determinant; if the determinant is zero, then the system is under-determined.

Syntax of the Procedure Call

```
Gauss_Seidel(Dimen, Coefficients, Constants, Tol, MaxIter, Solution, Iter, Error);
```

Sample Program

The sample program Gaussidl.pas provides I/O functions that demonstrate how to solve a system of linear equations with the iterative Gauss-Seidel method.

Input File

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. The dimension of the matrix
2. The elements of the matrix in row order; that is,
   [1, 1], [1, 2], ..., [1, *N*], [2, 1], ..., [2, *N*], ..., [*N*, *N*],
   where *N* is the dimension of the matrix
3. The elements of the first constant vector, in the order [1],...,[*N*]

For example, to solve the systems

$$\begin{aligned} 20x + 3y &= 10 \\ -4y &= 10 \end{aligned}$$

a text file could be created to look like this:

```
 2
20   3
 0  -4
10
10
```

Example

Problem. Solve the following linear system to within a tolerance of 1E − 12:

$$\begin{aligned} 10v + w + 2x - 3y + 2z &= -29 \\ 4v + 50w + x + z &= 35 \\ -2v + 5w - 30x + y + z &= -25 \\ 6v + 4w + 10y + 3z &= -46 \\ -3v - 2w - x + 6y + 25z &= -106 \end{aligned}$$

Run Gaussidl.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample6D.dat

Tolerance (> 0): 1E-12

Maximum number of iterations (> 0): 100
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The coefficients:
 10.000000000  1.000000000   2.000000000 -3.000000000  2.000000000
  4.000000000 50.000000000   1.000000000  0.000000000  1.000000000
 -2.000000000  5.000000000 -30.000000000  1.000000000  1.000000000
  6.000000000  4.000000000   0.000000000 10.000000000  3.000000000
 -3.000000000 -2.000000000  -1.000000000  6.000000000 25.000000000

The constants:
-2.90000000000000e+1
 3.50000000000000e+1
-2.50000000000000e+1
-4.60000000000000e+1
-1.06000000000000e+2
```

```
                   Tolerance:   1.00000000000000e-12
Maximum number of iterations: 100

   Number of iterations:  15
The result:
-2.99999999999997e+0
 9.99999999999999e-1
 9.99999999999998e-1
-1.99999999999999e+0
-4.00000000000000e+0
```

C H A P T E R 7

Eigenvalues and Eigenvectors

The routines in this chapter can find the eigenvalues and eigenvectors. A scalar c is an *eigenvalue* (or characteristic value) of a square matrix A if there is a nonzero vector v satisfying

$$A v = c v$$

The vector v is called the *eigenvector* corresponding to c.

The eigenvalues and eigenvectors of a matrix provide a lot of information about the matrix. If a matrix is written in terms of a basis of eigenvectors, then it is *diagonal*, meaning that its only nonzero terms are on the main diagonal.

Each procedure in this chapter attempts to approximate at least one real eigenvalue (and associated eigenvector) of a real square matrix. The eigenvector is normalized so that the element with the largest magnitude is 1.

The *power* method approximates the eigenvalue that is largest in magnitude (dominant eigenvalue). The iterative process will converge slowly or not at all if the dominant eigenvalue is not simple or if it has nearly the same magnitude as the next most-dominant eigenvalue.

The *inverse power* method approximates the eigenvalue nearest to a user-supplied real value. This process usually converges more rapidly than the power method, and may be used to refine the approximate value of the eigenvalue determined by the latter method.

The *Wielandt* method attempts to approximate a user-specified number of eigenvalues of a given matrix. The power method is first used to approximate the dominant eigenvalue of the matrix. Deflation is employed to form a deflated, square matrix (that is, a square matrix whose dimension is one less than the original matrix). The eigenvalues of the deflated matrix are identical to those of the original matrix except for the determined dominant eigenvalue. Eigenvectors of the remaining eigenvalues from the original matrix are also contained in the deflated matrix. The dominant eigenvalue of the new deflated matrix is then determined using the power method. Wielandt's method is susceptible to round-off error, thus it may be desirable to use its results as input to the inverse power method.

The *cyclic Jacobi* method approximates all the eigenvalues of a symmetric matrix. The iterative process uses orthogonal plane rotations to reduce the given matrix into a diagonal form. Although Jacobi's method is only applicable to symmetric matrices, it is much more efficient and accurate than Wielandt's method.

Real Dominant Eigenvalue and Eigenvector of a Real Matrix Using the Power Method (Power.pas)

Description

The power method (Burden and Faires 1985, 452–456) approximates the dominant real eigenvalue of a matrix and its associated eigenvector. The dominant eigenvalue is the eigenvalue of the largest absolute magnitude. Given a square matrix A and a real nonzero vector v, a vector w is constructed by the matrix operation $Av = w$. The vector w is normalized by dividing by its element of the largest absolute magnitude q. If the absolute difference between each of the corresponding elements in w and v is less than a specified tolerance, then the procedure halts. Otherwise, v is set equal to w, and the operation repeats until a solution is found. The magnitude q is the dominant eigenvalue, and w will be the associated eigenvector of the matrix A.

You must supply the matrix A, an initial approximation to the eigenvector v, and the tolerance.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;

TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

`Dimen : Integer;`	Dimension of the matrix *Mat*
`Mat : TNmatrix;`	The matrix
`GuessVector : TNvector;`	Initial approximation to the eigenvector
`MaxIter : Integer;`	Maximum number of iterations
`Tolerance : Extended;`	Indicates accuracy in solution

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 1
2. *Dimen* ≤ *TNArraySize*
3. *Tolerance* > 0
4. *MaxIter* > 0

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`Eigenvalue : Extended;`	Approximation to the dominant eigenvalue of the matrix
`Eigenvector : TNvector;`	Approximate eigenvector associated with the dominant eigenvalue
`Iter : Integer;`	Number of iterations required to find the solution
`Error : Byte;`	0: No errors 1: *Dimen* ≤ 1 2: *Tolerance* ≤ 0 3: *MaxIter* ≤ 0 4: *Iter* ≥ *MaxIter*

Syntax of the Procedure Call

```
Power(Dimen, Mat, GuessVector, MaxIter, Tolerance,
     Eigenvalue, Eigenvector, Iter, Error);
```

Comments

The power method will not converge if the initial approximation (*Guess*) to the eigenvector is orthogonal to the dominant eigenvector. If the initial approximation is *orthogonal*, then the power method will converge to a different eigenvector without warning. If you suspect this has happened, run the routine with several different initial approximations.

The power method may not converge to repeated eigenvalues with linearly dependent eigenvectors. Repeated eigenvalues with linearly independent eigenvectors do not pose a problem.

The eigenvectors are normalized such that the element of largest absolute magnitude in each vector is equal to one.

Sample Program

The sample program Power.pas provides I/O functions that demonstrate the power method of approximating eigenvalues.

Input File

Data may be input from a text file. Entries in the text file should be separated by spaces or carriage returns, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. Dimension of the matrix
2. Elements of the matrix, in the order
   [1, 1], [1, 2], ..., [1, *N*], ..., [*N*, 1], ..., [*N*, *N*],
   where *N* is the dimension of the matrix

For example, to find the dominant eigenvalue of the matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

you could first create the following text file:

```
4
1
2
3
4
```

Example

Problem. Find the dominant eigenvalue of the matrix:

$$\begin{bmatrix} 2 & 10 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 4 \end{bmatrix}$$

using the initial guess (1, 2, 3).

Run Power.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Dimension of the matrix (1-30)? 3

Matrix[1, 1]:  2
Matrix[1, 2]: 10
Matrix[1, 3]:  0
Matrix[2, 1]:  0
Matrix[2, 2]:  1
Matrix[2, 3]:  0
Matrix[3, 1]:  0
Matrix[3, 2]:  2
Matrix[3, 3]:  4

Now input an initial guess for the eigenvector:
Vector[1]: 1
Vector[2]: 2
Vector[3]: 3

Tolerance (> 0): 1E-8

Maximum number of iterations (> 0): 100
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The matrix:
 2.00000000000000e+0    1.00000000000000e+1    0.00000000000000e+0
 0.00000000000000e+0    1.00000000000000e+0    0.00000000000000e+0
 0.00000000000000e+0    2.00000000000000e+0    4.00000000000000e+0

                   Tolerance:  1.00000000000000e-8
Maximum number of iterations: 100


        Number of iterations:  12
The approximate eigenvector:
-2.30295124326775e-14
 3.53562219190609e-30
 1.00000000000000e+0

   The associated eigenvalue:  4.00000000000000e+0
```

The exact solution is

Eigenvalue = 4
Eigenvector = (0, 0, 1)

Real Eigenvalue and Eigenvector of a Real Matrix Using the Inverse Power Method (InvPower.pas)

Description

Where the power method converges to the dominant real eigenvalue of a matrix (see Power.pas), the inverse power method (Burden and Faires 1985, 459–462) converges to the real eigenvalue nearest to a user-supplied real value. Given a square matrix A, an initial approximation p to the eigenvalue, and an initial approximation v to the eigenvector, the linear system $(A - pI)w = v$ (where I is the identity matrix) is solved via LU decomposition (see Chapter 6, "Solving a System of Linear Equations with Direct Factoring"). The vector w is normalized by dividing through by the element q with the largest absolute magnitude. If the absolute difference between each of the corresponding elements in v and w is less than a specified tolerance, then the procedure halts. Otherwise, v is set equal to w, and the previous matrix equation is solved again. The process repeats until a solution is reached. The eigenvalue of A closest to p will be $(1/q + p)$, and w will be the associated eigenvector.

You must supply the matrix A, the initial approximations p and v, and the tolerance.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;

TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

`Dimen : Integer;`	Dimension of the matrix *Mat*
`Mat : TNmatrix;`	The matrix
`GuessVector : TNvector;`	Initial approximation (*Guess*) of the eigenvector
`ClosestVal : Extended;`	The approximate eigenvalue
`MaxIter : Integer;`	Maximum number of iterations
`Tolerance : Extended;`	Indicates accuracy of solution

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 1
2. *Dimen* ≤ *TNArraySize*
3. *Tolerance* > 0
4. *MaxIter* > 0

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`Eigenvalue : Extended;`	Approximation to the eigenvalue closest to *ClosestVal*
`Eigenvector : TNvector;`	Approximation to the eigenvector associated with *Eigenvalue*
`Iter : Integer;`	Number of iterations required to find the solution
`Error : Byte;`	0: No errors 1: *Dimen* ≤ 1 2: *Tolerance* ≤ 0 3: *MaxIter* ≤ 0 4: *Iter* ≥ *MaxIter* 5: *Eigenvalue/Eigenvector* not calculated (see "Comments")

Syntax of the Procedure Call

```
InversePower(Dimen, Mat, GuessVector, ClosestVal, MaxIter,
             Tolerance, Eigenvalue, Eigenvector, Iter, Error);
```

Comments

The inverse power method approximates the solution of a system of linear equations. If the matrix (*Mat* - *Eigenvalue* * *I*) is singular, where *I* is the ***identity matrix***, the method will not converge to a solution and Error 5 will be returned. If this occurs, run the routine again with a slightly different initial approximation, *ClosestVal*.

The power method may not converge to repeated eigenvalues with linearly dependent eigenvectors. Repeated eigenvalues with linearly independent eigenvectors do not pose a problem.

The inverse power method is sensitive to the initial approximation (*ClosestVal*). If *ClosestVal* is not close to an eigenvalue or lies midway between two eigenvalues, the algorithm will converge very slowly, if at all.

The eigenvectors are normalized such that the element of the largest absolute magnitude in each vector is equal to one.

Sample Program

The sample program InvPower.pas provides I/O functions that demonstrate the inverse power method of approximating eigenvalues.

Input File

Data may be input from a text file. Entries in the text file should be separated by spaces or carriage returns, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. Dimension of the matrix
2. Elements of the matrix, in the order
   [1, 1], [1, 2], ..., [1, *N*], ..., [*N*, 1], ..., [*N*, *N*],
   where *N* is the dimension of the matrix
3. Elements of the initial guess, in the order
   [1], [2], ..., [*N*],
   where *N* is the dimension of the matrix

For example, to find an eigenvalue of the matrix

```
1 2
3 4
```

with an initial guess of (11, 10), you could first create the following text file:

```
 4
 1
 2
 3
 4
11
10
```

Example

Problem. Suppose you know that two of the eigenvalues of the matrix

```
2 10 0
0  1 0
0  2 4
```

are approximately 1.999 and 0.7. Use the inverse power method with an initial guess of (1, 2, 3) to refine these approximations.

Run InvPower.pas with 1.999 as the approximate eigenvalue:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Dimension of the matrix (1-30)? 3

Matrix[1, 1]:  2
Matrix[1, 2]: 10
Matrix[1, 3]:  0
Matrix[2, 1]:  0
Matrix[2, 2]:  1
Matrix[2, 3]:  0
Matrix[3, 1]:  0
Matrix[3, 2]:  2
Matrix[3, 3]:  4

Now input an initial guess for the eigenvector:
Vector[1]: 1
Vector[2]: 2
Vector[3]: 3

Approximate eigenvalue : 1.999

Tolerance (> 0): 1E-8

Maximum number of iterations (> 0): 200
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The matrix:
 2.00000000000000e+0    1.00000000000000e+1    0.00000000000000e+0
 0.00000000000000e+0    1.00000000000000e+0    0.00000000000000e+0
 0.00000000000000e+0    2.00000000000000e+0    4.00000000000000e+0

      Approximate eigenvalue:  1.99900000000000e+0
                   Tolerance:  1.00000000000000e-8
Maximum number of iterations: 200


        Number of iterations:   4
 The approximate eigenvector:
 1.00000000000000e+0
 9.56200019081920e-14
-5.08756039829010e-14

    The associated eigenvalue:  2.00000000000096e+0
```

Run InvPower.pas with 0.7 as the approximate eigenvalue:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Dimension of the matrix (1-30)? 3

Matrix[1, 1]:  2
Matrix[1, 2]: 10
Matrix[1, 3]:  0
Matrix[2, 1]:  0
Matrix[2, 2]:  1
Matrix[2, 3]:  0
Matrix[3, 1]:  0
Matrix[3, 2]:  2
Matrix[3, 3]:  4

Now input an initial guess for the eigenvector:
Vector[1]: 1
Vector[2]: 2
Vector[3]: 3

Approximate eigenvalue : 0.7

Tolerance (> 0): 1E-8

Maximum number of iterations (> 0): 200
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The matrix:
 2.00000000000000e+0    1.00000000000000e+1    0.00000000000000e+0
 0.00000000000000e+0    1.00000000000000e+0    0.00000000000000e+0
 0.00000000000000e+0    2.00000000000000e+0    4.00000000000000e+0

      Approximate eigenvalue:  7.00000000000000e-1
                   Tolerance:  1.00000000000000e-8
Maximum number of iterations: 200


        Number of iterations:  12
 The approximate eigenvector:
 1.00000000000000e+0
-1.00000002395103e-1
 6.66666682633328e-2

    The associated eigenvalue:  9.99999976048973e-1
```

The exact solutions are

Eigenvalue = 2; *Eigenvector* = (1, 0, 0)
Eigenvalue = 1; *Eigenvector* = (1, -0.1, 2/30)

Real Eigenvalues and Eigenvectors of a Real Matrix Using the Power Method and Wielandt's Deflation (Wielandt.pas)

Description

Wielandt's deflation is a technique that approximates each real eigenvalue and related eigenvector of a matrix (Burden and Faires 1985, 452–456). Once the dominant real eigenvalue/vector of a matrix has been approximated with the power method (see "Real Dominant Eigenvalue and Eigenvector of a Real Matrix Using the Power Method"), the next most dominant real eigenvalue/vector is approximated by removing the dominant solution. This deflates the matrix. The deflated matrix will have the same eigenvalues as the original matrix (except for the removed ones). The eigenvectors of the deflated matrix will be related to the eigenvectors of the original matrix. (They will not be identical because the dimension of the deflated matrix is less than the dimension of the original matrix.) The power method then approximates the dominant eigenvalue of the deflated matrix. This process is repeated until the appropriate number (user-supplied) of eigenvalues/vectors have been approximated.

You must supply the matrix, the number of eigenvalues/vectors to approximate, and the tolerance with which to approximate the eigenvalues/vectors.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;

TNmatrix = array[1..TNArraySize] of TNvector;

TNIntVector = array[1..TNArraySize] of Integer;
```

Input Parameters

`Dimen : Integer;`	Dimension of the matrix *Mat*
`Mat : TNmatrix;`	The matrix
`GuessVector : TNvector;`	Initial approximation (*Guess*) of an eigenvector

`MaxEigens : Integer;`	Number of eigenvalues/vectors to find (at most, *Dimen*), (see "Comments")
`MaxIter : Integer;`	Maximum number of iterations
`Tolerance : Extended;`	Indicates accuracy in solution

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 1
2. *Dimen* ≤ *TNArraySize*
3. *Tolerance* > 0
4. *MaxIter* > 0
5. *MaxEigens* > 0
6. *MaxEigens* ≤ *Dimen*

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`NumEigens : Integer;`	The number of eigenvectors returned (will be ≤ *MaxEigens*).
`Eigenvalues : TNvector;`	The first *NumEigens* eigenvalues of the matrix.
`Eigenvectors : TNmatrix;`	The eigenvectors associated with the eigenvalues.
`Iter : TNIntVector;`	Number of iterations required to find each eigenvalue/vector.
`Error : Byte;`	0: No errors. 1: *Dimen* ≤ 1. 2: *Tolerance* ≤ 0. 3: *MaxIter* ≤ 0. 4: *MaxEigens* ≤ 0, *MaxEigens* > *Dimen*. 5: *Iter* ≥ *MaxIter*. 6: Warning! Not a fatal error! The last two eigenvalues aren't real.

Syntax of the Procedure Call

```
Wielandt(Dimen, Mat, GuessVector, MaxEigens, MaxIter, Tolerance,
        NumEigens, Eigenvalues, Eigenvectors, Iter, Error);
```

Comments

It is often unnecessary to determine the complete eigensystem of a matrix. The parameter *MaxEigens* prevents the routine from approximating more eigenvalues/ vectors than needed. For example, if the four most dominant eigenvalues of a 20 × 20 matrix are desired, set *MaxEigens* equal to 4. The algorithm will halt when it has approximated the four most dominant eigenvalues, thus saving a considerable amount of time. Note, however, that the dimension of the vector eigenvalues and the matrix eigenvectors must still be *TNArraySize* (that is, the same as the dimension of the matrix).

The power method may not converge to repeated eigenvalues with linearly dependent eigenvectors. Repeated eigenvalues with linearly independent eigenvectors do not pose a problem.

The eigenvectors are normalized such that the element of the largest absolute magnitude in each vector is equal to one.

It is difficult to determine why the power method doesn't converge to a particular eigenvector; usually the eigenvalue is complex, or eigenvectors of repeated eigenvalues are linearly dependent. However, when Wielandt's deflation has deflated the matrix to a 2 × 2, it is easy to determine if the eigenvalues of the 2 × 2 are real or complex. If the last two eigenvalues are real, then they (and their associated eigenvectors) are returned; if the last two eigenvalues are complex, Error 6 is returned. (Error 6 is only a warning; it is not a fatal error.) It is returned to give you some information about the undetermined eigenvectors.

Sample Program

The sample program Wielandt.pas provides I/O functions that demonstrate Wielandt's method of approximating eigensystems.

Input File

Data may be input from a text file. Entries in the text file should be separated by spaces or carriage returns, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. Dimension of the matrix
2. Elements of the matrix, in the order
   [1, 1], [1, 2], ..., [1, *N*], ..., [*N*, 1], ..., [*N*, *N*],
   where *N* is the dimension of the matrix

For example, to find the dominant eigenvalue of the matrix

```
1 2
3 4
```

you could first create the following text file:

```
4
1
2
3
4
```

Example

Problem. Find all real eigenvalues and eigenvectors of the matrix

$$\begin{bmatrix} 2 & 10 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 4 \end{bmatrix}$$

using an initial guess of (1, 2, 3).

Run Wielandt.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **Keyboard** and click **OK**. Then input the data as follows:

```
Dimension of the matrix (1-30)? 3

Matrix[1, 1]:  2
Matrix[1, 2]: 10
Matrix[1, 3]:  0
Matrix[2, 1]:  0
Matrix[2, 2]:  1
Matrix[2, 3]:  0
Matrix[3, 1]:  0
Matrix[3, 2]:  2
Matrix[3, 3]:  4
```

```
Now input an initial guess for the eigenvector:
Vector[1]: 1
Vector[2]: 2
Vector[3]: 3

Tolerance (> 0): 1E-6

Maximum number of eigenvalues/eigenvectors to find (<= 3): 3

Maximum number of iterations (> 0): 200
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The matrix:
 2.00000000000000e+0    1.00000000000000e+1    0.00000000000000e+0
 0.00000000000000e+0    1.00000000000000e+0    0.00000000000000e+0
 0.00000000000000e+0    2.00000000000000e+0    4.00000000000000e+0

                   Tolerance:  1.00000000000000e-6
Maximum number of eigenvalues/eigenvectors to find: 3
Maximum number of iterations: 200



        Number of iterations:  10
The approximate eigenvector:
-8.32731765655097e-7
 4.60590248231668e-15
 1.00000000000000e+0

   The associated eigenvalue:  4.00000000000004e+0


        Number of iterations:   0
The approximate eigenvector:
 1.00000000000000e+0
-0.00000000000000e+0
-0.00000000000000e+0

   The associated eigenvalue:  2.00000000000000e+0


        Number of iterations:   0
The approximate eigenvector:
 1.00000000000000e+0
-9.99999888969116e-2
 6.66666592646069e-2

   The associated eigenvalue:  9.99999999999991e-1
```

The exact solution is

$$\begin{aligned}
&\textit{Eigenvalue} = 4;\ \textit{Eigenvector} = (0, 0, 1)\\
&\textit{Eigenvalue} = 2;\ \textit{Eigenvector} = (1, 0, 0)\\
&\textit{Eigenvalue} = 1;\ \textit{Eigenvector} = (1, -0.1, 2/30)
\end{aligned}$$

The Complete Eigensystem of a Symmetric Real Matrix Using the Cyclic Jacobi Method (Jacobi.pas)

Description

The *eigensystem* of a symmetric matrix can be computed much more simply and efficiently than the eigensystem of an asymmetric matrix. The cyclic Jacobi method (Atkinson and Harley 1983, 154–160) is an iterative technique for approximating the complete eigensystem of a symmetric matrix to within a given tolerance. It consists of multiplying the matrix A by a series of *rotation matrices* R_i. The rotation matrices are chosen so that the elements of the upper triangular part of A (excluding the diagonal) are systematically annihilated; that is, R_1 is chosen so that $A[1, 2]$ becomes zero, R_2 is chosen so that $A[1, 3]$ becomes zero, and so on. Since the matrix is symmetric, this will also annihilate the lower triangular part of A. Because each rotation will probably change the value of elements annihilated in previous rotations, the method is iterative. Eventually, the matrix will be diagonalized. The eigenvalues will be the elements of the main diagonal of the diagonal matrix; the eigenvectors will be the corresponding rows of the matrix created by the product of the rotation matrices R_i.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;

TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

`Dimen : Integer;`	Dimension of the matrix *Mat*
`Mat : TNmatrix;`	The symmetric matrix
`MaxIter : Integer;`	Maximum number of iterations
`Tolerance : Extended;`	Accuracy in solution

The preceding parameters must satisfy the following conditions:

1. *Dimen* $>$ 1.
2. *Dimen* $\leq$ *TNArraySize*.
3. *Tolerance* $>$ 0.
4. *MaxIter* $>$ 0.
5. *Mat* must be symmetric.

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`Eigenvalues : TNvector;`	Approximation to the eigenvalues of the matrix
`Eigenvectors : TNmatrix;`	Approximation to the eigenvectors associated with the eigenvalues
`Iter : Integer;`	Number of iterations required to find eigenvalues/vectors
`Error : Byte;`	0: No errors 1: *Dimen* $\leq$ 1 2: *Tolerance* $\leq$ 0 3: *MaxIter* $\leq$ 0 4: *Mat* not symmetric 5: *Iter* $\geq$ *MaxIter*

Syntax of the Procedure Call

```
Jacobi(Dimen, Mat, MaxIter, Tolerance, Eigenvalues, Eigenvectors, Iter, Error);
```

Comments

For symmetric matrices, the Jacobi method is preferred to Wielandt's deflation.

Unlike the power and inverse power methods, the efficiency of the Jacobi method is not affected by repeated eigenvalues with linearly dependent eigenvectors.

The eigenvectors are normalized such that the element of largest absolute magnitude in each vector is equal to one.

Sample Program

The sample program Jacobi.pas provides I/O functions that demonstrate Jacobi's method of approximating the eigensystem of symmetric matrices.

Input File

Data may be input from a text file. Entries in the text file should be separated by spaces or carriage returns, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. Dimension of the matrix
2. Elements of the matrix, in the order
   [1, 1], [1, 2], ..., [1, *N*], ..., [*N*, 1], ..., [*N*, *N*],
   where *N* is the dimension of the matrix

For example, to find the dominant eigenvalue of the matrix

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$$

you could first create the following text file:

```
4
1
2
2
1
```

Example

Problem. Find the complete eigensystem of the symmetric matrix

$$\begin{bmatrix} 1 & 2 & -3 & -1 \\ 2 & 1 & -1 & -3 \\ -3 & -1 & 1 & 2 \\ -1 & -3 & 2 & 1 \end{bmatrix}$$

Run Jacobi.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample7A.dat

Tolerance (> 0): 1E-8

Maximum number of iterations (> 0): 200
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The matrix:
  1.000000000  2.000000000 -3.000000000 -1.000000000
  2.000000000  1.000000000 -1.000000000 -3.000000000
 -3.000000000 -1.000000000  1.000000000  2.000000000
 -1.000000000 -3.000000000  2.000000000  1.000000000

                    Tolerance:  1.00000000000000e-8
 Maximum number of iterations: 200

         Number of iterations:   4

  The approximate eigenvector:
-1.00000000000000e+0
-1.00000000000000e+0
 1.00000000000000e+0
 1.00000000000000e+0

    The associated eigenvalue:  7.00000000000000e+0

  The approximate eigenvector:
 9.99999999977805e-1
-9.99999999977804e-1
-1.00000000000000e+0
 1.00000000000000e+0

    The associated eigenvalue:  1.00000000000000e+0
  The approximate eigenvector:
 1.00000000000000e+0
-9.99999556935429e-1
 9.99999999977805e-1
-9.99999556913233e-1

    The associated eigenvalue: -2.99999999999990e+0

  The approximate eigenvector:
 9.99999556935428e-1
 1.00000000000000e+0
 9.99999556935429e-1
 1.00000000000000e+0

    The associated eigenvalue: -1.00000000000010e+0
```

The exact solution is

$$\begin{aligned}
\mathit{Eigenvalue} &= 7; \ \mathit{Eigenvector} = (1, 1, -1, -1)\\
\mathit{Eigenvalue} &= -3; \ \mathit{Eigenvector} = (1, -1, 1, -1)\\
\mathit{Eigenvalue} &= 1; \ \mathit{Eigenvector} = (-1, 1, 1, -1)\\
\mathit{Eigenvalue} &= -1; \ \mathit{Eigenvector} = (1, 1, 1, 1)
\end{aligned}$$

CHAPTER 8

Initial Value and Boundary Value Methods

A *differential equation* is like an ordinary equation except that the unknown is a function, and derivatives of the function appear in the equation. For example,

$$f''(x) + f(x) = 0$$

is a differential equation. $f''(x)$ is the second derivative of $f(x)$. The solutions are the functions of the form

$$f(x) = a * \cos(x) + b * \sin(x)$$

The function is uniquely determined by suitable initial conditions, such as

$$f(0) = 3$$
$$f'(0) = 4$$

in which case the solution is

$$f(x) = 3 * \cos(x) + 4 * \sin(x)$$

The routines in this chapter solve differential equations that are *ordinary* and *linear.* A differential equation is ordinary if there is only an independent variable (that is, the unknown function is a function of only one variable), and thus the derivatives are ordinary derivatives and not partial derivatives. A differential equation is linear if the unknown function and its derivatives appear linearly in the equation.

This chapter describes routines that specifically solve: (1) initial value problems for nth-order ordinary differential equations, (2) initial value problems for systems of coupled first-order and second-order ordinary differential equations, and (3) boundary value problems for second-order ordinary differential equations.

Note that these routines work only with ordinary differential equations, not partial differential equations. All of the routines in this chapter can solve problems involving nonlinear equations.

Two one-step techniques that solve initial value problems for first-order ordinary differential equations are implemented. The first technique employs the *fourth-order Runge-Kutta* method, also known as the *classical Runge-Kutta* method. The second employs the *Runge-Kutta-Fehlberg* method.

Each one-step technique approximates the value of the dependent variable at a *mesh point*, which is a value of the independent variable, by using only the information obtained from the preceding mesh point. The Runge-Kutta method employs equally spaced mesh points. On the other hand, the Runge-Kutta-Fehlberg method varies the spacing of the mesh points in order to control the local truncation error. This produces a corresponding bound on the global error.

The *Adams-Bashforth/Adams-Moulton predictor/corrector* method is a multistep method that uses information obtained at several preceding mesh points to approximate the value of the dependent variable at the current mesh point. The procedure employs the Adams-Bashforth four-step method to obtain a predictor. It is subsequently used as input for the Adams-Moulton three-step method to obtain a corrector. The corrector is the approximate value of the solution. Mesh points are equally spaced, and the starting values for the process are determined by the one step, fourth-order Runge-Kutta method.

The Runge-Kutta methods are the most reliable and should be used when you are uncertain of the behavior of the differential equation (for example, if the solution to the differential equation is not very smooth). If you want the output to be evenly spaced (in x) or do not require a high degree of accuracy, use the classical Runge-Kutta method. Otherwise, the Runge-Kutta-Fehlberg method is the best general purpose routine to use, since it provides control over the accuracy of the solution.

The Adams-Bashforth/Adams-Moulton method achieves the same accuracy (for equally spaced mesh points) as the fourth-order Runge-Kutta formula, but it is significantly faster. Consequently, the Adams-Bashforth/Adams-Moulton method is the most desirable method if you are reasonably certain that the differential equation is well-behaved.

Initial value problems for first-order ordinary differential equations are guaranteed to have a unique solution on the interval a, b if the function

$$x' = f(t, x)$$

is continuous over the interval a, b, and if the function satisfies the ***Lipshitz condition.*** The Lipshitz condition states that there exists a positive number L such that

$$|f(t, x_2) - f(t, x_1)| \leq L|x_2 - x_1|$$

for all $a \leq t \leq b, -\infty < x < \infty$.

Initial value problems for second-order ordinary differential equations can be solved via a fourth-order Runge-Kutta method (Runge_2.pas). This procedure reduces a given differential equation to a system of two, first-order ordinary differential equations. The solution to this system is approximated at equally spaced mesh points with the fourth-order Runge-Kutta method.

Initial value problems for second-order ordinary differential equations are guaranteed to have a unique solution on the interval a, b if the function

$$x'' = f(t, x, x')$$

is continuous over the interval a, b and if the function satisfies the Lipshitz condition. For a second-order differential equation, the Lipshitz condition states that there exists a positive number L such that

$$|f(t, x_2, x'_2) - f(t, x_1, x'_2)| \leq L(|x_2 - x_1| + |x'_2 - x'_1|)$$

for all $a \leq t \leq b, -\infty < x < \infty, -\infty < x' < \infty$.

The Runge-Kutta method can be generalized for any order ordinary differential equation. The file Runge_N.pas contains an algorithm that can solve an initial value problem for an nth-order differential equation with the fourth-order Runge-Kutta formulas. The Lipshitz condition can be generalized for any order ordinary differential equation. (For details, consult the reference book listed in the section, "Solution to an Initial Value Problem for a First-Order Ordinary Differential Equation Using the Runge-Kutta Method.")

Although Runge_N.pas can be used to solve initial value problems for first-order and second-order ordinary differential equations, we recommend that Runge_1.pas and Runge_2.pas be used instead. The notation used by these routines is somewhat simpler than the general case. There is no significant difference in computation time between the general program (Runge_N.pas) and the specific programs (Runge_1.pas and Runge_2.pas).

Systems of coupled equations may also be solved with Runge-Kutta techniques. A system of up to ten first-order ordinary differential equations can be solved with the file Runge_S1.pas. A system of up to ten second-order ordinary differential equations can be solved with the file Runge_S2.pas. The algorithms in both these files are based on the classical Runge-Kutta method with uniform spacing between mesh points; hence, they do not allow for accuracy control (as in the Runge-Kutta-Fehlberg method). (The Lipshitz condition for systems of equations is given in the reference in the sections about Runge_S1.pas and Runge_S2.pas.)

Boundary value problems for second-order ordinary differential equations (where the value of the dependent variable is specified at the two endpoints of interval) can be solved using *shooting techniques.* Shooting techniques converge onto the slope of the function at one boundary. This reduces the boundary value problem to a series of initial value problems. The series concludes when the initial value problem satisfies the boundary condition at the other boundary.

If the second-order differential equation is linear (that is, linear in the *dependent* variable(s), not necessarily linear in the *independent* variable), the linear-shooting method (linshot2.pas) may be used. A linear combination of solutions to two initial value problems yields the solution to the boundary value problem.

If the second-order differential equation is nonlinear, the routine Shoot2.pas must be used. The secant method generates a sequence of solutions with different values of the first derivative until the appropriate boundary condition, subject to a desired accuracy, is satisfied. Although Shoot2.pas may be used to solve linear boundary value problems, Linshot2.pas is more efficient for the linear case.

Boundary value problems for second-order differential equations are guaranteed to have a unique solution on the interval a, b if the function

$$y'' = f(x, y, y')$$

and the two partial derivatives $\partial f/\partial y$, $\partial f/\partial y'$ are continuous on the interval $[a, b]$. Furthermore, $\partial f/\partial y$ must be positive and $\partial f/\partial y'$ must be bounded for all x, y, y' $a \leq x \leq b$, $-\infty < y < \infty$, $-\infty < y' < \infty$.

The convergence to the appropriate initial value of the first derivative is not assured for nonlinear boundary value problems. A good guess of the derivative boundary condition is often required and may involve considerable trial and error.

Interpolation techniques (see Chapter 3) may be used to approximate the solution of values of the independent variable that are not mesh points.

Solution to an Initial Value Problem for a First-Order Ordinary Differential Equation Using the Runge-Kutta Method (Runge_1.pas)

Description

This example uses the Runge-Kutta method (Burden and Faires 1985, 220–227) to approximate the solution to a first-order ordinary differential equation with a specified initial condition.

Given a function of the form

$dx/dt = TNTargetF(t, x)$

which satisfies the conditions given at the beginning of this chapter, and an initial condition

$x[LowerLimit] = XInitial$

and spacing

$h = (UpperLimit - LowerLimit)/NumIntervals$

the fourth-order Runge-Kutta method approximates x in the interval [*LowerLimit*, *UpperLimit*].

The fourth-order Runge-Kutta formulas consist of the following:

$$\begin{aligned} F1 &= h * TNTargetF(t, x[t]) \\ F2 &= h * TNTargetF(t + h/2, x[t] + F1/2) \\ F3 &= h * TNTargetF(t + h/2, x[t] + F2/2) \\ F4 &= h * TNTargetF(t + h, x[t] + F3) \\ x[t + 1] &= x[t] + (F1 + 2 * F2 + 2 * F3 + F4)/6 \end{aligned}$$

where t ranges from *LowerLimit* to *UpperLimit* in steps of h. These formulas give a truncation error of order h^4.

You must supply *LowerLimit*, *UpperLimit*, *XInitial*, *NumIntervals*, and *TNTargetF*.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

User-Defined Function

```
TNTargetF(t, X : Extended) : Extended;
```

$dx/dt = TNTargetF(t, x)$

The function *TNTargetF*(t, x) is a user-defined function that calculates the derivative dx/dt.

Input Parameters

`LowerLimit : Extended;`	Lower limit of interval
`UpperLimit : Extended;`	Upper limit of interval
`XInitial : Extended;`	Value of X at *LowerLimit*
`NumReturn : Integer;`	Number of (t, x) pairs returned from the procedure
`NumIntervals : Integer;`	Number of subintervals used in calculations

The preceding parameters must satisfy the following conditions:

1. *NumReturn* > 0
2. *NumIntervals* $\geq$ *NumReturn*
3. *LowerLimit* $\neq$ *UpperLimit*

Output Parameters

`TValues : TNvector;`	Values of t between the limits
`XValues : TNvector;`	Values of X approximated at the values in *TValues*
`Error : Byte;`	0: No errors 1: *NumReturn* < 1 2: *NumIntervals* $<$ *NumReturn* 3: *LowerLimit* $=$ *UpperLimit*

Syntax of the Procedure Call

```
InitialCond1stOrder(LowerLimit, UpperLimit, XInitial, NumReturn,
                    NumIntervals, TValues, XValues, Error, @TNTargetF);
```

The procedure *InitialCondition1stOrder* integrates the first-order differential equation.

Comments

This procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use all the values. The vectors *TValues* and *XValues* will contain only *NumReturn* values at roughly equally spaced t-values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

Warning: A *stiff differential equation* occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program Runge_1.pas provides I/O functions that demonstrate the Runge-Kutta method of solving initial value problems. Note that the address of *TNTargetF* is passed into the *InitialCondition1stOrder* procedure.

Example

Problem. Solve the following initial value problem with the Runge-Kutta method:

$$x' = x/t + t - 1 \qquad 1 \le t \le 2$$
$$x(1) = 1$$

1. Code the equation into the program Runge_1.pas:

```
function TNTargetF(t, X : Extended) : Extended;

{-----------------------------------------------------------------}
{---        THIS IS THE FIRST-ORDER DIFFERENTIAL EQUATION      ---}
{-----------------------------------------------------------------}

begin
  TNTargetF := x/t + t - 1
end;                                   { function TNTargetF }
```

2. Run Runge_1.pas:

```
Lower limit of interval? 1

Upper limit of interval? 2

X value at t = 1.00000e+0: 1

Number of values to return (1-40)? 10

Number of intervals (>= 10)? 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
         Lower limit:  1.00000000000000e+0
         Upper limit:  2.00000000000000e+0
Value of X at 1.0000:  1.00000000000000e+0
 Number of intervals: 100

    t              X
1.00000000  1.00000000000000e+0
1.10000000  1.10515880220649e+0
1.20000000  1.22121413182916e+0
1.30000000  1.34892645616477e+0
1.40000000  1.48893886869362e+0
1.50000000  1.64180233779216e+0
1.60000000  1.80799419315265e+0
1.70000000  1.98793197313186e+0
1.80000000  2.18198400310574e+0
1.90000000  2.39047761619428e+0
2.00000000  2.61370563879444e+0
```

The exact solution is

$$\begin{aligned} X &= t^2 - t * \ln(t) \\ X(2) &= 2.6137056 \end{aligned}$$

Solution to an Initial Value Problem for a First-Order Ordinary Differential Equation Using the Runge-Kutta-Fehlberg Method (RKF_1.pas)

Description

This example uses the Runge-Kutta-Fehlberg method (Burden and Faires 1985, 230–235) to approximate a solution within a specified tolerance to a first-order ordinary differential equation with a specified initial condition.

Where the Runge-Kutta method (see Runge_1.pas) uses a constant spacing h, the Runge-Kutta-Fehlberg method varies the spacing so that the solution can be approximated with accuracy.

Given a function of the form

$$dx/dt = TNTargetF(t, x)$$

which satisfies the conditions given at the beginning of this chapter, and an initial condition

$$x[LowerLimit] = XInitial$$

both the fourth-order and fifth-order Runge-Kutta formulas are used to approximate x in the interval [*LowerLimit*, *UpperLimit*]. The number of subintervals is continually increased until the fractional difference between the results of the fourth-order and fifth-order formulas (which give a truncation error of h^4 and h^5, respectively) in each subinterval is less than the specified tolerance.

You must supply *LowerLimit*, *UpperLimit*, *Tolerance*, and *TNTargetF*.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

User-Defined Function

```
TNTargetF(t, X : Extended) : Extended;
```

$$dx/dt = TNTargetF(t, x)$$

Input Parameters

`LowerLimit : Extended;`	Lower limit of interval
`UpperLimit : Extended;`	Upper limit of interval
`XInitial : Extended;`	Value of X at *LowerLimit*
`Tolerance : Extended;`	Maximum tolerable fractional difference between iterate values
`NumReturn : Integer;`	Number of (t, x) values to be returned

The preceding parameters must satisfy the following conditions:

1. *Tolerance* > 0
2. *NumReturn* > 0
3. *LowerLimit* $\neq$ *UpperLimit*

Output Parameters

`TValues : TNvector;`	Values of t at which X was approximated
`XValues : TNvector;`	Values of X at the values in *TValues*
`Error : Byte;`	0: No errors 1: *Tolerance* ≤ 0 2: *NumReturn* ≤ 0 3: *LowerLimit* = *UpperLimit* 4: *Tolerance* not reached

Syntax of the Procedure Call

```
RungeKuttaFehlberg(LowerLimit, UpperLimit, XInitial, Tolerance,
                   NumReturn, TValues, XValues, Error, @TNTargetF);
```

The procedure *RungeKuttaFehlberg* integrates the first-order differential equation *TNTargetF*.

Comments

This procedure will compute more values in its calculations than it will return in the vectors *TValues* and *XValues*. The vectors *TValues* and *XValues* will contain only *NumReturn* values at subintervals between the lower and upper limits. More values will be returned in regions of large functional variation than in regions of small functional variation. Thus, you can ensure a highly accurate solution (by making the *Tolerance* small) without generating an excessive amount of output (by making *NumReturn* small).

The Runge-Kutta-Fehlberg method improves the accuracy in the solution by reducing the spacing between successive values of t. However, if the *Tolerance* is too small, the spacing required to reach *Tolerance* may be beyond the machine's limit of precision. Consequently, the routine will not converge to a solution that meets the required *Tolerance* and Error 5 will be returned.

Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta-Fehlberg method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program RKF_1.pas provides I/O functions that demonstrate the Runge-Kutta-Fehlberg method of solving initial value problems. Note that the address of *TNTargetF* is passed into the *Runge-Kutta-Fehlberg* procedure.

Example

Problem. Use the Runge-Kutta-Fehlberg method to solve the following initial value problem with a tolerance of 1E-6:

$$x' = x/t + t - 1 \qquad 1 \le t \le 2$$
$$x(1) = 1$$

1. Code the differential equation into the program RKF_1.pas:

   ```
   function TNTargetF(t, X : Extended) : Extended;

   {-------------------------------------------------------------------}
   {---          THIS IS THE FIRST-ORDER DIFFERENTIAL EQUATION        ---}
   {-------------------------------------------------------------------}

   begin
     TNTargetF := x/t + t - 1;
   end;                                      { function TNTargetF }
   ```

2. Run RKF_1.pas:

   ```
   Lower limit of interval? 1

   Upper limit of interval? 2

   X value at t = 1.00000e+0: 1

   Number of values to return (1-40)? 10

   Tolerance (> 0)? 1E-6
   ```

   Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

   ```
            Lower limit:  1.00000000000000e+0
            Upper limit:  2.00000000000000e+0
   Value of X at 1.0000:  1.00000000000000e+0
              Tolerance:  1.00000000000000e-6

        t              X
   1.00000000  1.00000000000000e+0
   1.10000000  1.10515881708653e+0
   1.20000000  1.22121416069278e+0
   1.30000000  1.34892649817459e+0
   1.40000000  1.48893892310351e+0
   1.50000000  1.64180240395245e+0
   1.60000000  1.80799427050390e+0
   1.70000000  1.98793206119471e+0
   1.80000000  2.18198410146987e+0
   1.90000000  2.39047772450816e+0
   2.00000000  2.61370575675625e+0
   ```

Now solve the same problem with a smaller tolerance, 1.000E-08:

```
Lower limit of interval? 1

Upper limit of interval? 2

X value at t = 1.00000e+0: 1

Number of values to return (1-40)? 10

Tolerance (> 0)? 1E-8
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
           Lower Limit: 1.00000000000000e+0
           Upper Limit: 2.00000000000000e+0
Value of X at 1.0000 : 1.00000000000000e+0
             Tolerance: 1.00000000000000e-8

     T              X
  1.00000000 1.00000000000000e+0
  1.12208941 1.12982837401487e+0
  1.20585321 1.22836146842843e+0
  1.29271260 1.33921121932749e+0
  1.38286653 1.46405185232472e+0
  1.47648998 1.60468229893107e+0
  1.57374241 1.76304147999705e+0
  1.67477301 1.94122165035498e+0
  1.77972398 2.14148082489667e+0
  1.88873280 2.36625482901586e+0
  2.00193373 2.61816928271558e+0
```

The exact solution is

$$X = t^2 - t\ln(t)$$
$$X(2) = 2.6137056$$
$$X(2.00193373) = 2.6181693$$

In the first run, a solution could be approximated within tolerance with a spacing of 0.1. In the second run, the algorithm had to vary the spacing in order to approximate a solution within the tolerance.

Solution to an Initial Value Problem for a First-Order Ordinary Differential Equation Using the Adams-Bashforth/Adams-Moulton Predictor/Corrector Scheme (Adams_1.pas)

Description

This example approximates the solution to a first-order ordinary differential equation with a specified initial condition using the four-step Adams-Bashforth/Adams-Moulton formulas (Burden and Faires 1985, 238–247). Runge-Kutta methods are one-step methods, because each calculation uses information from only one previous point. The Adams' formulas use information from four previous points, thus the four-step method.

Given a function of the form

$dx/dt = TNTargetF(t, x)$

which satisfies the conditions given at the beginning of this chapter, and an initial condition

$x[LowerLimit] = XInitial$

and spacing

$h = (UpperLimit - LowerLimit)/NumIntervals$

the fourth-order Runge-Kutta formula (see Runge_1.pas) is used to find approximations at the first three points in the interval [*LowerLimit*, *UpperLimit*]. Then the following explicit Adams-Bashforth formula:

$$\begin{aligned} x_o[i+1] = x[i] + h/24 * \{ & 55 * TNTargetF(t[i], x[i]) \\ & - 59 * TNTargetF(t[i-1], x[i-1]) \\ & + 37 * TNTargetF(t[i-2], x[i-2]) \\ & - 9 * TNTargetF(t[i-3], x[i-3]) \} \end{aligned}$$

and the following implicit Adams-Moulton formula:

$$\begin{aligned} x[i+1] = x[i] + h/24 * \{ & 9 * TNTargetF(t[i+1], x_o[i+1]) \\ & + 19 * TNTargetF(t[i], x[i]) \\ & - 5 * TNTargetF(t[i-1], x[i-1]) \\ & + TNTargetF(t[i-2], x[i-2]) \} \end{aligned}$$

approximate (predict) and refine (correct) all other points in the interval.

You must supply *UpperLimit*, *LowerLimit*, *XInitial*, *NumIntervals*, and *TNTargetF*.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

User-Defined Function

```
TNTargetF(t, X : Extended) : Extended;
```

$dx/dt = TNTargetF(t, x)$

Input Parameters

`LowerLimit : Extended;`	Lower limit of interval
`UpperLimit : Extended;`	Upper limit of interval
`XInitial : Extended;`	Value of X at *LowerLimit*
`NumReturn : Integer;`	Number of (t, x) values to be returned from the procedure
`NumIntervals : Integer;`	Number of subintervals to be used in calculations

The preceding parameters must satisfy the following conditions:

1. $NumReturn > 0$
2. $NumIntervals \geq NumReturn$
3. $LowerLimit \neq UpperLimit$

Output Parameters

`TValues : TNvector;`	Values of t between the limits
`XValues : TNvector;`	Values of X determined at the values in *TValues*
`Error : Byte;`	0: No errors 1: $NumReturn < 1$ 2: $NumIntervals < NumReturn$ 3: $LowerLimit = UpperLimit$

Syntax of the Procedure Call

```
Adams(LowerLimit, UpperLimit, XInitial, NumReturn,
      NumIntervals,TValues, XValues, Error, @TNTargetF);
```

The procedure *Adams* integrates the first-order differential equation *TNTargetF*.

Comments

This procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use the values. The vectors *TValues* and *XValues* will contain only *NumReturn* values at roughly equally spaced t-values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Adams-Bashforth/Adams-Moulton method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program Adams_1.pas provides I/O functions that demonstrate the Adams-Bashforth/Adams-Moulton predictor/corrector method of solving initial value problems. Note that the address of *TNTargetF* gets passed into the *Adams* procedure.

Example

Problem. Solve the following initial value problem with the Adams-Bashforth/Adams-Moulton method:

$$x' = x/t + t - 1 \qquad 1 \le t \le 2$$
$$x(1) = 1$$

1. Code the differential equation into the program Adams_1.pas:

```
function TNTargetF(t, X : Extended) : Extended;

{-------------------------------------------------------------------}
{---        THIS IS THE FIRST-ORDER DIFFERENTIAL EQUATION        ---}
{-------------------------------------------------------------------}

begin
  TNTargetF := x/t + t - 1;
end;                                      { function TNTargetF }
```

2. Run Adams_1.pas:

```
Lower limit of interval? 1

Upper limit of interval? 2

X value at t = 1.00000e+0: 1

Number of values to return (1-40)? 10

Number of intervals (>= 10)? 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
         Lower limit:   1.00000000000000e+0
         Upper limit:   2.00000000000000e+0
Value of X at 1.0000:   1.00000000000000e+0
 Number of intervals: 100

     t              X
1.00000000  1.00000000000000e+0
1.10000000  1.10515880229293e+0
1.20000000  1.22121413201736e+0
1.30000000  1.34892645643801e+0
1.40000000  1.48893886904034e+0
1.50000000  1.64180233820416e+0
1.60000000  1.80799419362396e+0
1.70000000  1.98793197365806e+0
1.80000000  2.18198400368348e+0
1.90000000  2.39047761682098e+0
2.00000000  2.61370563946811e+0
```

The exact solution is

$$X = t^2 - t\ln(t)$$
$$x(2) = 2.6137056$$

Solution to an Initial Value Problem for a Second-Order Ordinary Differential Equation Using the Runge-Kutta Method (Runge_2.pas)

Description

This example approximates the solution to a second-order ordinary differential equation with specified initial conditions using the two variable Runge-Kutta formulas (Burden and Faires 1985, 261–269).

Given a function of the form

$d^2x/dt^2 = TNTargetF(t, x, x')$

where x' indicates dx/dt (which satisfies the Lipshitz condition given at the beginning of this chapter), the initial conditions

$x[LowerLimit] = InitialValue$
$x'[LowerLimit] = InitialDeriv$

and spacing

$h = (UpperLimit - LowerLimit)/NumIntervals$

rewrite the second-order differential equation as two, first-order differential equations:

$x' = y$
$y' = TNTargetF(t, x, y)$

Then the fourth-order, two-variable Runge-Kutta method can be used to approximate simultaneously x and y (x and x').

The fourth-order Runge-Kutta formulas for these equations consist of the following:

$F1x = h * y[t]$

$F1y = h * TNTargetF(t, x[t], y[t])$

$F2x = h * (y[t] + F1y/2)$

$F2y = h * TNTargetF(t + h/2, x[t] + F1x/2, y[t] + F1y/2)$

$F3x = h * (y[t] + F2y/2)$

$F3y = h * TNTargetF(t + h/2, x[t] + F2x/2, y[t] + F2y/2)$

$F4x = h * (y[t] + F3y)$

$F4y = h * TNTargetF(t + h, x[t] + F3x, y[t] + F3y)$

$$x[t+1] = x[t] + (F1x + 2 * F2x + 2 * F3x + F4x)/6$$
$$y[t+1] = y[t] + (F1y + 2 * F2y + 2 * F3y + F4y)/6$$

where t ranges from *LowerLimit* to *UpperLimit* in steps of h. These formulas give a truncation error of order h^4.

You must supply *LowerLimit*, *UpperLimit*, *XInitial*, *NumIntervals*, and *TNTargetF*.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

User-Defined Function

```
TNTargetF(t, X, XPrime : Extended) : Extended;
```

$$dx^2/dt^2 = TNTargetF(t, x, dx/dt)$$

Input Parameters

`LowerLimit : Extended;`	Lower limit of interval
`UpperLimit : Extended;`	Upper limit of interval
`InitialValue : Extended;`	Value of X at *LowerLimit*
`InitialDeriv : Extended;`	Derivative of X at *LowerLimit*
`NumReturn : Integer;`	Number of (t, x) values returned from the procedure
`NumIntervals : Integer;`	Number of subintervals used in the calculations

The preceding parameters must satisfy the following conditions:

1. *NumReturn* > 0
2. *NumIntervals* $\geq$ *NumReturn*
3. *LowerLimit* $\neq$ *UpperLimit*

Output Parameters

`TValues : TNvector;`	Values of *t* between the limits
`XValues : TNvector;`	Values of *X* determined at the values in *TValues*
`XDerivValues : TNvector;`	Values of the first derivative of *X* determined at the values in *TValues*
`Error : Byte;`	0: No errors 1: *NumReturn* < 1 2: *NumIntervals* < *NumReturn* 3: *LowerLimit* = *UpperLimit*

Syntax of the Procedure Call

```
InitialCond2ndOrder(LowerLimit, UpperLimit, InitialValue, InitialDeriv,
                    NumReturn, NumIntervals, TValues, XValues,
                    XDerivValues, Error, @TNTargetF);
```

The procedure *InitialCondition2ndOrder* integrates the second-order differential equation *TNTargetF.*

Comments

This procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use all these values. The vectors *TValues*, *XValues*, and *XDerivValues* will contain only *NumReturn* values at roughly equally spaced *t*-values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn.*) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

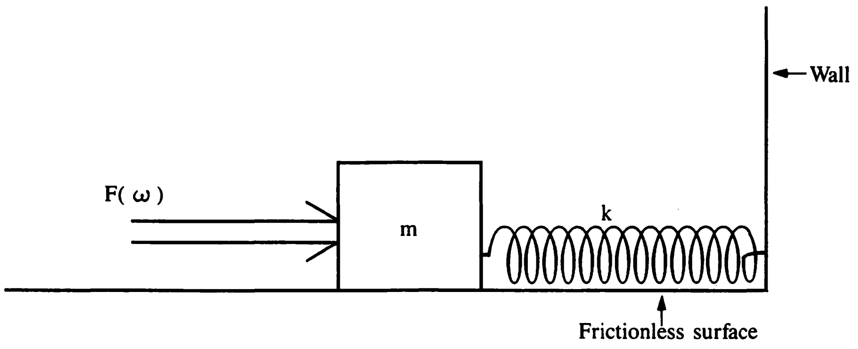
Warning: A differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program Runge_2.pas provides I/O functions that demonstrate the Runge-Kutta method of solving initial value problems for second-order ordinary differential equations. Note that the address of *TNTargetF* gets passed into the *InitialCondition2ndOrder* procedure.

Example

Problem. A weight with mass m lies on a frictionless table and is connected to a spring with spring constant k:



If the weight is subject to a driving force $F \sin(\omega t)$ (ω represents the frequency of the driving force and t is time), the equation of motion of the mass is as follows:

$$m\, d^2x/dt^2 + k\, x = F \sin(\omega\, t)$$

Given

$m = 2$ kg
$F = 9$ N
$k = 32$ N/m
$\omega = 5$ cycles/sec
$x(0) = 0$ m
$dx(0)/dt = -2.5$ m/sec

find the position and velocity of the block from $t = 0$ second to $t = 2$ seconds.

1. Rewrite the preceding second-order differential equation:

$$d^2x/dt^2 = F/m \sin(\omega\, t) - k/m\, x$$

2. Code this second-order differential equation into the program Runge_2.pas:

```
function TNTargetF(t : Extended;
                   X : Extended;
                   XPrime : Extended) : Extended;

{------------------------------------------------------------------}
{---         THIS IS THE SECOND-ORDER DIFFERENTIAL EQUATION      ---}
{------------------------------------------------------------------}

begin
  TNTargetF := 9/2 * Sin (5 * t) - 32/2 * x;
end;                                  { function TNTargetF }
```

3. Run Runge_2.pas:

```
Lower limit of interval? 0

Upper limit of interval? 2

Enter X value at t = 0.00000e+0: 0
Enter derivative of X at t = 0.00000e+0: -2.5

Number of values to return (1-40)? 10

Number of intervals (>= 10)? 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
                Lower Limit: 0.00000000000000e+0
                Upper Limit: 2.00000000000000e+0
        Value of X at 0.0000 : 0.00000000000000e+0
       Value of X' at 0.0000 :-2.50000000000000e+0
          Number of intervals: 100

     T           Value of X           Derivative of X
0.00000000   0.00000000000000e+0   -2.50000000000000e+0
0.20000000  -4.20735284275848e-1   -1.35075642830665e+0
0.40000000  -4.54648724216734e-1    1.04036531118478e+0
0.60000000  -7.05605786993375e-2    2.47497991717220e+0
0.80000000   3.78400378699554e-1    1.63411037473655e+0
1.00000000   4.79461767300631e-1   -7.09151289407566e-1
1.20000000   1.39708469016312e-1   -2.40042152228323e+0
1.40000000  -3.28491796183335e-1   -1.88475529635975e+0
1.60000000  -4.94677974769031e-1    3.63745224811835e-1
1.80000000  -2.06059519715177e-1    2.27781864414105e+0
2.00000000   2.72008842396951e-1    2.09767516082022e+0
```

The exact solution is

$$x = \frac{F \sin(\omega\, t)}{m\,(\omega_o^2 - \omega^2)}$$

$$dx/dt = \frac{F_\omega \cos(\omega\, t)}{m\,(\omega_o^2 - \omega^2)}$$

where ω_o is the natural frequency of the system

$$\omega_o^2 = k/m$$

The period of oscillation is given by

$$t = 2\,\pi/\omega = 1.257 \text{ sec}$$

The data is taken from a function of which the derivative could be computed exactly. Following are the actual values:

t	Values of X	Derivative of X
0.0	0.000000000000E+000	−2.500000000000E+000
0.2	−4.207354924039E−001	−1.350755764670E+000
0.4	−4.546487134128E−001	1.040367091367E+000
0.6	−7.056000402993E−002	2.474981241501E+000
0.8	3.784012476539E−001	1.634109052159E+000
1.0	4.794621373315E−001	−7.091554636580E−001
1.2	1.397077490994E−001	−2.400425716625E+000
1.4	−3.284932993593E−001	−1.884755635858E+000
1.6	−4.946791233116E−001	3.637500845215E−001
1.8	−2.060592426208E−001	2.277825654711E+000
2.0	2.720105554446E−001	2.097678822691E+000

Solution to an Initial Value Problem for an nth-Order Ordinary Differential Equation Using the Runge-Kutta Method (Runge_N.pas)

Description

This example integrates an *n*th-order ordinary differential equation with specified initial conditions using the generalized Runge-Kutta formulas (Burden and Faires 1985, 261–269).

Given a function of the form

$$d^n x/dt^n = TNTargetF(t, x, x^{(1)}, \ldots, x^{(n-1)})$$

where $x^{(j)}$ indicates $d^j x/dt^j$, which satisfies the general Lipshitz condition (the Lipshitz condition for first-order and second-order ordinary differential equations is given at the beginning of this chapter, and initial condition

$$\begin{aligned} x[LowerLimit] &= a_1 \\ x^{(1)}[LowerLimit] &= a_2 \\ &\vdots \\ x^{(n-1)}[LowerLimit] &= a_n \end{aligned}$$

and spacing

$$h = (UpperLimit - LowerLimit)/NumIntervals$$

rewrite the *n*th-order differential equation as *n* first-order differential equations:

$$\begin{aligned} x^{(1)} &= y_1 \\ x^{(2)} &= y^{(1)}{}_1 = y_2 \\ x^{(3)} &= y^{(1)}{}_2 = y_3 \\ &\vdots \\ x^{(n-1)} &= y^{(1)}{}_{n-2} = y_{n-1} \\ x^{(n)} &= y^{(1)}{}_{n-1} = TNTargetF(t, x, y_1, y_2, \ldots, y_{n-1}) \end{aligned}$$

Then the fourth-order general Runge-Kutta method can be used to approximate simultaneously the *y*'s (*x* and its derivatives).

The general Runge-Kutta formulas for these equations consist of the following:

$$
\begin{aligned}
F1x &= h * y_1[t] \\
F1y_1 &= h * y_2[t] \\
&\cdot \\
&\cdot \\
F1y_{n-2} &= h * y_{n-1}[t] \\
F1y_{n-1} &= h * TNTargetF(t, x[t], y_1[t], \ldots, y_{n-1}[t]) \\
F2x &= h * (y_1[t] + F1y_1/2) \\
F2y_1 &= h * (y_2[t] + F1y_2/2) \\
&\cdot \\
&\cdot \\
F2y_{n-2} &= h * (y_{n-1}[t] + F1y_{n-1}/2) \\
F2y_{n-1} &= h * TNTargetF(t + h/2, x[t] + F1x/2, y_1[t] + F1y_1/2, \ldots, y_{n-1}[t] \\
&\quad + F1y_{n-1}/2) \\
F3x &= h * (y_1[t] + F2y_1/2) \\
F3y_1 &= h * (y_2[t] + F2y_2/2) \\
&\cdot \\
&\cdot \\
F3y_{n-2} &= h * (y_{n-1}[t] + F2y_{n-1}/2) \\
F3y_{n-1} &= h * TNTargetF(t + h/2, x[t] + F2x/2, y_1[t] + F2y_1/2, \ldots, y_{n-1}[t] \\
&\quad + F2y_{n-1}/2) \\
F4x &= h * (y_1[t] + F3y_1) \\
F4y_1 &= h * (y_2[t] + F3y_2) \\
&\cdot \\
&\cdot \\
F4y_{n-2} &= h * (y_{n-1}[t] + F3y_{n-1}) \\
F4y_{n-1} &= h * TNTargetF(t + h, x[t] + F3x, y_1[t] + F3y_1, \ldots, y_{n-1}[t] \\
&\quad + F3y_{n-1}) \\
x[t+1] &= x[t] + (F1x + 2 * F2x + 2 * F3x + F4x)/6 \\
y_1[t+1] &= y_1[t] + (F1y_1 + 2 * F2y_1 + 2 * F3y_1 + F4y_1)/6 \\
y_2[t+1] &= y_2[t] + (F1y_2 + 2 * F2y_2 + 2 * F3y_2 + F4y_2)/6 \\
&\cdot \\
&\cdot \\
y_{n-2}[t+1] &= y_{n-2}[t] + (F1y_{n-2} + 2 * F2y_{n-2} + 2 * F3y_{n-2} + F4y_{n-2})/6 \\
y_{n-1}[t+1] &= y_{n-1}[t] + (F1y_{n-1} + 2 * F2y_{n-1} + 2 * F3y_{n-1} + F4y_{n-1})/6
\end{aligned}
$$

where t ranges from *LowerLimit* to *UpperLimit* in steps of h. These formulas give a truncation error of order h^4.

You must supply the order, limits, initial values, and *TNTargetF*. The order may be arbitrarily large.

User-Defined Types

```
TNvector = array[0..TNRowSize] of Extended;
TNmatrix = array[0..TNColumnSize] of TNvector;
```

TNRowSize is an upper bound for the number of values returned for a particular variable (*NumReturn*). *TNColumnSize* is an upper bound for the order of the differential equation (*Order*).

User-Defined Function

```
TNTargetF(V : TNvector) : Extended;
```

The elements of *V* are defined as

V[0] corresponds to *t*
V[1] corresponds to *x*
V[2] corresponds to first derivative of *x*
V[3] corresponds to second derivative of *x*
.
.
.

This is the differential equation:

$d^nx/dt^n = TNTargetF(t, x, x^{(1)}, \dots x^{(n-1)})$ where *n* is the order of the equation.

The procedure *InitialCondition* integrates this *n*th-order differential equation.

Input Parameters

`Order : Integer;`	Order of the differential equation
`LowerLimit : Extended;`	Lower limit of interval
`UpperLimit : Extended;`	Upper limit of interval
`InitialValues : TNvector;`	Values of *X* and its derivatives at *LowerLimit*
`NumReturn : Integer;`	Number of $(t, x, x^{(1)}, \dots, x^{(n)})$ values returned from the procedure
`NumIntervals : Integer;`	Number of subintervals used in the calculations

The preceding parameters must satisfy the following conditions:

1. *NumReturn* > 0
2. *NumIntervals* $\geq$ *NumReturn*
3. *Order* > 0
4. *LowerLimit* $\neq$ *UpperLimit*

Output Parameters

`SolutionValues : TNmatrix;`	Values of t, x and the derivatives of x between the limits
`Error : Byte;`	0: No errors 1: *NumReturn* < 1 2: *NumIntervals* $<$ *NumReturn* 3: *Order* < 1 4: *LowerLimit* $=$ *UpperLimit*

Syntax of the Procedure Call

```
InitialCondition(Order, LowerLimit, UpperLimit, InitialValues,
                 NumReturn, NumIntervals, SolutionValues, Error, @TNTargetF);
```

Comments

The first row of *SolutionValues* will be the values of t between the limits, the second row of *SolutionValues* will be the values of x between the limits, the third row of *SolutionValues* will be the values of $x^{(1)}$ between the limits, and so on.

This procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use all those values. The rows of *SolutionValues* will contain only *NumReturn* values at roughly equally spaced t-values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program Runge_N.pas provides I/O functions that demonstrate the Runge-Kutta method of solving initial value problems for high-order ordinary differential equations. Note that the address of *TNTargetF* gets passed into the *InitialCondition* procedure.

Example

Problem. Find the solution to the following fourth-order ordinary differential equation from $t = 0$ to $t = 1$:

$$d^4x(t)/dt^4 = -4\,x(t)\;d^3x(t)/dt^3$$

$$\begin{aligned} x(0) &= 1 \\ dx(0)/dt &= -1 \\ d^2x(0)/dt^2 &= 2 \\ d^3x(0)/dt^3 &= -6 \end{aligned}$$

1. Code the equation into the program Runge_N.pas:

```
function TNTargetF(V : TNvector) : Extended;

{------------------------------------------}
{    THIS IS THE DIFFERENTIAL EQUATION      }
{------------------------------------------}
{                                           }
{  d^n x                    (1)       (n-1) }
{  -----  = TNTargetF(t, x, x    , ... x    ) }
{                                           }
{   dt^n                                    }
{                                           }
{ where n is the order of the equation.     }
{                                           }
{ The elements of V are defined:            }
{ V[0] corresponds to t                     }
{ V[1] corresponds to X                     }
{ V[2] corresponds to 1st derivative of X   }
{ V[3] corresponds to 2nd derivative of X   }
{                  .                        }
{                  .                        }
{                  .                        }
{------------------------------------------}

begin
  TNTargetF := -4 * V[1] * V[4];
end;                      { function TNTargetF }
```

2. Run Runge_N.pas:

```
Order of the equation (1-40)? 4

Lower limit of interval? 0

Upper limit of interval? 1

    Enter X value at t = 0.00000e+0:  1
Derivative 1 of X at t = 0.00000e+0: -1
Derivative 2 of X at t = 0.00000e+0:  2
Derivative 3 of X at t = 0.00000e+0: -6

Number of values to return (1-40)? 10

Number of intervals (>= 10)? 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
                       Lower Limit: 0.00000000000000e+0
                       Upper Limit: 1.00000000000000e+0
               Number of intervals: 100

 Initial conditions at lower limit:
                  X[1]= 1.00000000000000e+0
                  X[2]=-1.00000000000000e+0
                  X[3]= 2.00000000000000e+0
                  X[4]=-6.00000000000000e+0
```

```
    t              Value X[1]
0.00000000   1.00000000000000e+0
0.10000000   9.09090909737517e-1
0.20000000   8.33333334189336e-1
0.30000000   7.69230770157394e-1
0.40000000   7.14285715280102e-1
0.50000000   6.66666667788519e-1
0.60000000   6.25000001337168e-1
0.70000000   5.88235295769619e-1
0.80000000   5.55555557625526e-1
0.90000000   5.26315792064849e-1
1.00000000   5.00000003213983e-1

    t              Value X[2]
0.00000000  -1.00000000000000e+0
0.10000000  -8.26446283273189e-1
0.20000000  -6.94444446826215e-1
0.30000000  -5.91715977923112e-1
0.40000000  -5.10204082090465e-1
0.50000000  -4.44444443661452e-1
0.60000000  -3.90624997971428e-1
0.70000000  -3.46020758007956e-1
0.80000000  -3.08641970911504e-1
0.90000000  -2.77008304743045e-1
1.00000000  -2.49999993429933e-1

    t              Value X[3]
0.00000000   2.00000000000000e+0
0.10000000   1.50262961438149e+0
0.20000000   1.15740742373768e+0
0.30000000   9.10332288053840e-1
0.40000000   7.28862989793594e-1
0.50000000   5.92592607536866e-1
0.60000000   4.88281263842229e-1
0.70000000   4.07083261374879e-1
0.80000000   3.42935540127152e-1
0.90000000   2.91587706310718e-1
1.00000000   2.50000010753536e-1

    t              Value X[4]
0.00000000  -6.00000000000000e+0
0.10000000  -4.09808076056272e+0
0.20000000  -2.89351855059016e+0
0.30000000  -2.10076680857258e+0
0.40000000  -1.56184925333600e+0
0.50000000  -1.18518520443061e+0
0.60000000  -9.15527359078898e-1
0.70000000  -7.18382215400418e-1
0.80000000  -5.71559223064178e-1
0.90000000  -4.60401631119694e-1
1.00000000  -3.75000005740566e-1
```

$X[1]$ are the values of $x(t)$.
$X[2]$ are the values of $dx(t)/dt$.
$X[3]$ are the values of $d^2x(t)/dt^2$.
$X[4]$ are the values of $d^3x(t)/dt^3$.

The exact solution is

$$\begin{aligned} x(t) &= (t+1)^{-1} \\ dx(t)/dt &= -(t+1)^{-2} \\ d^2x(t)/dt^2 &= 2(t+1)^{-3} \\ d^3x(t)/dt^3 &= -6(t+1)^{-4} \end{aligned}$$

$$\begin{aligned} x(1) &= 0.5 \\ dx(1)/dt &= -0.25 \\ d^2x(1)/dt^2 &= 0.25 \\ d^3x(1)/dt^3 &= -0.375 \end{aligned}$$

Solution to an Initial Value Problem for a System of Coupled First-Order Ordinary Differential Equations Using the Runge-Kutta Method (Runge_S1.pas)

Description

This example integrates a system of coupled first-order ordinary differential equations with specified initial conditions using the generalized Runge-Kutta formulas (Burden and Faires 1985, 261–269).

Given m first-order ordinary differential equations in the form

$$
\begin{aligned}
dx_1/dt &= TNTargetF1(t, x_1, x_2, \ldots, x_m)\\
dx_2/dt &= TNTargetF2(t, x_1, x_2, \ldots, x_m)\\
&\;\;\vdots\\
dx_m/dt &= TNTargetFm(t, x_1, x_2, \ldots, x_m)
\end{aligned}
$$

which satisfies the Lipshitz condition (the Lipshitz condition for first-order and second-order ordinary differential equations is given at the beginning of this chapter; consult the previous book reference for details of the Lipshitz condition for systems), and initial conditions

$$
\begin{aligned}
x_1[LowerLimit] &= a_1\\
x_2[LowerLimit] &= a_2\\
&\;\;\vdots\\
x_m[LowerLimit] &= a_m
\end{aligned}
$$

and spacing

$$h = (UpperLimit - LowerLimit)/NumIntervals$$

the fourth-order general Runge-Kutta method can be used to approximate simultaneously the x_j's.

The general Runge-Kutta formulas for these equations are as follows:

$$F1x_1 = h * TNTargetF1(t, x_1[t], x_2[t], ..., x_m[t])$$
$$F1x_2 = h * TNTargetF2(t, x_1[t], x_2[t], ..., x_m[t])$$
$$\vdots$$
$$F1x_m = h * TNTargetFm(t, x_1[t], x_2[t], ..., x_m[t])$$
$$F2x_1 = h * TNTargetF1(t + h/2, x_1[t] + F1x_1/2, x_2[t] + F1x_2/2, ..., x_m[t] + F1x_m/2)$$
$$F2x_2 = h * TNTargetF2(t + h/2, x_1[t] + F1x_1/2, x_2[t] + F1x_2/2, ..., x_m[t] + F1x_m/2)$$
$$\vdots$$
$$F2x_m = h * TNTargetFm(t + h/2, x_1[t] + F1x_1/2, x_2[t] + F1x_2/2, ..., x_m[t] + F1x_m/2)$$
$$F3x_1 = h * TNTargetF1(t + h/2, x_1[t] + F2x_1/2, x_2[t] + F2x_2/2, ..., x_m[t] + F2x_m/2)$$
$$F3x_2 = h * TNTargetF2(t + h/2, x_1[t] + F2x_1/2, x_2[t] + F2x_2/2, ..., x_m[t] + F2x_m/2)$$
$$\vdots$$
$$F3x_m = h * TNTargetFm(t + h/2, x_1[t] + F2x_1/2, x_2[t] + F2x_2/2, ..., x_m[t] + F2x_m/2)$$
$$F4x_1 = h * TNTargetF1(t + h, x_1[t] + F3x_1, x_2[t] + F3x_2, ..., x_m[t] + F3x_m)$$
$$F4x_2 = h * TNTargetF2(t + h, x_1[t] + F3x_1, x_2[t] + F3x_2, ..., x_m[t] + F3x_m)$$
$$\vdots$$
$$F4x_m = h * TNTargetFm(t + h, x_1[t] + F3x_1, x_2[t] + F3x_2, ..., x_m[t] + F3x_m)$$
$$x_1[t+1] = x_1[t] + (F1x_1 + 2*F2x_1 + 2*F3x_1 + F4x_1)/6$$
$$x_2[t+1] = x_2[t] + (F1x_2 + 2*F2x_2 + 2*F3x_2 + F4x_2)/6$$
$$\vdots$$
$$x_m[t+1] = x_m[t] + (F1x_m + 2*F2x_m + 2*F3x_m + F4x_m)/6$$

where *t* ranges from *LowerLimit* to *UpperLimit* in steps of *h*. These formulas give a truncation error of order h^4.

You must supply the number of differential equations, the limits, initial values, and *TNTargetF*'s.

This procedure can solve a system of up to ten differential equations (see "Comments" for information about how to increase this limit).

User-Defined Types

```
TNvector = array[0..TNRowSize] of Extended;
TNmatrix = array[0..TNColumnSize] of TNvector;
```

TNRowSize is an upper bound for the number of values returned for a particular variable (*NumReturn*). *TNColumnSize* is an upper bound for the number of differential equations (*NumEquations*).

User-Defined Functions

```
function TNTargetF1(V : TNvector) : Extended;

function TNTargetF2(V : TNvector) : Extended;

function TNTargetF3(V : TNvector) : Extended;

function TNTargetF4(V : TNvector) : Extended;

function TNTargetF5(V : TNvector) : Extended;

function TNTargetF6(V : TNvector) : Extended;

function TNTargetF7(V : TNvector) : Extended;

function TNTargetF8(V : TNvector) : Extended;

function TNTargetF9(V : TNvector) : Extended;

function TNTargetF10(V : TNvector) : Extended;
```

These are the differential equations:

$$dx_j/dt = TNTargetFj(t, x_1, x_2, ..., x_{10})$$

where *j* ranges from 1 to 10.

The elements of the vector V are defined as follows:

$V[0] = t$
$V[1] = x_1$
$V[2] = x_2$

$\cdot$

$\cdot$

$\cdot$

$V[10] = x_{10}$

The procedure *InitialConditionSystem* solves this system of coupled differential equations (a maximum of ten equations). All ten functions must be defined, even if your system contains less than ten equations.

Input Parameters

`NumEquations : Integer;`	Number of first-order differential equations
`LowerLimit : Extended;`	Lower limit of interval
`UpperLimit : Extended;`	Upper limit of interval
`InitialValues : TNvector;`	Values of $x_1, x_2, ..., x_m$ at *LowerLimit*
`NumReturn : Integer;`	Number of $(t, x_1, x_2, ..., x_m)$ values returned from the procedure
`NumIntervals : Integer;`	Number of subintervals used in the calculations
`FuncVect : array[1..10] of ProcPtr;`	Pointers to the ten equations

The preceding parameters must satisfy the following conditions:

1. *NumReturn* > 0
2. *NumIntervals* $\geq$ *NumReturn*
3. *NumEquations* > 0
4. *LowerLimit* $\neq$ *UpperLimit*

Output Parameters

`SolutionValues : TNmatrix;`	Values of t, x_1, x_2, ... x_m between the limits
`Error : Byte;`	0: No errors
	1: *NumReturn* < 1
	2: *NumIntervals* < *NumReturn*
	3: *NumEquations* < 1
	4: *LowerLimit* = *UpperLimit*

Syntax of the Procedure Call

```
InitialConditionSystem(NumEquations, LowerLimit, UpperLimit,
                       InitialValues, NumReturn, NumIntervals,
                       SolutionValues, Error, FuncVect);
```

Comments

The first row of *SolutionValues* will be the values of t between the limits, the second row of *SolutionValues* will be the values of x_1 between the limits, the third row of *SolutionValues* will be the values of x_2 between the limits, and so on.

All ten user-defined functions are called from the procedure. If your system has less than ten equations, you must still define all ten functions or the program will not compile. The superfluous functions should be defined as follows (*TNTargetF*10 is used as an example):

```
function TNTargetF10(V : TNvector) : Extended;

  begin
    TNTargetF10 : = 0.0;
  end;          { function TNTargetF10 }
```

If you need to solve a system with more than ten equations, then edit the include file Runge_S1.pas. The following line should be added to the end of procedure *Step*:

```
F[11] := Spacing * TNTargetF11(CurrentValues);
```

More statements (for *F*[12], and so on) may be added as necessary. All new functions (for example, *TNTargetF*11) must be defined in your top-level program. **Note:** Before making any changes to the include file, make sure you have a backup copy.

This procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use these values. The rows of *SolutionValues* will contain only *NumReturn* values at roughly equally spaced t-values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

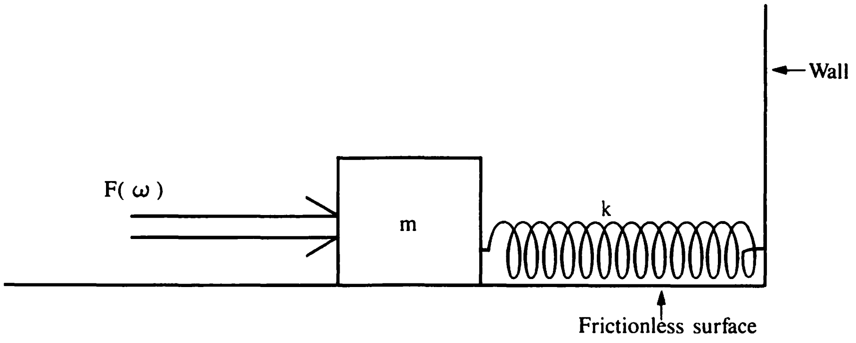
Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program Runge_S1.pas provides I/O functions that demonstrate the Runge-Kutta method of solving initial value problems for systems of first-order ordinary differential equations. Note that the addresses of the ten equations get passed into the procedure *InitialConditionSystem* in the variable *FuncVect*.

Example

Problem. A weight with mass m lays on a frictionless table and is connected to a spring with spring constant k:



If the mass is subject to a driving force $F \sin(\omega t)$ (ω represents the frequency of the driving force and t is time), the equation of motion of the mass is as follows:

$$m\, d^2x/dt^2 + k\, x = F \sin(\omega t)$$

Given

$m = 2$ kg
$F = 9$ N
$k = 32$ N/m
$\omega = 5$ cycles/sec
$x(0) = 0$ m
$dx(0)/dt = -2.5$ m/sec

find the position and velocity of the block from $t = 0$ second to $t = 2$ seconds.

1. Write the second-order ordinary differential equations as a system of two coupled first-order ordinary differential equations:

   $dx_1/dt = x_2$
   $dx_2/dt = (F/m) \sin(\omega t) - (k/m)\, x_1$

2. Code these equations into the program Runge_S1.pas:

```
function TNTargetF1(V : TNvector) : Extended;

{-----------------------------------------------}
{ THIS IS THE FIRST DIFFERENTIAL EQUATION       }
{-----------------------------------------------}
{                                               }
{  dx[1]                                        }
{  -----   = TNTargetF1(t, x[1], x[2], ... x[m]) }
{   dt                                          }
{                                               }
{ The vector V is defined:                      }
{      V[0] = t                                 }
{      V[1] = X[1]                              }
{      V[2] = X[2]                              }
{           .                                   }
{           .                                   }
{           .                                   }
{      V[m] = X[m]                              }
{                                               }
{ where m is the number of coupled equations.   }
{-----------------------------------------------}


begin
  TNTargetF1 := V[2];
end;                        { function TNTargetF1 }
```

```
function TNTargetF2(V : TNvector) : Extended;

{-----------------------------------------------}
{   THIS IS THE SECOND DIFFERENTIAL EQUATION     }
{-----------------------------------------------}
{                                               }
{  dx[2]                                        }
{  -----  = TNTargetF2(t, x[1], x[2], ... x[m]) }
{   dt                                          }
{                                               }
{ The vector V is defined:                      }
{      V[0] = t                                 }
{      V[1] = X[1]                              }
{      V[2] = X[2]                              }
{           .                                   }
{           .                                   }
{           .                                   }
{      V[m] = X[m]                              }
{                                               }
{ where m is the number of coupled equations.   }
{-----------------------------------------------}

begin
  TNTargetF2 := 9/2 * Sin(5 * V[0]) - 32/2 * V[1];
end;                        { function TNTargetF2 }

function TNTargetF3(V : TNvector) : Extended;

{-----------------------------------------------}
{    THIS IS THE THIRD DIFFERENTIAL EQUATION     }
{-----------------------------------------------}
{                                               }
{  dx[3]                                        }
{  -----  = TNTargetF3(t, x[1], x[2], ... x[m]) }
{   dt                                          }
{                                               }
{ The vector V is defined:                      }
{      V[0] = t                                 }
{      V[1] = X[1]                              }
{      V[2] = X[2]                              }
{           .                                   }
{           .                                   }
{           .                                   }
{      V[m] = X[m]                              }
{                                               }
{ where m is the number of coupled equations.   }
{-----------------------------------------------}

begin
  TNTargetF3 : = 0.0;
end;                        { function TNTargetF3 }
```

Functions *TNTarget4* to *TNTarget10* should be defined like the function *TNTargetF3*.

3. Run Runge_S1.pas:

```
Number of first order equations: (1-40)? 2

Lower limit of interval? 0

Upper limit of interval? 2

Enter X[1] value at t = 0.00000e+0:  0
Enter X[2] value at t = 0.00000e+0: -2.5

Number of values to return (1-40)? 10

Number of intervals (> = 10)? 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
                      Lower Limit: 0.00000000000000e+0
                      Upper Limit: 2.00000000000000e+0
               Number of intervals: 100

Initial conditions at lower limit:
                 X[1]= 0.00000000000000e+0
                 X[2]=-2.50000000000000e+0

       T            Value X[1]
   0.00000000    0.00000000000000e+0
   0.20000000   -4.20735284275848e-1
   0.40000000   -4.54648724216734e-1
   0.60000000   -7.05605786993375e-2
   0.80000000    3.78400378699554e-1
   1.00000000    4.79461767300631e-1
   1.20000000    1.39708469016312e-1
   1.40000000   -3.28491796183335e-1
   1.60000000   -4.94677974769031e-1
   1.80000000   -2.06059519715177e-1
   2.00000000    2.72008842396951e-1

       T            Value X[2]
   0.00000000   -2.50000000000000e+0
   0.20000000   -1.35075642830665e+0
   0.40000000    1.04036531118478e+0
   0.60000000    2.47497991717220e+0
   0.80000000    1.63411037473655e+0
   1.00000000   -7.09151289407566e-1
   1.20000000   -2.40042152228323e+0
   1.40000000   -1.88475529635975e+0
   1.60000000    3.63745224811835e-1
   1.80000000    2.27781864414105e+0
   2.00000000    2.09767516082022e+0
```

X[1] are the values of $x(t)$, the position. *X*[2] are the values of $dx(t)/dt$, the velocity.

The exact solution is

$$x = \frac{F \sin(\omega t)}{m\,(\omega_o^2 - \omega^2)}$$

$$dx/dt = \frac{F_\omega \cos(\omega t)}{m\,(\omega_o^2 - \omega^2)}$$

where ω_o is the natural frequency of the system:

$\omega_o^2 = k/m$

The period of oscillation is given by

$T = 2\pi/\omega = 1.257$ sec

The data is taken from a function of which the derivative could be computed exactly. The actual values are as follows:

t	Values of X	Derivative of X
0.0	0.000000000000E+000	−2.500000000000E+000
0.2	−4.207354924039E−001	−1.350755764670E+000
0.4	−4.546487134128E−001	1.040367091367E+000
0.6	−7.056000402993E−002	2.474981241501E+000
0.8	3.784012476539E−001	1.634109052159E+000
1.0	4.794621373315E−001	−7.091554636580E−001
1.2	1.397077490994E−001	−2.400425716625E+000
1.4	−3.284932993593E−001	−1.884755635858E+000
1.6	−4.946791233116E−001	3.637500845215E−001
1.8	−2.060592426208E−001	2.277825654711E+000
2.0	2.720105554446E−001	2.097678822691E+000

Solution to an Initial Value Problem for a System of Coupled Second-Order Ordinary Differential Equations Using the Runge-Kutta Method (Runge_S2.pas)

Description

This example integrates a system of coupled second-order ordinary differential equations with specified initial conditions using the generalized Runge-Kutta formulas (Burden and Faires 1985, 261–269).

Given m coupled second-order ordinary differential equations of the form

$$d^2x_1/dt^2 = TNTargetF1(t, x_1, x'_1, x_2, x'_2, \ldots, x_m, x'_m)$$
$$d^2x_2/dt^2 = TNTargetF2(t, x_1, x'_1, x_2, x'_2, \ldots, x_m, x'_m)$$
$$\vdots$$
$$d^2x_m/dt^2 = TNTargetFm(t, x_1, x'_1, x_2, x'_2, \ldots, x_m, x'_m)$$

where x'_j indicates dx_j/dt, which satisfies the Lipshitz condition (the Lipshitz condition for first-order and second-order ordinary differential equations is given at the beginning of this chapter; consult the previous book reference for details of the Lipshitz condition for systems), and initial condition

$$x_1[LowerLimit] = a_1 \qquad x'_1[LowerLimit] = b_1$$
$$x_2[LowerLimit] = a_2 \qquad x'_2[LowerLimit] = b_2$$
$$\vdots$$
$$x_m[LowerLimit] = a_m \qquad x'_m[LowerLimit] = b_m$$

and spacing

$$h = (UpperLimit - LowerLimit)/NumIntervals$$

rewrite each of the second-order differential equations as two, first-order differential equations:

$$dx_1/dt = y_1$$
$$dy_1/dt = TNTargetF1(t, x_1, y_1, x_2, y_2, ..., x_m, y_m)$$
$$dx_2/dt = y_2$$
$$dx_2/dt = TNTargetF2(t, x_1, y_1, x_2, y_2, ..., x_m, y_m)$$

.

.

.

$$dx_m/dt = y_m$$
$$dx_m/dt = TNTargetFm(t, x_1, y_1, x_2, y_2, ..., x_m, y_m)$$

Then the fourth-order general Runge-Kutta method can be used to approximate the x_j's and the y_j's simultaneously.

The general Runge-Kutta formulas for these equations are as follows:

$$F1x_1 = h * y_1$$
$$F1y_1 = h * TNTargetF1(t, x_1[t], y_1[t], x_2[t], y_2[t], ..., x_m[t], y_m[t])$$
$$F1x_2 = h * y_2$$
$$F1y_2 = h * TNTargetF2(t, x_1[t], y_1[t], x_2[t], y_2[t], ..., x_m[t], y_m[t])$$

.

.

.

$$F1x_m = h * y_m$$
$$F1y_m = h * TNTargetFm(t, x_1[t], y_1[t], x_2[t], y_2[t], ..., x_m[t], y_m[t])$$

$$F2x_1 = h * (y_1 + F1y_1/2)$$
$$F2y_1 = h * NTargetF1(t + h/2, x_1[t] + F1x_1/2, y_1[t] + F1y_1/2, x_2[t] + F1x_2/2, y_2[t] + F1y_2/2, ..., x_m[t] + F1x_m/2, y_m[t] + F1y_m/2)$$
$$F2x_2 = h * (y_2 + F1y_2/2)$$
$$F2y_2 = h * NTargetF2(t + h/2, x_1[t] + F1x_1/2, y_1[t] + F1y_1/2, x_2[t] + F1x_2/2, y_2[t] + F1y_2/2, ..., x_m[t] + F1x_m/2, y_m[t] + F1y_m/2)$$

.

.

.

$$F2x_m = h * (y_m + F1y_m/2)$$
$$F2y_m = h * TNTargetFm(t + h/2, x_1[t] + F1x_1/2, y_1[t] + F1y_1/2, x_2[t] + F1x_2/2, y_2[t] + F1y_2/2, ..., x_m[t] + F1x_m/2, y_m[t] + F1y_m/2)$$

$$
\begin{aligned}
F3x_1 &= h * (y_1 + F2y_1/2)\\
F3y_1 &= h * TNTargetF1(t + h/2, x_1[t] + F2x_1/2, y_1[t] + F2y_1/2, x_2[t]\\
&\quad + F2x_2/2, y_2[t] + F2y_2/2, \ldots, x_m[t] + F2x_m/2, y_m[t] + F2y_m/2)\\
F3x_2 &= h * (y_2 + F2y_2/2)\\
F3y_2 &= h * NTargetF2(t + h/2, x_1[t] + F2x_1/2, y_1[t] + F2y_1/2, x_2[t]\\
&\quad + F2x_2/2, y_2[t] + F2y_2/2, \ldots, x_m[t] + F2x_m/2, y_m[t] + F2y_m/2)
\end{aligned}
$$

.

.

.

$$
\begin{aligned}
F3x_m &= h * (y_m + F2y_m/2)\\
F3y_m &= h * TNTargetFm(t + h/2, x_1[t] + F2x_1/2, y_1[t] + F2y_1/2, x_2[t]\\
&\quad + F2x_2/2, y_2[t] + F2y_2/2, \ldots, x_m[t] + F2x_m/2, y_m[t] + F2y_m/2)
\end{aligned}
$$

$$
\begin{aligned}
F4x_1 &= h * (y_1 + F3y_1)\\
F4y_1 &= h * TNTargetF1(t + h, x_1[t] + F3x_1, y_1[t] + F3y_1, x_2[t] + F3x_2, y_2[t]\\
&\quad + F3y_2, \ldots, x_m[t] + F3x_m, y_m[t] + F3y_m)\\
F4x_2 &= h * (y_2 + F3y_2)\\
F4y_2 &= h * TNTargetF2(t + h, x_1[t] + F3x_1, y_1[t] + F3y_1, x_2[t] + F3x_2, y_2[t]\\
&\quad + F3y_2, \ldots, x_m[t] + F3x_m, y_m[t] + F3y_m)
\end{aligned}
$$

.

.

.

$$
\begin{aligned}
F4x_m &= h * (y_m + F3y_m)\\
F4y_m &= h * TNTargetFm(t + h, x_1[t] + F3x_1, y_1[t] + F3y_1, x_2[t] + F3x_2, y_2[t]\\
&\quad + F3y_2, \ldots, x_m[t] + F3x_m, y_m[t] + F3y_m)
\end{aligned}
$$

$$
\begin{aligned}
x_1[t+1] &= x_1[t] + (F1x_1 + 2 * F2x_1 + 2 * F3x_1 + F4x_1)/6\\
y_1[t+1] &= y_1[t] + (F1y_1 + 2 * F2y_1 + 2 * F3y_1 + F4y_1)/6\\
x_2[t+1] &= x_2[t] + (F1x_2 + 2 * F2x_2 + 2 * F3x_2 + F4x_2)/6\\
y_2[t+1] &= y_2[t] + (F1y_2 + 2 * F2y_2 + 2 * F3y_2 + F4y_2)/6
\end{aligned}
$$

.

.

$$
\begin{aligned}
x_m[t+1] &= x_m[t] + (F1x_m + 2 * F2x_m + 2 * F3x_m + F4x_m)/6\\
y_m[t+1] &= y_m[t] + (F1y_m + 2 * F2y_m + 2 * F3y_m + F4y_m)/6
\end{aligned}
$$

where t ranges from *LowerLimit* to *UpperLimit* in steps of h. These formulas give a truncation error of order h^4.

You must supply the number of equations, limits, initial values, and *TNTargetF*'s.

This procedure can solve a system of up to ten, second-order ordinary differential equations (see "Comments" for information about how to increase this limit).

User-Defined Types

```
TNData = record
           x : Extended;
           xDeriv : Extended;
         end;                   { TNData record }
TNvector = array[0..TNRowSize] of TNData;
TNmatrix = array[0..TNColumnSize] of TNvector;
```

TNRowSize is an upper bound for the number of values returned for a particular variable (*NumReturn*). *TNColumnSize* is an upper bound for the number of second-order differential equations (*NumEquations*).

User-Defined Functions

```
function TNTargetF1(V : TNvector) : Extended;

function TNTargetF2(V : TNvector) : Extended;

function TNTargetF3(V : TNvector) : Extended;

function TNTargetF4(V : TNvector) : Extended;

function TNTargetF5(V : TNvector) : Extended;

function TNTargetF6(V : TNvector) : Extended;

function TNTargetF7(V : TNvector) : Extended;

function TNTargetF8(V : TNvector) : Extended;

function TNTargetF9(V : TNvector) : Extended;

function TNTargetF10(V : TNvector) : Extended;
```

Here are the differential equations:

$$d^2x_j/dt^2 = TNTargetFj(t, x_1, x'_1, x_2, x'_2, ..., x_{10}, x'_{10})$$

where j ranges from 1 to 10.

The elements of the vector *V* are defined as follows:

$$\begin{aligned} V[0].x &= t \\ V[1].x &= x[1] \\ V[1].xDeriv &= x'[1] \\ V[2].x &= x[2] \\ V[2].xDeriv &= x'[2] \\ &\vdots \\ V[10].x &= x[10] \\ V[10].xDeriv &= x'[10] \end{aligned}$$

The procedure used in Runge_S2.pas solves this system of coupled differential equations (a maximum of ten equations). All ten functions must be defined, even if your system contains less than ten equations.

Input Parameters

`NumEquations : Integer;`	Number of second-order differential equations
`LowerLimit : Extended;`	Lower limit of interval
`UpperLimit : Extended;`	Upper limit of interval
`InitialValues : TNvector2;`	Values of x_j's and x'_j's at *LowerLimit*
`NumReturn : Integer;`	Number of $(t, x_1, x'_1, x_2, x'_2, ..., x_m, x'_m)$ values returned from the procedure
`NumIntervals : Integer;`	Number of subintervals used in the calculations
`FuncVect : array[1..10] of ProcPtr;`	Pointers to the ten equations

The preceding parameters must satisfy the following conditions:

1. *NumReturn* > 0
2. *NumIntervals* $\geq$ *NumReturn*
3. *NumEquations* > 0
4. *LowerLimit* $\neq$ *UpperLimit*

Output Parameters

`SolutionValues : TNmatrix2;`	Values of t, x_j, and x'_j between the limits
`Error : Byte;`	0: No errors
	1: *NumReturn* $<$ 1
	2: *NumIntervals* $<$ *NumReturn*
	3: *NumEquations* $<$ 1
	4: *LowerLimit* $=$ *UpperLimit*

Syntax of the Procedure Call

```
InitialConditionSystem2(NumEquations, LowerLimit, UpperLimit,
                        InitialValues, NumReturn, NumIntervals,
                        SolutionValues, Error, FuncVect);
```

Comments

The first row of *SolutionValues* will be the values of t between the limits, the second row of *SolutionValues* will be the values of x_1 and x'_1 between the limits, the third row of *SolutionValues* will be the values of x_2 and x'_2 between the limits, and so on.

All ten user-defined functions are called from the procedure. If your system has less than ten equations, you must still define all ten functions or the program will not compile. The superfluous functions should be defined as follows (*TNTargetF*10 is used as an example):

```
function TNTargetF10(V : TNvector) : Extended;

  begin
    TNTargetF10 : = 0.0
  end;          { function TNTargetF10 }
```

If you need to solve a system with more than ten equations, then edit the source code for the *InitialValRoutines* unit. The following lines should be added to the end of procedure *Step*:

```
F[11].xDeriv := Spacing * CurrentValues[11].xDeriv;
F[11].x := Spacing * TNTargetF11(CurrentValues);
```

More statements (for F[12], and so on) may be added as necessary. All new functions (for example, *TNTargetF*11) must be defined in your top-level program. **Note:** Before making any changes to the include file, make sure you have a backup copy.

The procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use these values. The rows of *SolutionValues* will contain only *NumReturn* values at roughly equally spaced t-values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

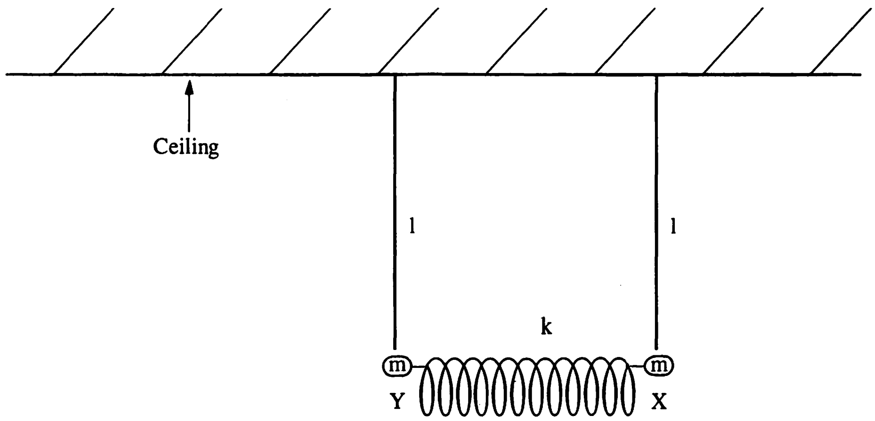
Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program Runge_S2.pas provides I/O functions that demonstrate the Runge-Kutta method of solving initial value problems for systems of first-order ordinary differential equations. Note that the addresses of the ten equations gets passed to the *InitialConditionSystem2* procedure in the variable *FuncVect*.

Example

Problem. Two weights of mass m each hang from a pendulum of length l and are connected by a spring with spring constant k:



The equations of motion of these two masses are as follows:

$$m\, d^2x/dt^2 = -m\, g\, x/l - k(x - y)$$
$$m\, d^2y/dt^2 = -m\, g\, y/l + k(x - y)$$

where g is the acceleration due to gravity, t is time, and x and y are the displacements of the two weights from their rest positions. Given

$m = 2$ kg
$k = 32$ N/m
$g = 9.8$ m/sec^2
$l = 0.6125$ m
$x(0) = 1$ m
$y(0) = -1$ m
$dx(0)/dt = 0$ m/sec
$dy(0)/dt = 0$ m/sec

find the positions and velocities of the two weights from $t = 0$ second to $t = 2$ seconds.

1. Rewrite the equations of motion as shown here:

$$d^2x/dt^2 = -g\, x/l - k/m(x - y)$$
$$d^2y/dt^2 = -g\, y/l + k/m(x - y)$$

2. Code these equations into the program Runge_S2.pas:

```
function TNTargetF1(V : TNvector) : Extended;
{---------------------------------------------------}
{    THIS IS THE FIRST DIFFERENTIAL EQUATION        }
{---------------------------------------------------}
{                                                   }
{                                                   }
{  d² x[1]                                          }
{  ------  = TNTargetF1(t, x[1], x'[1], x[2], x'[2], }
{                      ..., x[m], x'[m]             }
{                                                   }
{    dt²                                            }
{                                                   }
{   The elements of the vector V are defined:       }
{         V[0].x = t                                }
{         V[1].x = X[1]                             }
{    V[1].xDeriv = X'[1]                            }
{         V[2].x = X[2]                             }
{    V[2].xDeriv = X'[2]                            }
{           .                                       }
{           .                                       }
{           .                                       }
{         V[m].x = X[m]                             }
{    V[m].xDeriv = X'[m]                            }
{                                                   }
{ where m is the number of coupled equations.       }
{---------------------------------------------------}

var
  t : Extended;

begin
  t := v[0].x;
  TNTargetF1 := -9.8 * V[1].x/0.6125 - 32/2 * (V[1].x - V[2].x);
end;                                  { function TNTargetF1 }
```

```
function TNTargetF2(V : TNvector) : Extended;

{---------------------------------------------------}
{  THIS IS THE SECOND DIFFERENTIAL EQUATION          }
{---------------------------------------------------}
{                                                    }
{                                                    }
{  d²x[2]                                            }
{  ------  = TNTargetF2(t, x[1], x'[1], x[2], x'[2], }
{                      ..., x[m], x'[m]              }
{   dt²                                              }
{                                                    }
{                                                    }
{   The elements of the vector V are defined:        }
{         V[0].x = t                                 }
{         V[1].x = X[1]                              }
{    V[1].xDeriv = X'[1]                             }
{         V[2].x = X[2]                              }
{    V[2].xDeriv = X'[2]                             }
{           .                                        }
{           .                                        }
{           .                                        }
{         V[m].x = X[m]                              }
{    V[m].xDeriv = X'[m]                             }
{                                                    }
{ where m is the number of coupled equations.        }
{---------------------------------------------------}

var
  t : Extended;

begin
  t := v[0].x;
  TNTargetF2 := -9.8 * V[2].x/0.6125 + 32/2 * (V[1].x - V[2].x);
end;                                    { function TNTargetF2 }
```

```
function TNTargetF3(V : TNvector) : Extended;

{-------------------------------------------------------}
{  THIS IS THE THIRD DIFFERENTIAL EQUATION              }
{-------------------------------------------------------}
{                                                       }
{                                                       }
{  d^2 x[3]                                             }
{  ------   = TNTargetF3(t, x[1], x'[1], x[2], x'[2],   }
{                         ..., x[m], x'[m]              }
{   dt^2                                                }
{                                                       }
{                                                       }
{   The elements of the vector V are defined:           }
{         V[0].x = t                                    }
{         V[1].x = X[1]                                 }
{    V[1].xDeriv = X'[1]                                }
{         V[2].x = X[2]                                 }
{    V[2].xDeriv = X'[2]                                }
{           .                                           }
{           .                                           }
{           .                                           }
{         V[m].x = X[m]                                 }
{    V[m].xDeriv = X'[m]                                }
{                                                       }
{ where m is the number of coupled equations.           }
{-------------------------------------------------------}

var
  t : Extended;
begin
  TNTargetF3 : = 0.0;
end;                                { function TNTargetF3 }
```

Functions *TNTargetF4* to *TNTargetF10* should be defined like function *TNTargetF3*.

3. Run Runge_S2.pas:

```
Number of second order equations: (1-20)? 2

Lower limit of interval? 0

Upper limit of interval? 1

Enter  X[1] value at t = 0.00000e+0:  0.01
Enter X'[1] value at t = 0.00000e+0:  0.00
Enter  X[2] value at t = 0.00000e+0: -0.01
Enter X'[2] value at t = 0.00000e+0:  0.00

Number of values to return (1-70)? 10

Number of intervals (>= 10)? 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
                          Lower Limit: 0.00000000000000e+0
                          Upper Limit: 1.00000000000000e+0
                 Number of intervals: 100
Initial conditions at lower limit:
                        X[1]= 1.00000000000000e-2
                       X'[1]= 0.00000000000000e+0
                        X[2]=-1.00000000000000e-2
                       X'[2]= 0.00000000000000e+0

       T                    Value X[1]             Deriv X[1]
   0.00000000          1.00000000000000e-2    0.00000000000000e+0
   0.10000000          7.69447788485895e-3   -4.42511063153028e-2
   0.20000000          1.84099813762452e-3   -6.80978317847279e-2
   0.30000000         -4.86137387553900e-3   -6.05443464988731e-2
   0.40000000         -9.32214486443693e-3   -2.50735962983904e-2
   0.50000000         -9.48443369885918e-3    2.19586991271007e-2
   0.60000000         -5.27340834792187e-3    5.88657408762406e-2
   0.70000000          1.36920877108260e-3    6.86295294795967e-2
   0.80000000          7.38047758874091e-3    4.67479393932010e-2
   0.90000000          9.98857556718864e-3    3.31066873866278e-3
   1.00000000          7.99089728515028e-3   -4.16531651968366e-2

       T                    Value X[2]             Deriv X[2]
   0.00000000         -1.00000000000000e-2    0.00000000000000e+0
   0.10000000         -7.69447788485895e-3    4.42511063153028e-2
   0.20000000         -1.84099813762452e-3    6.80978317847279e-2
   0.30000000          4.86137387553900e-3    6.05443464988731e-2
   0.40000000          9.32214486443693e-3    2.50735962983904e-2
   0.50000000          9.48443369885918e-3   -2.19586991271007e-2
   0.60000000          5.27340834792187e-3   -5.88657408762406e-2
   0.70000000         -1.36920877108260e-3   -6.86295294795967e-2
   0.80000000         -7.38047758874091e-3   -4.67479393932010e-2
   0.90000000         -9.98857556718864e-3   -3.31066873866278e-3
   1.00000000         -7.99089728515028e-3    4.16531651968366e-2
```

The weights move in opposite directions; the system is in one of its normal modes. The natural frequency ω_o is given by the following:

$$\omega_o^2 = g/l + 2k/m$$
$$\omega_o = 6.928 \text{ cycles/sec}$$

Thus the period of oscillation, t, is

$$t = 2\pi/\omega_o$$
$$t = 0.9069 \text{ sec}$$

Solution to Boundary Value Problem for a Second-Order Ordinary Differential Equation Using the Shooting and Runge-Kutta Methods (Shoot2.pas)

Description

This example uses the shooting method to approximate the solution to a second-order ordinary differential equation with specified boundary conditions (Burden and Faires 1985, 526–531).

Given a second-order differential equation (Burden and Faires 1985, 261–269) of the form

$$d^2y/dx^2 = TNTargetF(x, y, y')$$

where y' represents dy/dx, which satisfies the conditions given at the beginning of this chapter, boundary conditions

$$y[LowerLimit] = LowerInitial$$
$$y[UpperLimit] = UpperInitial$$

and spacing

$$h = (UpperLimit - LowerLimit)/NumIntervals$$

and an initial approximation (guess) to the slope at *LowerLimit*

$$y'[LowerLimit] = InitialSlope$$

the shooting method first solves the second-order initial value problem (using the method described in Runge_2.pas). Based on a comparison of the solution at *UpperLimit* with the boundary condition *UpperInitial*, a new approximation to the slope at *LowerLimit* is made. In this way, a new "shot" at the solution is made by observing the result of the previous "shot." Subsequent iterations use information from two previous shots and the secant method (see Chapter 2, "Roots of a Function Using the Secant Method") to approximate the slope at *LowerLimit*. This process is repeated until the fractional difference between successive approximations to the boundary condition at *UpperLimit* is less than a specified tolerance.

You must supply the *LowerLimit*, *UpperLimit*, *LowerInitial*, *UpperInitial*, *InitialSlope*, *NumIntervals*, *Tolerance*, and *TNTargetF*.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

User-Defined Functions

```
TNTargetF(x, y, yPrime : Extended) : Extended;
```

$$d^2y/dx^2 = TNTargetF(x, y, dy/dx)$$

The procedure *Shooting* integrates this second-order differential equation.

Input Parameters

`LowerLimit : Extended;`	Lower limit of interval
`UpperLimit : Extended;`	Upper limit of interval
`LowerInitial : Extended;`	Value of y at *LowerLimit*
`UpperInitial : Extended;`	Value of y at *UpperLimit*
`InitialSlope : Extended;`	Approximation to the slope at *LowerLimit*
`NumReturn : Integer;`	Number of (x, y, y') values returned from the procedure
`Tolerance : Extended;`	Indicates accuracy in solution
`MaxIter : Integer;`	Maximum number of iterations
`NumIntervals : Integer;`	Number of subintervals used in calculations

The preceding parameters must satisfy the following conditions:

1. *NumReturn* > 0
2. *NumIntervals* $\geq$ *NumReturn*
3. *LowerLimit* $\neq$ *UpperLimit*
4. *Tolerance* > 0
5. *MaxIter* > 0

Output Parameters

`Iter : Integer;`	Number of iterations required to reach a solution
`XValues : TNvector;`	Values of x between the limits
`YValues : TNvector;`	Values of y determined at values in *XValues*
`YDerivValues : TNvector;`	Values of the first derivative of y determined at values in *XValues*
`Error : Byte;`	0: No errors 1: *NumReturn* < 1 2: *NumIntervals* < *NumReturn* 3: *LowerLimit* = *UpperLimit* 4: *Tolerance* ≤ 0 5: *MaxIter* ≤ 0 6: *Iter* > *MaxIter* 7: Convergence not possible

Syntax of the Procedure Call

```
Shooting(LowerLimit, UpperLimit, LowerInitial, UpperInitial, InitialSlope,
         NumReturn, Tolerance, MaxIter, NumIntervals, Iter, XValues,
         YValues, YDerivValues, Error, @TNTargetF);
```

Comments

The parameter *Tolerance* can be misleading. The shooting method converges to the initial slope, which satisfies the upper boundary condition. Convergence is achieved when the fractional difference between *UpperInitial* and the upper boundary approximation is less than *Tolerance*. This does not mean that every value between the boundaries has been approximated with the same degree of accuracy. To improve the accuracy of the entire approximation, increase the number of intervals. The example demonstrates the different effects of *Tolerance* and *NumIntervals*.

The shooting algorithm will compute *NumIntervals* values in its calculations. However, you will rarely need to use all those values. The vectors *XValues, YValues*, and *YDerivValues* will contain only *NumReturn* values at roughly equally spaced t-values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

Boundary value problems are notoriously difficult to solve. The shooting method is extremely sensitive to the approximation of the initial slope. If the shooting method does not converge onto a solution (Error 7), run the program with a different value of the initial slope *InitialSlope*. You may also alleviate some stability problems by solving the equation backwards (from *UpperLimit* to *LowerLimit*). Considerable trial and error may be involved before a solution is found.

Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{100x}$. The shooting method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program Shoot2.pas provides I/O functions that demonstrate the shooting method of solving boundary value problems. Note that the address of *TNTargetF* gets passed into the *Shooting* procedure.

Example

Problem. Use the nonlinear shooting method to solve the following boundary value problem:

$$y'' = 192\ \mathrm{sqr}(y/y') \qquad 0 \le x \le 1$$

$$y(1) = 1$$
$$y(2) = 16$$

1. Code the differential equation into the program:

```
function TNTargetF(x : Extended;
                   y : Extended;
                   yPrime : Extended) : Extended;

{----------------------------------------------------------------}
{   THIS IS THE SECOND-ORDER NONLINEAR DIFFERENTIAL EQUATION     }
{----------------------------------------------------------------}

begin
  TNTargetF := 192 * Sqr(y/yPrime);
end;                                      {function TNTargetF}
```

2. Run Shoot2.pas:

```
Lower limit of interval? 0

Upper limit of interval? 1

Enter Y value at X = 0.00000e+0:  1
Enter Y value at X = 1.00000e+0: 16

Enter a guess for the slope at X = 0.00000e+0 : 15

Number of points returned (1-40)? 10

Number of intervals (>= 10)? 10

Tolerance (> 0)? 1E-12

Maximum number of iterations (> 0)? 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
                 Lower Limit: 0.00000000000000e+0
                 Upper Limit: 1.00000000000000e+0
        Value of Y at 0.0000 : 1.00000000000000e+0
        Value of Y at 1.0000 : 1.60000000000000e+1
     Initial slope at 0.0000 : 1.50000000000000e+1
                NumIntervals:  10
                   Tolerance: 1.00000000000000e-12
Maximum number of iterations: 100

        Number of iterations:   6

          X                     Y Value              Derivative of Y
 0.00000000000000e+0    1.00000000000000e+0    4.00053795390884e+0
 1.00000000000000e-1    1.46417721408153e+0    5.32386904044879e+0
 2.00000000000000e-1    2.07370562259973e+0    6.91162114244397e+0
 3.00000000000000e-1    2.85621262766442e+0    8.78752756627335e+0
 4.00000000000000e-1    3.84170902091389e+0    1.09754927855527e+1
 5.00000000000000e-1    5.06259931530967e+0    1.34994802016423e+1
 6.00000000000000e-1    6.55368547624580e+0    1.63834750611955e+1
 7.00000000000000e-1    8.35216836918581e+0    1.96514712240017e+1
 8.00000000000000e-1    1.04976483580762e+1    2.33274661179548e+1
 9.00000000000000e-1    1.30321255669365e+1    2.74354587043772e+1
 1.00000000000000e+0    1.60000000000094e+1    3.19994486182108e+1
```

Now solve the same problem using a smaller spacing, but with a greater tolerance:

```
Lower limit of interval? 0

Upper limit of interval? 1

Enter Y value at X = 0.00000e+0:  1
Enter Y value at X = 1.00000e+0: 16

Enter a guess for the slope at X = 0.00000e+0 : 15

Number of points returned (1-40)? 10

Number of intervals (>= 10)? 100

Tolerance (> 0)? 1E-6

Maximum number of iterations (> 0)? 100
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
                 Lower Limit: 0.00000000000000e+0
                 Upper Limit: 1.00000000000000e+0
        Value of Y at 0.0000 : 1.00000000000000e+0
        Value of Y at 1.0000 : 1.60000000000000e+1
     Initial slope at 0.0000 : 1.50000000000000e+1
                NumIntervals: 100
                   Tolerance: 1.00000000000000e-6
Maximum number of iterations: 100

        Number of iterations:   5

         X                    Y Value             Derivative of Y
 0.00000000000000e+0   1.00000000000000e+0   4.00000062625639e+0
 1.00000000000000e-1   1.46410005120828e+0   5.32400035609946e+0
 2.00000000000000e-1   2.07360008576235e+0   6.91200027103432e+0
 3.00000000000000e-1   2.85610011557157e+0   8.78800025750412e+0
 4.00000000000000e-1   3.84160014547825e+0   1.09760002747783e+1
 5.00000000000000e-1   5.06250017769403e+0   1.35000003070170e+1
 6.00000000000000e-1   6.55360021337285e+0   1.63840003476283e+1
 7.00000000000000e-1   8.35210025321452e+0   1.96520003937080e+1
 8.00000000000000e-1   1.04976002977125e+1   2.33280004439140e+1
 9.00000000000000e-1   1.30321003472617e+1   2.74360004976014e+1
 1.00000000000000e+0   1.60000004022081e+1   3.20000005544507e+1
```

The exact solution is

$$y = (x + 1)^4$$

X	Y Value	Derivative of Y
0.0	1.0000000000	4.000000000
0.1	1.4641000000	5.324000000
0.2	2.0736000000	6.912000000
0.3	2.8561000000	8.788000000
0.4	3.8416000000	1.097600000
0.5	5.0625000000	1.350000000
0.6	6.5536000000	1.638400000
0.7	8.3521000000	1.965200000
0.8	1.0497600000	2.332800000
0.9	1.3032100000	2.743600000
1.0	1.6000000000	3.200000000

Although the tolerance is smaller (that is, more exacting) in the first case, the accuracy of the approximation is greater in the second case. The spacing in the first case is too large to permit a more accurate approximation.

Solution to a Boundary Value Problem for a Second-Order Ordinary Linear Differential Equation Using the Linear Shooting and Runge-Kutta Methods (Linshot2.pas)

Description

This example uses the linear shooting method to approximate the solution to a second-order linear ordinary differential equation with specified boundary conditions (Burden and Faires 1985, 519–524).

Given a second-order differential equation (Burden and Faires 1985, 261–264) of the form

$$d^2y/dx^2 = TNTargetF(x, y, y')$$

which is linear in both y and y', where y' represents dy/dx, and which satisfies the conditions given at the beginning of this chapter, boundary conditions

$$y[LowerLimit] = LowerInitial$$
$$y[UpperLimit] = UpperInitial$$

and spacing

$$h = (UpperLimit - LowerLimit)/NumIntervals$$

the shooting method solves the two initial value problems (see Runge_2.pas):

$$y'[LowerLimit] = 0 \qquad y[LowerLimit] = LowerInitial$$
$$y'[LowerLimit] = 1 \qquad y[LowerLimit] = LowerInitial$$

(These values are particular to this implementation; any other nonidentical set of initial conditions will suffice.) Since neither of these initial values of y' is likely to be correct, the solutions generated are not likely to satisfy the boundary condition at *UpperInitial*. However, because of the linearity of the equation, an appropriate linear combination of these two solutions will be a solution to the boundary value problem. The linear shooting method requires that only two initial value problems be solved, where the ordinary shooting method (Shoot2.pas) requires that an unknown number of initial value problems be solved before the method converges to a solution.

You must supply the *LowerLimit*, *UpperLimit*, *LowerInitial*, *UpperInitial*, *NumIntervals*, and *TNTargetF*.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Extended;
```

User-Defined Functions

```
TNTargetF(x, y, yPrime : Extended) : Extended;
```

$$d^2y/dx^2 = TNTargetF(x, y, dy/dx)$$

The procedure *LinearShooting* integrates this second-order differential equation.

Input Parameters

`LowerLimit : Extended;`	Lower limit of interval
`UpperLimit : Extended;`	Upper limit of interval
`LowerInitial : Extended;`	Value of y at *LowerLimit*
`UpperInitial : Extended;`	Value of y at *UpperLimit*
`NumIntervals : Integer;`	Number of subintervals used in calculations
`NumReturn : Integer;`	Number of (x, y, y') triples returned from the procedure

The preceding parameters must satisfy the following conditions:

1. *NumReturn* > 0
2. *NumIntervals* $\geq$ *NumReturn*
3. *LowerLimit* $\neq$ *UpperLimit*

Output Parameters

`XValues : TNvector;`	Values of x between the limits
`YValues : TNvector;`	Values of y determined at values in *XValues*
`YDerivValues : TNvector;`	Values of the first derivative of y determined at values in *XValues*
`Error : Byte;`	0: No errors 1: *NumReturn* < 1 2: *NumIntervals* $<$ *NumReturn* 3: *LowerLimit* $=$ *UpperLimit* 4: Equation not linear

Syntax of the Procedure Call

```
LinearShooting(LowerLimit, UpperLimit, LowerInitial, UpperInitial,
               NumReturn, NumIntervals, XValues, YValues,
               YDerivValues, Error, @TNTargetF);
```

Comments

If *TNTargetF* is a nonlinear function, the linear shooting algorithm will usually compute a solution (albeit an incorrect one) without returning an error message. Error 4 (nonlinear equation) will be returned in only a few cases where the two initial value problems happen to yield solutions with the same y-value at x = *UpperLimit*.

The procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use these values. The vectors *XValues*, *YValues*, and *YDerivValues* will contain only *NumReturn* values at roughly evenly spaced intervals between the lower and upper limits. (They will be exactly evenly spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program Linshot2.pas provides I/O functions that demonstrate the linear shooting method of solving boundary value problems. Note that the address of *TNTargetF* gets passed into the *LinearShooting* procedure.

Example

Problem. Use the linear shooting method to solve the following boundary value problem:

$$y'' = y'/x - y/\text{sqr}(x) + 1 \qquad 1 = x \le 10$$

$$y(1) = 1$$

$$y(10) = 76.974149$$

1. Code the differential equation into the program Linshot2.pas:

```
function TNTargetF(x : Extended;
                   y : Extended;
              yPrime : Extended) : Extended;

{-----------------------------------------------------------}
{        THIS IS THE SECOND-ORDER DIFFERENTIAL EQUATION     }
{-----------------------------------------------------------}

  begin
    TNTargetF := yPrime/x - y/Sqr(x) + 1;
  end;          { function TNTargetF }
```

2. Run Linshot2.pas:

```
Lower limit of interval?  1

Upper limit of interval? 10

Enter Y value at X = 1.00000e+0:  1
Enter Y value at X = 1.00000e+1: 76.974149

Number of points returned (1-40)? 9

Number of intervals (>= 9)? 9
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
          Lower limit:  1.00000000000000e+0
          Upper limit:  1.00000000000000e+1
   Value of Y at 1.0000:  1.00000000000000e+0
  Value of Y at 10.0000:  7.69741490000000e+1
         NumIntervals:  9.00000000000000e+0

         X                    Y Value          Derivative of Y
1.00000000000000e+0   1.00000000000000e+0  1.00042467674563e+0
2.00000000000000e+0   2.61170356138588e+0  2.30627678512124e+0
3.00000000000000e+0   5.70207271413620e+0  3.90115296191831e+0
4.00000000000000e+0   1.04528257144925e+1  5.61367861126495e+0
5.00000000000000e+0   1.69509897305375e+1  7.39067355864438e+0
6.00000000000000e+0   2.52478687612139e+1  9.20845513089500e+0
7.00000000000000e+0   3.53773649984557e+1  1.10543869346579e+1
8.00000000000000e+0   4.73635728977226e+1  1.29209245920937e+1
9.00000000000000e+0   6.12245068576119e+1  1.48032011472994e+1
1.00000000000000e+1   7.69741490000000e+1  1.66978931711222e+1
```

Now solve the same problem with a spacing of only 0.1:

```
Lower limit of interval?  1

Upper limit of interval? 10

Enter Y value at X = 1.00000e+0:  1
Enter Y value at X = 1.00000e+1: 76.974149

Number of points returned (1-40)? 9

Number of intervals (>= 9)? 90
```

Now a dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
          Lower limit:  1.00000000000000e+0
          Upper limit:  1.00000000000000e+1
   Value of Y at 1.0000:  1.00000000000000e+0
  Value of Y at 10.0000:  7.69741490000000e+1
         NumIntervals:  9.00000000000000e+1

         X                    Y Value            Derivative of Y
1.00000000000000e+0     1.00000000000000e+0  1.00000001942594e+0
2.00000000000000e+0     2.61370547174514e+0  2.30685275847028e+0
3.00000000000000e+0     5.70416298088411e+0  3.90138768358927e+0
4.00000000000000e+0     1.04548224122436e+1  5.61370562650429e+0
5.00000000000000e+0     1.69528103026793e+1  7.39056208402440e+0
6.00000000000000e+0     2.52494430584438e+1  9.20824053324639e+0
7.00000000000000e+0     3.53786288412165e+1  1.10540898579729e+1
8.00000000000000e+0     4.73644675641047e+1  1.29205584690303e+1
9.00000000000000e+0     6.12249787166508e+1  1.48027754364805e+1
1.00000000000000e+1     7.69741490000000e+1  1.66974149235206e+1
```

The exact solution is

$y = x * x - x * ln(x)$

$y(1) = 1$ $\qquad$ $y'(1) = 1$
$y(10) = 7.6974149$ $\qquad$ $y'(10) = 16.6974149$

The second approximation is more accurate.

CHAPTER 9

Least-Squares Approximation

Given a set of data points, this chapter provides routines to model the data with a function of a given type. The most common application of this concept is *linear regression.*

With linear regression, there is some control variable, say X, and some observed variable, say Y. X and Y are known or suspected to have some linear relationship, say

$$Y = a * X + b$$

but the parameters a and b are unknown. Usually there is some experimental error or some other nonlinear influence on Y, so that there are no values of a and b for which the preceding equation holds exactly. The method of regression provides a formula for a and b in terms of the values of X and Y such that the error is minimized. The error is the sum of squares of the errors $(a * X + b - Y)$ on each data point. Except in certain unusual cases, there is exactly one value for a and one value for b that makes this sum of squares the least possible. This is called the *least-squares* solution.

This concept of least squares also applies when more variables are present—then it is often called *multiple regression.* Using this method, you can find the best model for a given set of data that is linear in a given set of other data sets or functions. Models that are nonlinear variables can also be treated as long as the unknown parameters appear linearly.

Least-Squares Approximation (Least.pas)

Description

This model provides a method for finding a least-squares approximation (Cheney and Kincaid 1985, 362–387) to a set of data points (x, y). The approximation must be a linear combination of a set of ***basis vectors***. The functional form of the approximation (polynomial, logarithmic, and so on) is therefore determined by the user, as long as it is represented linearly. (How to represent logarithmic, and other functions linearly is discussed later.)

Given a set of m data points (x, y), an $m \times n$ matrix $(m \geq n)$, A, is constructed, where n is the number of basis vectors in the approximation. The elements of the matrix are

$$A[i, j] = V_j(X_i)$$

where $V_j(X_i)$ is the jth basis vector evaluated at the data value $X[i]$. A vector Y is constructed that contains the y-values of the data points. The coefficients of the basis vectors that form the least-squares approximation will be the n vector C, such that the Euclidean norm of $(AC - Y)$ (represented by $\| AC - Y \|_2$) is a minimum. This requirement is converted to the requirement that

$$\| BC - Z \|_2 + \| R \|_2$$

be a minimum. Here B is an $n \times n$ matrix, Z is an n vector, and R is an $(m - n)$ vector. The equations $BC = Z$ are the normal equations. The previous expression will be minimized when C solves the equation $BC = Z$. Gaussian elimination with partial pivoting (see Chapter 6, "Solving a System of Linear Equations with Gaussian Elimination and Partial Pivoting") is used to solve the normal equations.

The *goodness of fit* is indicated by the standard deviation:

$$\text{S.D.} = ((Y[i] - F(X[i]))^2/(m - n))^{1/2}$$

where $F(X[i])$ is the least-squares solution at the point $X[i]$, $(Y[i] - F(X[i]))$ is the residual, and $(m - n)$ is the degree of freedom of the fit.

User-Defined Types

```
TNColumnVector = array[1..TNColumnSize] of Extended;
TNRowVector = array[1..TNRowSize] of Extended;
```

(*TNColumnSize* will usually be much larger than *TNRowSize*.)

```
TNmatrix = array[1..TNColumnSize] of TNRowVector;
TNSquareMatrix = array[1..TNRowSize] of TNRowVector;
TNString40 = string[40];
FitType = (Expo, Fourier, Log, Poly, Power, User);
```

Input Parameters

`NumPoints : Integer;`	Number of data points
`XData : TNColumnVector;`	*X* coordinates of the data points
`YData : TNColumnVector;`	*Y* coordinates of the data points
`NumTerms : Integer;`	Number of terms in the least-squares approximation
`Fit : FitType;`	Type of least-squares fit requested

The preceding parameters must satisfy the following conditions:

1. *NumPoints* > 1.
2. *NumTerms* ≤ *NumPoints*.
3. *NumPoints* ≤ *TNColumnSize*.
4. *NumTerms* ≤ *TNRowSize*.
5. The *XData* points cannot all be identical.

TNColumnSize and *TNRowSize* set an upper bound on the number of elements in a vector. Neither of these identifiers are variable names and are never referenced by the procedure. If conditions 3 or 4 are violated, the program will crash with an Index Out of Range error (assuming the directive {$R+} is active).

Output Parameters

`Solution : TNRowVector;`	Coefficients of the basis vectors that form the least-squares approximation
`YFit : TNColumnVector;`	Values of the least-squares fit evaluated at the *XData* values
`Residual : TNColumnVector;`	Difference between *YData* and *YFit* values
`StandardDeviation : Extended;`	Square root of the variance—indicates the goodness of fit
`Error : Byte;`	0: No error 1: *NumPoints* < 2 2: *NumTerms* < 1 3: *NumTerms* > *NumPoints* 4: Least-squares solution does not exist (see "Comments")

Syntax of the Procedure Call

```
LeastSquares(NumPoints, XData, YData, NumTerms, Solution,
             YFit, Residual, StandardDeviation, Variance, Error, Fit);
```

Comments

The least-squares routine is defined in LeastSquares.unit. The choice of parameter passed in for *FitType* will depend upon the functional form (basis vectors) to which you fit the data. Following are the five choices for the *FitType* parameter:

Poly

This method uses *Chebyshev polynomials* to fit a polynomial to the data points. *NumTerms* must be one greater than the degree of the polynomial (for example, to fit a fourth-degree polynomial, input *NumTerms* = 5). To get a straight-line least-squares fit, use this module and fit a curve with only two coefficients. The elements of the *Solution* vector will be as follows:

$$Solution[j] = a_j \qquad 1 \leq j \leq NumTerms$$

where a_j is the coefficient of x^{j-1}.

Fourier

This method will fit a finite Fourier series to the data points. The number of terms in the approximation will be *NumTerms*. The elements of the *Solution* vector will be as follows:

$$Solution[j] = F_{j-1} \qquad 1 \le j \le NumTerms$$

where F_{j-1} is the $(j-1)$th term in the Fourier series. Following are the first few terms in the Fourier series:

$$F[0] = 1$$
$$F[1] = \cos(x)$$
$$F[2] = \sin(x)$$
$$F[3] = \cos(2x)$$
$$F[4] = \sin(2x)$$
$$F[5] = \cos(3x)$$
$$F[6] = \sin(3x)$$

Power

This method will fit the function

$$y = ax^b$$

where a and b are real numbers to the data points. A linear equation is obtained by taking the log of both sides, like so:

$$\ln(y) = \ln(a) + b * \ln(x)$$

and expanding on basis vectors 1 and $\ln(x)$. The x-values of the data points must all be positive, and the y-values of the data points must all have the same sign. The number of coefficients in the approximation will be two regardless of the value of *NumTerms* (unless *NumTerms* > *NumPoints*, in which case Error 3 will occur). The elements of the *Solution* vector will be as follows:

$$Solution[1] = a$$
$$Solution[2] = b$$

Expo

This method will fit the function

$$y = ae^{bx}$$

where a and b are real numbers to the data points. A linear equation is obtained by taking the log of both sides, like so:

$$\ln(y) = \ln(a) + bx$$

and expanding on basis vectors 1 and x. The y-values of the data points must all have the same sign. The number of coefficients in the approximation will be two regardless of the value of *NumTerms* (unless *NumTerms* > *NumPoints*, in which case Error 3 will occur). The elements of the *Solution* vector will be as follows:

Solution[1] = a
Solution[2] = b

Log

This method will fit the function

$$y = a \ln(bx)$$

where a and b are real numbers to the data points. A linear equation is obtained by rewriting the equation:

$$y = a \ln(b) + a \ln(x)$$

and expanding on basis vectors 1 and $\ln(x)$. The x-values of the data points must all have the same sign. The number of coefficients in the approximation will be two regardless of the value of *NumTerms* (unless *NumTerms* > *NumPoints*, in which case Error 3 will occur). The elements of the *Solution* vector will be as follows:

Solution[1] = a
Solution[2] = b

User

This method is included if you need a least-squares approximation on a set of basis vectors different from the ones listed earlier. This method allows you to create your own set of basis vectors. The source code contains detailed instructions of how to flesh out the skeleton for the user-defined method.

A least-squares solution may not exist for some input data and choice of basis vectors (Error 4). The reasons for this will depend on the module you are using. For example, it is impossible to fit an exponential function to data with y-values of differing signs; Error 4 will occur if you try. The same data could be fit with a polynomial and no error would result. Error 4 will also occur if all the x-values of the data are identical.

Sample Program

The demonstration program Least.pas contains I/O routines that allow you to run the least-squares approximation routine.

To change the basis vectors of the approximation, simply pass in a different parameter for *FitType* to select the method used.

Input Files

Data may be entered from a text file. The x- and y-coordinates should be separated by a space and followed by a carriage return. For example, data values of sqr(x) could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Example

Problem. Given the following data (contained in the file Sample9A.dat), fit a fourth-degree polynomial and a logarithmic function to the data:

```
0.00000000000000e+0  1.33830225764886e-3
0.10000000000000e+0  4.43184841193803e-2
0.20000000000000e+0  5.39909665131879e-1
0.30000000000000e+0  2.41970724519143e+0
0.40000000000000e+0  3.98942280401433e+0
0.02000000000000e+0  2.91946925791461e-3
0.04000000000000e+0  6.11901930113775e-3
0.06000000000000e+0  1.23221916847303e-2
0.08000000000000e+0  2.38408820146486e-2
0.12000000000000e+0  7.91545158298001e-2
0.14000000000000e+0  1.35829692336855e-1
0.16000000000000e+0  2.23945302948430e-1
0.18000000000000e+0  3.54745928462313e-1
0.22000000000000e+0  7.89501583008939e-1
0.24000000000000e+0  1.10920834679455e+0
0.26000000000000e+0  1.49727465635745e+0
0.28000000000000e+0  1.94186054983213e+0
0.32000000000000e+0  2.89691552761483e+0
0.34000000000000e+0  3.33224602891800e+0
0.36000000000000e+0  3.68270140303323e+0
0.38000000000000e+0  3.91042693975456e+0
```

(The function is the left-hand side of a Gaussian distribution curve with mean = 0.5 and standard deviation = 0.1.) Note that the points do not have to be in any particular order.

First fit the polynomial; set the *FitType* parameter to *Poly* in the call to procedure *LeastSquares.*

Run Least.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample9A.dat

Number of terms in the least squares fit (<= 21)? 5
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The Data Points:
    X                 Y
   0.200           0.0443185
   0.300           0.5399097
   0.400           2.4197072
   0.500           3.9894228
   0.100           0.0013383
   0.120           0.0029195
   0.140           0.0061190
   0.160           0.0123222
   0.180           0.0238409
   0.220           0.0791545
   0.240           0.1358297
   0.260           0.2239453
   0.280           0.3547459
   0.320           0.7895016
   0.340           1.1092083
   0.360           1.4972747
   0.380           1.9418605
   0.420           2.8969155
   0.440           3.3322460
   0.460           3.6827014
   0.480           3.9104269

*----------------------------------------*
       Polynomial Least Squares Fit
*----------------------------------------*

Coefficients in least squares approximation:
   Coefficient 0: -3.1905595418e+0
   Coefficient 1:  6.4048009603e+1
   Coefficient 2: -4.3900537685e+2
   Coefficient 3:  1.2058567475e+3
   Coefficient 4: -1.0523352671e+3
```

```
        X          Least Squares Fit             Residual
     0.2000        2.1944857693e-2           -2.2373626426e-2
     0.3000        5.4757594259e-1            7.6662774599e-3
     0.4000        2.4228330082e+0            3.1257630445e-3
     0.5000        4.0432402964e+0            5.3817492388e-2
     0.1000       -7.5189129206e-2           -7.6527431463e-2
     0.1200        3.9032402642e-2            3.6112933385e-2
     0.1400        7.6262215354e-2            7.0143196053e-2
     0.1600        6.8115144544e-2            5.5792952859e-2
     0.1800        4.2165058402e-2            1.8324176388e-2
     0.2200        2.6946475755e-2           -5.2208040075e-2
     0.2400        7.2620878501e-2           -6.3208813836e-2
     0.2600        1.7037806442e-1           -5.3567238529e-2
     0.2800        3.2758706457e-1           -2.7158863892e-2
     0.3200        8.2963179469e-1            4.0130211685e-2
     0.3400        1.1690007497e+0            5.9792402868e-2
     0.3600        1.5568879689e+0            5.9613312500e-2
     0.3800        1.9804576462e+0            3.8597096380e-2
     0.4200        2.8630963140e+0           -3.3819213603e-2
     0.4400        3.2762888552e+0           -5.5957173721e-2
     0.4600        3.6334109560e+0           -4.9290447009e-2
     0.4800        3.9014219733e+0           -9.0049664558e-3

  Standard Deviation :  5.381534e-2
```

The fourth-degree polynomial that best fits this data is as follows:

$$y = -1052.34\,x^4 + 1205.86\,x^3 - 439.005\,x^2 + 64.0480\,x - 3.19056$$

Note that a fourth-degree polynomial requires five terms in the fit.

Now fit the logarithmic function; set the *FitType* parameter to Log in the call to procedure *LeastSquares*.

Run Least.pas:

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample9A.dat

Number of terms in the least squares fit (<= 21)? 2
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
The Data Points:
    X                  Y
   0.200            0.0443185
   0.300            0.5399097
   0.400            2.4197072
   0.500            3.9894228
   0.100            0.0013383
   0.120            0.0029195
   0.140            0.0061190
   0.160            0.0123222
   0.180            0.0238409
   0.220            0.0791545
   0.240            0.1358297
   0.260            0.2239453
   0.280            0.3547459
   0.320            0.7895016
   0.340            1.1092083
   0.360            1.4972747
   0.380            1.9418605
   0.420            2.8969155
   0.440            3.3322460
   0.460            3.6827014
   0.480            3.9104269

*----------------------------------------*
      Logarithmic Least Squares Fit
*----------------------------------------*

Coefficients in least squares approximation:
   Coefficient 0:  2.5984092388e+0
   Coefficient 1:  6.0253489684e+0

    X        Least Squares Fit              Residual
  0.2000     4.8470072527e-1            4.4038224115e-1
  0.3000     1.5382650082e+0            9.9835534307e-1
  0.4000     2.2857807631e+0           -1.3392648209e-1
  0.5000     2.8655990284e+0           -1.1238237756e+0
  0.1000    -1.3163793126e+0           -1.3177176148e+0
  0.1200    -8.4263329495e-1           -8.4555276421e-1
  0.1400    -4.4208674432e-1           -4.4820576362e-1
  0.1600    -9.5117540049e-2           -1.0743973173e-1
  0.1800     2.1093098798e-1            1.8709010596e-1
  0.2200     7.3235557703e-1            6.5320106120e-1
  0.2400     9.5844674288e-1            8.2261705055e-1
  0.2600     1.1664304540e+0            9.4248515105e-1
  0.2800     1.3589932935e+0            1.0042473651e+0
  0.3200     1.7059624978e+0            9.1646091478e-1
  0.3400     1.8634900752e+0            7.5428172842e-1
  0.3600     2.0120110258e+0            5.1473636946e-1
  0.3800     2.1524997931e+0            2.1063924325e-1
  0.4200     2.4125575764e+0           -4.8435795117e-1
  0.4400     2.5334356149e+0           -7.9881041405e-1
  0.4600     2.6489394854e+0           -1.0337619176e+0
  0.4800     2.7595267807e+0           -1.1509001590e+0

Standard Deviation :  8.320742e-1
```

The logarithmic function that best fits this data is as follows:

$$y = 2.59841 * \ln(6.02535x)$$

The standard deviation of the polynomial fit is much smaller than that of the logarithmic fit; a fourth-degree polynomial fits this data much better than a logarithmic function.

C H A P T E R 10

Fast Fourier Transform Routines

Fourier transforms are used to analyze periodic phenomena such as waves. A continuous function f that has period 2π ($= 2 * 3.14159265...$); that is, satisfies

$$f(x + 2\pi) = f(x)$$

for all x, can be decomposed into sines and cosines:

$$f(x) = a[0] + a[1] * \cos(x) + b[1] * \sin(x) + a[2] * \cos(2x) + b[2] * \sin(2x) + ...$$

This is an infinite series where the coefficients get closer and closer to zero. The routines in this chapter can be used to calculate the coefficients.

The *Fast Fourier Transform* (FFT) is a particular algorithm for computing Fourier transforms efficiently.

This chapter includes two kinds of units. One group consists of four variations of the FFT method of calculating discrete Fourier transforms, each optimized for certain conditions. All are variations of the original *Cooley-Tukey* method. The second group consists of six applications: ComplexFFT, RealFFT, ComplexConvolution, RealConvolution, ComplexCrossCorrelation, and RealCrossCorrelation. Each can be used with any of the FFT methods. You can select the FFT method most appropriate to the circumstances and combine it with the appropriate application or integrate it into another program (Brigham 1974; Nussbaumer 1982).

In each FFT unit the procedure calls have exactly the same form (although there are different restrictions on the data) so that any one FFT unit can be combined with any of the application units without rewriting code. Each of these algorithms will compute either a *forward* or an *inverse transform.*

Each unit contains two procedures needed to prepare for the FFT calculation: procedure *TestInput* and procedure *MakeSinCosTable*. *TestInput* examines the input data to ensure that it satisfies certain conditions (for example, that there is more than 1 data point). *MakeSinCosTable* precalculates a table of the *n*th roots of unity for look up in the FFT calculation.

When *Radix2* is passed in for the *RadixType* parameter, the Cooley-Tukey powers-of-two (radix2 or base2) Fast Fourier Transform is used. Complex multiplications are done with four real multiplications and two real additions. By using this standard form of complex multiplication, storage overhead and assignment statements are reduced. This algorithm is appropriate when the time for a real multiplication is close to the time for a real addition.

When *Radix4* is passed in for the *RadixType* parameter, the powers-of-four (radix4 or base4) Fast Fourier Transform is used. The powers-of-four method is the same as the Cooley-Tukey algorithm except at each stage of reduction a given transform is converted into four transforms each with one fourth the data points of its predecessor (Nussbaumer 1982). When this algorithm is optimized, there are about 25 percent fewer multiplications and slightly fewer additions than in a radix-2 algorithm. The algorithm has the disadvantage of only being applicable to data sets where the number of points is a power of four up to a maximum of 4,096 points. A reduction in execution time of about 20 percent is accomplished when *Radix4* is used over its *Radix2* counterpart.

The Application Programs

Fast Fourier Transforms are particularly useful for analyzing periodic signals. Such a signal is represented by a function f satisfying

$$f(t + T) = f(t)$$

where t is time and T is the period. Under mild hypotheses, f can be expanded into a Fourier series such as the following:

$$f(t) = N^{-1/2} \sum_{n=-\infty}^{\infty} F(n) \exp(2\pi\, i\, n\, t/T)$$

where i is the square root of -1. The term $\exp(2\pi\, i\, n\, t/T)$ is a sinusoid of period T/n and frequency n/T, and its coefficient $F(n)$ gives the strength of that frequency component in the original signal.

To analyze a signal on a digital computer, the signal must be *discretized.* Let $x(n)$ be computed by discretizing the function f at N equidistant points in one period. Thus, let

$$x(n) = f(nT/N) \qquad n = 0, 1, \ldots N - 1$$

Once we restrict attention to N points, it only makes sense to represent the signal in terms of N of the functions

$$\exp(2\pi\, i\, n\, t/T)$$

since the rest are redundant. For example:

$$\exp(2\pi\, i\,(-1)\, t/T) = \exp(2\pi\, i\,(N-1)\, t/T)$$

for $t = nT/N$, $n = 0, 1, \ldots N - 1$. The Fourier series for the signal is then a finite sum, and has the form

$$x(n) = N^{-1/2} \sum_{k=0}^{N-1} X(k) \exp(2\pi\, i\, k\, n/N)$$

(The factor of $N^{-1/2}$ is a matter of convention. Some books do not include it in this formula, resulting in a factor of $1/N$ in the formula for X that follows.)

The formula for the coefficients $X(k)$ is as follows:

$$X(k) = N^{-1/2} \sum_{n=0}^{N-1} x(n) \exp(-i\, 2\pi\, n\, k/N)$$

This formula for X makes sense for any integer k. X is then periodic, satisfying

$$X(k + N) = X(k)$$

for all k. In formulas and programs, it is more convenient to let k run from 0 to $N - 1$, but for analyzing signals it makes more sense to think of k as running from $(-N/2)$ to $(N/2 - 1)$. This is because values of k near zero represent the low frequency information, and values of k near or greater than $N/2$ represent frequencies that are so high that the discretization is too coarse to realize them accurately anyway. Therefore, if k is between $N/2$ and N, $X(k)$ should be thought of as the coefficient of

$$\exp(2\pi\, i\,(k-N)\, t/T)$$

rather than

$$\exp(2\pi\, i\, k\, t/T)$$

In other words, negative frequencies are represented on the right half of the transform.

ComplexFFT simply takes the complex Fast Fourier Transform of a set of complex data points. The complex Fourier transform is defined as

$$X_f = N^{-1/2} \sum_{n=0}^{N-1} x_n \exp(2\pi\, i\, n f/N) \qquad f = 0..N-1$$

where i is the square root of -1. The inverse Fourier transform (which may also be calculated with *ComplexFFT*) is defined as

$$\bar{x}_n = N^{-1/2} \sum_{f=0}^{N-1} \bar{X}_f \exp(2\pi\, i f\, n/N) \qquad n = 0..N-1$$

where the bar stands for complex conjugation.

RealFFT provides a procedure that is optimized for a discrete Fourier transform with all real data. It proceeds by mapping the N real data points onto $N/2$ complex points, applying one of the FFT routines, then reconstructing the N points of the desired transform. This reduces the computation time by about 25 percent compared to applying the complex FFT routine to the N real data points. *RealFFT* can be used with any of the given FFT methods, but note that if a radix-4 method is used, $N/2$ must be a power of four; so N must be of the form $2 * 4^k$.

ComplexConvolution provides a procedure for calculating convolutions of two complex vectors (Brigham 1974; Nussbaumer 1982). The discrete convolution of two complex functions x and h is defined by

$$y_m = \sum_{n=0}^{N-1} x_n h_{m-n} \qquad m = 0, 1, \ldots N-1$$

where subscripts are taken *modulo* N (*circular convolution*). The basic theorem that allows us to calculate these effectively using FFTs is shown in the following:

$$Y_m = X_m H_m \qquad m = 0, 1, \ldots N-1$$

where capital letters indicate the transforms of the functions represented by lowercase letters. Thus the procedure for convolution works like this:

1. Transform both given data sets using FFTs.
2. Multiply the resulting transforms point by point.
3. Find the inverse transform of this product using FFTs.

RealConvolution provides a procedure for calculating convolutions of two real vectors (Brigham 1984; Nussbaumer 1982). This procedure is exactly the same as the previous procedure (*ComplexConvolution*) for complex convolution except that only one forward Fourier transform need be performed. The procedure is as follows:

1. Given two real vectors *XReal* and *HReal*, combine them into a complex vector *XReal* plus *iHReal*, where i is the square root of -1.
2. Transform this complex vector.
3. Extract the transforms of the two real functions from the transform of the complex function (using the symmetry $X_f = \overline{X}_{-f}$, where the bar stands for complex conjugation).
4. Multiply the resulting transforms point by point.
5. Find the inverse transform of this product using FFTs. *RealConvolution* is about 25 percent faster than its complex counterpart for the same set of real data.

ComplexCrossCorrelation provides a procedure for calculating the crosscorrelation of two discrete complex functions or the autocorrelation of one discrete complex function (Brigham 1974). If x and h are the given discrete functions, then their correlation is defined as

$$c_m = \sum_{n=0}^{N-1} x_n h_{n+m} \qquad m = 0, 1, \ldots N - 1$$

where subscripts are taken modulo N (circular convolution). This can be computed using FFTs with a method analogous to that used in *ComplexConvolution*:

$$C_m = X_m H_{N-m} \qquad m = 0, 1, \ldots N - 1$$

Commonly x and h are real functions; in which case the preceding formula reduces to $C_m = X_m \overline{H}_m$, where the bar stands for complex conjugation. Thus the procedure for correlation works like this:

1. Transform both given data sets using FFTs.
2. Multiply each element of the transform of the first data set by the appropriate element of the transform of the second data.
3. Find the inverse transform of this product using FFTs.

RealCrossCorrelation provides a procedure for calculating the crosscorrelation of two discrete real functions or the autocorrelation of one discrete real function (Brigham 1974). This procedure is exactly the same as the previous procedure for complex correlation except that only one forward Fourier transform need be performed. The procedure is as follows:

1. Given two real vectors *XReal* and *HReal*, combine them into a complex vector *XReal* + *iHReal*, where *i* is the square root of -1.
2. Transform this complex vector.
3. Extract the transforms of the two real vectors from the transform of the complex vector (using the symmetry $X_f = \overline{X}_{-f}$, where the bar stands for complex conjugation).
4. Multiply each element of the transform of the first data set by the appropriate element of the transform of the second data.
5. Find the inverse transform of this product using FFTs.

Any one of the FFT include files can be used with any of the applications.

Data Sampling

While sampling theory is beyond the scope of this Toolbox, we would like to mention several common problems associated with data sampling (Brigham 1974; Press et al. 1986, Ch.12). The most common frustration is *aliasing*. A Fourier transform only represents frequencies up to a certain limit (called the *Nyquist limit*, or *Nyquist frequency*), which is given by half the sampling rate. (For example, if a signal is sampled sixty times a second, the Nyquist frequency will be 30 Hz.) A sample containing frequencies greater than this limit will not be properly transformed. The high frequencies will falsely contribute to the transform. This contribution will be indistinguishable from a contribution of a frequency below the Nyquist frequency.

There are several ways to combat aliasing. Increasing the sampling rate will increase the Nyquist frequency and thus reduce aliasing effects. It is also possible to pass the signal through a low pass filter, thus eliminating the high frequencies before sampling. If the Fourier transform of a signal does not converge to zero at the Nyquist frequency, the transform has very likely been aliased.

The Fourier transform assumes that the sample represents a periodic function and that the sample is an integer multiple of one period. If the latter condition is not true, spurious frequencies will show up in the transform. For example, if a sine wave is sampled from 0 to 270 degrees (instead of the full period), a sharp boundary is created because the sine of 0 does not equal the sine of 270. High frequencies will be introduced into the transform to account for that sharp boundary.

The assumption of periodicity can cause problems when convolving or correlating two signals that are not periodic. The convolution of each point in a signal affects the points surrounding it (the nature and extent of the affect depends on the particular convolving function). The assumption of periodicity means that the convolution at one end of the signal will affect the other end of the signal. This "end effect" can be eliminated by padding the data (on either end) with a sufficient number of zeros.

User-Defined Types

```
TNvector = array[0..TNArraySize] of Extended;

TNvectorPtr = ^TNvector;

RadixType = (Radix2, Radix4);
```

These user-defined types are different from others in this Toolbox, because they involve pointers. Pointers are used to transcend the limitations imposed by the 32K data segment size of Turbo Pascal. However, it is possible to store these arrays on the heap, and to point to them with pointers that only require 4 bytes. The size of the heap (and hence the maximum size and number of *TNvectors*) is determined by the amount of memory in the machine.

Fast Fourier Transform Algorithms

The following documentation generally applies to all FFT algorithms. When a difference between the radix-2 and radix-4 algorithms needs to be described, the radix-4 information will be placed in brackets following the radix-2 information (for example, the number of points must be a power of two [four]).

Procedure TestInput

Description

This example determines the number of data points in terms of a power of two [four]. If the number of data points is not a power of two [four], then an error is returned.

Input Parameters

`NumPoints : Integer;`	Number of data points

The preceding parameter must satisfy the following conditions:

1. *NumPoints* ≥ 2.
2. *NumPoints* must be a power of two [four].

Output Parameters

`NumberOfBits : Byte;`	Number of data points as a power of two [four]
`Error : Byte;`	0: No errors 1: *NumPoints* < 2 2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

```
TestInput(NumPoints, NumberOfBits, Error);
```

Procedure MakeSinCosTable

Description

This example creates a look-up table of *NumPoints*/2 [3/4 *NumPoints*] roots of unity. The roots of unity are defined as follows:

$$Root_n = \exp(-i\,2\pi\,n/NumPoints \qquad n = 0..NumPoints/2\ [3/4\ NumPoints]$$

where i is the square root of -1. These values are stored in two tables: *SinTable*, containing the imaginary parts of the roots of unity, and *CosTable*, containing the real parts of the roots of unity. It is faster to look up these values in a table than to calculate them in the FFT procedure.

Input Parameters

`NumPoints : Integer;` Number of data points

The preceding parameter must satisfy the following conditions:

1. *NumPoints* $\geq$ 2.
2. *NumPoints* must be a power of two [four].

Output Parameters

`SinTable : TNvectorPtr;` Table of sine values

`CosTable : TNvectorPtr;` Table of cosine values

Syntax of the Procedure Call

```
MakeSinCosTable(NumPoints, SinTable, CosTable);
```

Procedure ComplexFFT, RealFFT

Description

This example implements the particular variation of the Cooley-Tukey algorithm. The Fast Fourier Transform of the data *XReal*, *XImag* is made in place and is thus returned in the vectors *XReal*, *XImag*. The inverse transform of the data can also be calculated with this procedure.

It is essential that procedures *TestInput* and *MakeSinCosTable* be called before procedure *Fast Fourier Transform* is called. *TestInput* will flag any errors in the data (for example, number of points that are not a power of two [four]), and *MakeSinCosTable* generates a table of sine and cosine values referenced by *Fast Fourier Transform*. *TestInput* and *MakeSinCosTable* need only be called once, even if several calls to *Fast Fourier Transform* are made within the same program (for example, when computing the discrete convolution), as long as the number of data points is unchanged. If the number of data points changes between two calls of *Fast Fourier Transform*, *TestInput* and *MakeSinCosTable* must be called again. (Interested readers are urged to consult the references given in the beginning of the chapter for details about the Cooley-Tukey algorithm.)

Input Parameters

`NumberOfBits : Byte;`	Number of data points as a power of two [four]
`NumPoints : Integer;`	Number of data points
`Inverse : Boolean;`	FALSE equals forward transform; TRUE equals inverse transform
`XReal : TNvectorPtr;`	Pointer to real values of the data points
`XImag : TNvectorPtr;`	Pointer to imaginary values of the data points
`SinTable : TNvectorPtr;`	Table of sine values
`CosTable : TNvectorPtr;`	Table of cosine values
`Radix : RadixType;`	*Radix2* or *Radix4*

The preceding parameters must satisfy the following conditions:

1. *NumPoints* $\geq$ 2.
2. *NumPoints* must be a power of two [four].

Output Parameters

`XReal : TNvectorPtr;`	Pointer to real values of the discrete Fourier transform of the input data
`XImag : TNvectorPtr;`	Pointer to imaginary values of the discrete Fourier transform of the input data

Syntax of the Procedure Call

```
RealFFT(NumberOfBits, NumPoints, Inverse, XReal, XImag, SinTable, CosTable, Radix);
ComplexFFT(NumberOfBits, Numpoints, Inverse, XReal, XImag, SinTable, CosTable,
  Radix);
```

Fast Fourier Transform Applications

ComplexFFT

Description

This example is the most basic application, performing a complex Fast Fourier Transform. It simply calls *TestInput*, *MakeSinCosTable*, and *FFT* sequentially; thus accomplishing an in-place transformation of the complex data *XReal*, *XImag*.

Input Parameters

`NumPoints : Integer;`	Number of data points
`Inverse : Boolean;`	FALSE equals forward transform; TRUE equals inverse transform
`XReal : TNvectorPtr;`	Pointer to real values of the data points
`XImag : TNvectorPtr;`	Pointer to imaginary values of the data points
`Radix : RadixType;`	*Radix2* or *Radix4*

The preceding parameters must satisfy the following conditions:

1. *NumPoints* ≥ 2.
2. *NumPoints* must be a power of two [four].

Output Parameters

`XReal : TNvectorPtr;`	Pointer to real values of the discrete Fourier transform of the input data
`XImag : TNvectorPtr;`	Pointer to imaginary values of the discrete Fourier transform of the input data
`Error : Byte;`	0: No errors 1: *NumPoints* < 2 2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

```
ComplexFFT(NumPoints, Inverse, XReal, XImag, Error, Radix);
```

RealFFT

Description

This example performs a complex Fast Fourier Transform of real data. The *NumPoints* real data points are first mapped onto *NumPoints*/2 complex data points. A complex Fast Fourier Transform of these complex points is performed by calling *TestInput*, *MakeSinCosTable*, and *FFT*. The *NumPoints*/2 transform is then mapped onto *NumPoints* complex points. The real part of the transformation will be even, and the imaginary part of the transformation will be odd. If you are implementing this application with a radix-4 algorithm, be sure that the number of real data points (*NumPoints*) is twice the power of four.

Input Parameters

`NumPoints : Integer;`	Number of data points
`Inverse : Boolean;`	FALSE equals forward transform; TRUE equals inverse transform
`XReal : TNvectorPtr;`	Pointer to real values of the data points
`Radix : RadixType;`	*Radix2* or *Radix4*

The preceding parameters must satisfy the following conditions:

1. *NumPoints* $\geq$ 4.
2. *NumPoints* must be a power of two (twice a power of four for a radix-4 algorithm).

At least four data points are required, because this algorithm transforms the real vector to a complex vector half the size. If only two real data points were entered, the routine would have to take the transform of a single complex point.

Output Parameters

`XReal : TNvectorPtr;`	Pointer to real values of the Fourier transform of the input data
`XImag : TNvectorPtr;`	Pointer to imaginary values of the Fourier transform of the input data
`Error : Byte;`	0: No errors 1: *NumPoints* < 4 2: *NumPoints* not a power of two [twice a power of four]

Syntax of the Procedure Call

```
RealFFT(NumPoints, Inverse, XReal, XImag, Error, Radix);
```

ComplexConvolution

Description

The calculation of the convolution of two complex vectors is facilitated with a Fast Fourier Transform routine. The discrete convolution of two functions x and h is defined by

$$y_m = \sum_{n=0}^{N-1} x_n h_{m-n} \qquad m = 0, 1, \ldots N - 1$$

where subscripts are taken modulo N (circular convolution). The basic theorem that allows us to calculate these effectively using FFTs is as follows:

$$Y_m = X_m H_m \qquad m = 0, 1, \ldots N - 1$$

where capital letters indicate the transforms of the functions represented by lowercase letters. Thus the procedure for convolution works like this:

1. Transform both given data sets using FFTs.
2. Multiply the resulting transforms point by point.
3. Find the inverse transform of this product using FFTs.

Input Parameters

`NumPoints : Integer;`	Number of data points
`XReal : TNvectorPtr;`	Pointer to real values of the first set of data points
`XImag : TNvectorPtr;`	Pointer to imaginary values of the first set of data points
`HReal : TNvectorPtr;`	Pointer to real values of the second set of data points
`HImag : TNvectorPtr;`	Pointer to imaginary values of the second set of data points
`Radix : RadixType;`	*Radix2* or *Radix4*

The preceding parameters must satisfy the following conditions:

1. *NumPoints* $\geq$ 2.
2. *NumPoints* must be a power of two [four].

Output Parameters

`XReal : TNvectorPtr;`	Pointer to real values of the convolution of *XReal, XImag* and *HReal, HImag*
`XImag : TNvectorPtr;`	Pointer to imaginary values of the convolution of *XReal, XImag* and *HReal, HImag*
`Error : Byte;`	0: No errors 1: *NumPoints* < 2 2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

```
ComplexConvolution(NumPoints, XReal, XImag, HReal, HImag, Error, Radix);
```

RealConvolution

Description

The calculation of the convolution of two real vectors is facilitated with a Fast Fourier Transform routine. This procedure is exactly the same as the previous procedure for complex convolution except that only one Fourier transform need be performed. The procedure is as follows:

1. Given two real vectors *XReal* and *HReal*, combine them into a complex vector *XReal* + *iHReal*, where *i* is the square root of -1.
2. Transform this complex vector.

3. Extract the transforms of the two real functions from the transform of the complex function (using the symmetry $X_f = \overline{X}_{-f}$, where the bar stands for complex conjugation).
4. Multiply the resulting transforms point by point.
5. Find the inverse transform of this product using FFTs. *RealConvolution* is about 25 percent faster than its complex counterpart for the same set of real data.

Input Parameters

`NumPoints : Integer;`	Number of data points
`XReal : TNvectorPtr;`	Pointer to real values of the first set of data points
`HReal : TNvectorPtr;`	Pointer to real values of the second set of data points
`Radix : RadixType;`	*Radix2* or *Radix4*

The preceding parameters must satisfy the following conditions:

1. *NumPoints* $\geq$ 2.
2. *NumPoints* must be a power of two [four].

Output Parameters

`XReal : TNvectorPtr;`	Pointer to real values of the convolution of *XReal* and *HReal*
`XImag : TNvectorPtr;`	Pointer to imaginary values of the convolution of *XReal* and *HReal*
`Error : Byte;`	0: No errors 1: *NumPoints* < 2 2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

```
RealConvolution(NumPoints, XReal, XImag, HReal, Error, Radix);
```

Description

The calculation of the correlation of two complex vectors is facilitated with a Fast Fourier Transform routine. The discrete correlation of two complex functions x and h is defined by

$$y_m = \sum_{n=0}^{N-1} x_n h_{m+n} \qquad m = 0, 1, \dots N - 1$$

where subscripts are taken modulo N (circular correlation). The basic theorem that allows us to calculate these effectively using FFTs is as follows:

$$Y_m = X_m H_{N-m} \qquad m = 0, 1, \dots N - 1$$

where capital letters indicate the transforms of the functions represented by lower-case letters and $^-$ indicates the complex conjugate. (Commonly x and h are real functions, in which case the preceding formula reduces to the more familiar expression $C_m = X_m \overline{H}_m$, where the bar stands for complex conjugation. Thus the procedure for correlation works like this:

1. Transform both given data sets using FFTs.
2. Multiply each element of the transform of the first data set by the appropriate element of the transform of the second data.
3. Find the inverse transform of this product using FFTs.

If the functions x and h are different, the correlation is called *crosscorrelation;* if the functions x and h are the same, the correlation is called *autocorrelation.*

Input Parameters

`NumPoints : Integer;`	Number of data points
`Auto : Boolean;`	FALSE equals crosscorrelation; TRUE equals autocorrelation
`XReal : TNvectorPtr;`	Pointer to real values of the first set of data points
`XImag : TNvectorPtr;`	Pointer to imaginary values of the first set of data points
`HReal : TNvectorPtr;`	Pointer to real values of the second set of data points (for crosscorrelation)
`HImag : TNvectorPtr;`	Pointer to imaginary values of the second set of data points (for crosscorrelation)
`Radix : RadixType;`	*Radix2* or *Radix4*

The preceding parameters must satisfy the following conditions:

1. *NumPoints* ≥ 2.
2. *NumPoints* must be a power of two [four].

Output Parameters

`XReal : TNvectorPtr;`	Pointer to real values of the correlation of *XReal*, *XImag* and *HReal*, *HImag* (or the autocorrelation of *XReal*, *XImag* if *Auto* = TRUE)
`XImag : TNvectorPtr;`	Pointer to imaginary values of the correlation of *XReal*, *XImag* and *HReal*, *HImag* (or the autocorrelation of *XReal*, *XImag* if *Auto* = TRUE)
`Error : Byte;`	0: No errors 1: *NumPoints* < 2 2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

```
ComplexCorrelation(NumPoints, Auto, XReal, XImag, HReal, HImag, Error, Radix);
```

Comments

If you are performing an autocorrelation of the vector *XReal*, *XImag*, then set *Auto* = TRUE. In this case, the vector *HReal*, *HImag* will not contain any information but must still be passed into the procedure. Autocorrelations are faster to compute, since only one forward transformation must be made, as opposed to two for crosscorrelation.

RealCrossCorrelation

Description

The calculation of the convolution of two real vectors is facilitated with a Fast Fourier Transform routine. This procedure is exactly the same as the previous procedure for complex correlation except that only one forward Fourier transform need be performed. The procedure is as follows:

1. Given two real vectors *XReal* and *HReal*, combine them into a complex vector *XReal* + *iHReal*, where *i* is the square root of − 1.
2. Transform this complex vector.
3. Extract the transforms of the two real vectors from the transform of the complex vector (using the symmetry $X_f = \overline{X}_{-f}$, where the bar stands for complex conjugation).
4. Multiply each element of the transform of the first data set by the appropriate element of the transform of the second data.
5. Find the inverse transform of this product using FFTs.

Input Parameters

`NumPoints : Integer;`	Number of data points
`Auto : Boolean;`	FALSE equals crosscorrelation; TRUE equals autocorrelation
`XReal : TNvectorPtr;`	Pointer to real values of the first set of data points
`HReal : TNvectorPtr;`	Pointer to real values of the second set of data points (for crosscorrelation)
`Radix : RadixType;`	*Radix2* or *Radix4*

The preceding parameters must satisfy the following conditions:

1. *NumPoints* $\geq$ 2.
2. *NumPoints* must be a power of two [four].

Output Parameters

`XReal : TNvectorPtr;`	Pointer to real values of the correlation of *XReal* and *HReal* (or the autocorrelation of *XReal* if *Auto* = TRUE)
`XImag : TNvectorPtr;`	Pointer to imaginary values of the correlation of *XReal* and *HReal* (or the autocorrelation of *XReal* if *Auto* = TRUE)
`Error : Byte;`	0: No errors 1: *NumPoints* < 2 2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

```
RealCorrelation(NumPoints, Auto, XReal, XImag, HReal, Error, Radix);
```

Comments

If you are performing an autocorrelation of the vector *XReal*, then set *Auto* equal to TRUE. In this case, the vector *HReal* will not contain any information but must still be passed into the procedure. Autocorrelations are faster to compute, since only one forward transformation must be made, as opposed to two for crosscorrelation.

Sample Program

The sample program FFTProgs.pas provides I/O functions that demonstrate any of the application programs.

Input File

Data may be entered from a text file. The real and imaginary parts of a complex number should be separated by a space and followed by a carriage return. Real numbers should each be followed by a carriage return.

The procedures *ComplexFFT*, *ComplexConvolution*, and *ComplexCrossCorrelation* expect data to be in complex form. A data file containing a four-point complex square wave could look like this:

```
0 0
1 1
1 1
0 0
```

The procedures *RealFFT*, *RealConvolution*, and *RealCrossCorrelation* expect data to be in real form. A data file containing a four-point real square wave could look like this:

```
0
1
1
0
```

Example

Problem. Perform a Fourier transform and an autocorrelation of a 32-point square wave. Also, convolve and crosscorrelate this square wave with a saw-tooth wave.

1. The input data file Sample10A.dat is as follows (note that this is in real format):

```
0
0
0
0
0
0
0
0
0
0
0
1
1
1
1
1
1
1
1
1
1
1
0
0
0
0
0
0
0
```

```
0
0
0
```

2. Run FFTProgs.pas:

```
1. Real Fast Fourier Transform
2. Real Convolution
3. Real Autocorrelation
4. Real Crosscorrelation
5. Complex Fast Fourier Transform
6. Complex Convolution
7. Complex Autocorrelation
8. Complex Crosscorrelation

Select a number (1-8): 1

********* Real Fast Fourier Transform *********

(F)orward or (I)nverse transform? F
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample10A.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Results of real Fourier transform:
 1.94454364826301e+0     0.00000000000000e+0
-1.59057003804788e+0    -3.56682381055970e-17
 7.53417436515731e-1     4.55292313916419e-17
 5.96901852132470e-2    -2.69289447846945e-16
-4.26776695296637e-1    -2.87492515628133e-20
 2.89883706652938e-1    -3.48529572466949e-16
 6.20757203331860e-2     1.09874847931145e-16
-2.66655959906343e-1     2.77190700484791e-18
 1.76776695296637e-1     0.00000000000000e+0
 6.63840517512571e-2     9.35764180659936e-17
-2.08522329739913e-1     1.09798183260311e-16
 1.27160952826887e-1    -1.28863728588383e-16
 7.32233047033631e-2     1.05413922396982e-19
-1.83841625879619e-1    -4.96595405328328e-17
 1.00135954077543e-1     4.53375697145565e-17
 8.37351650164211e-2     5.53423092584155e-17
-1.76776695296637e-1     0.00000000000000e+0
 8.37351650164211e-2    -5.53423092584155e-17
 1.00135954077543e-1    -4.53375697145565e-17
-1.83841625879619e-1     4.96595405328328e-17
 7.32233047033631e-2    -1.05413922396982e-19
 1.27160952826887e-1     1.28863728588383e-16
-2.08522329739913e-1    -1.09798183260311e-16
```

```
 6.63840517512571e-2       -9.35764180659936e-17
 1.76776695296637e-1       -0.00000000000000e+0
-2.66655959906343e-1       -2.77190700484791e-18
 6.20757203331860e-2       -1.09874847931145e-16
 2.89883706652938e-1        3.48529572466949e-16
-4.26776695296637e-1        2.87492515628133e-20
 5.96901852132470e-2        2.69289447846945e-16
 7.53417436515731e-1       -4.55292313916419e-17
-1.59057003804788e+0        3.56682381055970e-17
```

Note that the transform of the even real-square wave is an even real function. If you take the inverse transform of this data, you should get back the original square wave.

3. Run FFTProgs.pas:

```
1. Real Fast Fourier Transform
2. Real Convolution
3. Real Autocorrelation
4. Real Crosscorrelation
5. Complex Fast Fourier Transform
6. Complex Convolution
7. Complex Autocorrelation
8. Complex Crosscorrelation

Select a number (1-8): 5

********* Complex Fast Fourier Transform *********

(F)orward or (I)nverse transform? I
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample10B.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Results of complex Fourier transform:
 1.83845713893878e-15      -0.00000000000000e+0
 1.68459114457461e-15      -6.70815869798976e-20
 2.11375997190428e-15      -1.05413922396982e-19
 1.89507399834982e-15       3.44909004811084e-17
 1.18630911648793e-15      -0.00000000000000e+0
 1.10496790073287e-15       3.42849429972988e-17
 9.86928259751718e-16       1.68143873326862e-16
 9.23818927547367e-16       2.37407332761055e-16
 1.00487796352727e-15      -0.00000000000000e+0
 3.31468256684932e-16       2.37751564484486e-16
-7.03172956429440e-17       1.68148262525511e-16
 1.00000000000000e+0        2.03763730303779e-16
 1.00000000000000e+0       -0.00000000000000e+0
 1.00000000000000e+0        2.03969687787589e-16
 1.00000000000000e+0      7.85812876050229e-19
 1.00000000000000e+0      1.60995808751754e-18
 9.99999999999999e-1     -0.00000000000000e+0
 1.00000000000000e+0      1.62912425522608e-19
 1.00000000000000e+0      1.05413922396982e-19
 9.99999999999999e-1     -2.02632926408975e-16
 1.00000000000000e+0      0.00000000000000e+0
 1.00000000000000e+0     -2.02417228203197e-16
-1.44345194948787e-16    -1.68143873326862e-16
-6.60945902546872e-17    -2.38193145637105e-16
 9.04264001920616e-16    -0.00000000000000e+0
 1.13846180955400e-15    -2.37847395323029e-16
 3.49146703466816e-16    -1.68148262525511e-16
 1.49507607827254e-15    -3.56217043759124e-17
 9.82879412429460e-16    -0.00000000000000e+0
 1.52785022505415e-15    -3.58374025816908e-17
 1.62624933006980e-15    -7.85812876050229e-19
 1.58864530902564e-15    -8.24145211467314e-19
```

You get back the original square wave, accurate to 15 significant figures.

The autocorrelation of a square wave is simply a triangle. Let's take the autocorrelation of the square wave.

4. Run FFTProgs.pas:

```
1. Real Fast Fourier Transform
2. Real Convolution
3. Real Autocorrelation
4. Real Crosscorrelation
5. Complex Fast Fourier Transform
6. Complex Convolution
7. Complex Autocorrelation
8. Complex Crosscorrelation

Select a number (1-8): 3

********* Real Autocorrelation *********
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample10A.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Results of real autocorrelation:
 1.94454364826301e+0      -1.86068547103024e-18
 1.76776695296637e+0      -7.31524706015784e-17
 1.59099025766973e+0      -1.20291519030913e-16
 1.41421356237310e+0      -2.28393637498843e-16
 1.23743686707646e+0      -1.08420217248551e-18
 1.06066017177982e+0      -3.11917400601706e-16
 8.83883476483185e-1      -2.67794627815503e-16
 7.07106781186548e-1      -3.24279578773716e-16
 5.30330085889911e-1       1.54428189954649e-19
 3.53553390593273e-1      -2.64227183800926e-16
 1.76776695296636e-1      -2.67190893532684e-16
-7.13134768099437e-16     -1.92763731728663e-16
-8.91610121801382e-16      1.08420217248551e-18
-6.53642983532122e-16     -1.08779980600795e-16
-6.24203749931802e-16     -1.18700727111104e-16
-7.13441426782774e-16     -1.28557069905047e-17
-4.75627617855183e-16      1.55182909112094e-18
-7.13364762111940e-16      1.34977736087408e-17
-6.24050420590133e-16      1.20128888705040e-16
-6.54026306886293e-16      1.09467566867339e-16
-8.91610121801382e-16      1.08420217248551e-18
-7.13748085466111e-16      1.92996121512129e-16
 1.76776695296636e-1       2.67957258141376e-16
 3.53553390593273e-1       2.64586549445461e-16
 5.30330085889911e-1       1.54428189954649e-19
 7.07106781186548e-1       3.23881880793764e-16
 8.83883476483185e-1       2.67353523858557e-16
 1.06066017177982e+0       3.11689802360167e-16
 1.23743686707646e+0      -1.08420217248551e-18
 1.41421356237310e+0       2.27701259690372e-16
 1.59099025766973e+0       1.18538096785231e-16
 1.76776695296637e+0       7.25487363187593e-17
```

Keeping in mind that this is a periodic function (see "Data Sampling"), you can see that this is a triangle wave.

Let's now convolve the square wave with a saw-tooth wave. The input file for the saw-tooth wave (Sample10C.dat) is as follows:

```
0
0
0
0
0
0
0
0
0
0
0
1
2
3
```

```
 4
 5
 6
 7
 8
 9
10
 1
 1
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
```

5. Run FFTProgs.pas:

```
1. Real Fast Fourier Transform
2. Real Convolution
3. Real Autocorrelation
4. Real Crosscorrelation
5. Complex Fast Fourier Transform
6. Complex Convolution
7. Complex Autocorrelation
8. Complex Crosscorrelation

Select a number (1-8): 2

********* Real Convolution *********
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
The first function:

File name? Sample10A.dat

The second function:
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample10C.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Results of real convolution:
 1.16672618895780e+1     -0.00000000000000e+0
 1.14904851942814e+1      5.39795947343382e-16
 1.11369318036881e+1     -1.17756934401283e-16
 1.06066017177982e+1     -6.23437103223460e-16
 9.89949493661167e+0     -0.00000000000000e+0
 9.01561146012848e+0     -1.63431828667510e-15
 7.95495128834866e+0     -1.33473191922288e-15
 6.71751442127220e+0     -2.27837818635295e-15
 5.30330085889911e+0     -0.00000000000000e+0
 3.71231060122937e+0     -2.05062661547234e-15
 1.94454364826300e+0     -1.87280291147249e-15
-4.48963645339059e-15    -1.68784939308506e-15
-4.65630544778407e-15    -0.00000000000000e+0
-3.65659814010651e-15    -1.09893055790468e-15
-3.47045631932115e-15    -1.31403245809765e-15
-3.85439299085867e-15    -2.51249292491279e-16
-2.87891171916470e-15    -0.00000000000000e+0
-4.57044101644980e-15    -9.33009044051833e-17
-3.77067517030775e-15     1.17756934401283e-16
-4.08070709916113e-15     2.09754539402286e-16
-5.84767443254705e-15    -0.00000000000000e+0
-3.65997138562321e-15     6.25142892149520e-16
 1.76776695296633e-1      1.33473191922288e-15
 5.30330085889907e-1      1.11725341423405e-15
 1.06066017177982e+0     -0.00000000000000e+0
 1.76776695296637e+0      1.60413157253415e-15
 2.65165042944955e+0      1.87280291147249e-15
 3.71231060122937e+0      2.10153195690623e-15
 4.94974746830584e+0     -0.00000000000000e+0
 6.36396103067893e+0      2.10810595243026e-15
 7.95495128834866e+0      1.31403245809765e-15
 9.72271824131503e+0      1.41237406461018e-15
```

Now let's crosscorrelate the square wave with the saw-tooth wave.

6. Run FFTProgs.pas:

```
1. Real Fast Fourier Transform
2. Real Convolution
3. Real Autocorrelation
4. Real Crosscorrelation
5. Complex Fast Fourier Transform
6. Complex Convolution
7. Complex Autocorrelation
8. Complex Crosscorrelation

Select a number (1-8): 4

********* Real Crosscorrelation *********
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
The first function:

File name? Sample10A.dat

The second function:
```

A dialog box appears asking you whether you will input data from the **Keyboard** or from a **File**. Select **File** and click **OK**. Then select the following file from the standard dialog box:

```
File name? Sample10C.dat
```

Now another dialog box appears asking you whether you would like the output sent to the **Screen**, directly to the **Printer**, or into a **File**. Make your selection and click **OK**.

```
Results of real crosscorrelation:
 1.16672618895780e+1      -0.00000000000000e+0
 9.72271824131503e+0      -1.39794194032565e-15
 7.95495128834866e+0      -1.30705597305174e-15
 6.36396103067893e+0      -2.10114863355206e-15
 4.94974746830583e+0      -0.00000000000000e+0
 3.71231060122937e+0      -2.09953867546454e-15
 2.65165042944955e+0      -1.87232375727978e-15
 1.76776695296637e+0      -1.61001558602067e-15
 1.06066017177982e+0      -0.00000000000000e+0
 5.30330085889907e-1      -1.12643400856644e-15
 1.76776695296632e-1      -1.34178506893962e-15
-4.54836159124956e-15     -6.36393432594434e-16
-6.31870217015218e-15     -0.00000000000000e+0
-3.84059335010852e-15     -2.20870916673240e-16
-2.91417746774842e-15     -1.27416682926388e-16
-3.80256767337477e-15      8.09195600654651e-17
-2.82248652143076e-15     -0.00000000000000e+0
-3.79950108654140e-15      2.39002111325521e-16
-3.67714427189007e-15      1.30705597305174e-15
-3.77895495475784e-15      1.08963496656604e-15
-4.41588504004812e-15     -0.00000000000000e+0
-4.31192774639698e-15      1.67956960863497e-15
 1.94454364826300e+0       1.87232375727978e-15
 3.71231060122937e+0       2.05122076667131e-15
 5.30330085889911e+0      -0.00000000000000e+0
 6.71751442127221e+0       2.28537383756657e-15
 7.95495128834866e+0       1.34178506893962e-15
 9.01561146012849e+0       1.64790709958046e-15
 9.89949493661167e+0      -0.00000000000000e+0
 1.06066017177982e+1       6.40839983502816e-16
 1.11369318036881e+1       1.27416682926388e-16
 1.14904851942814e+1      -5.22124740716106e-16
```

C H A P T E R 11

Graphics Programs

There are some programs that graphically demonstrate the usefulness of the least-squares routines in Chapter 9 and the Fourier transforms in Chapter 10. Each program reads a data set from an input file, and displays the results. You will see curves being fitted to data using the least-squares routines and also see a signal being transformed into its Fourier spectrum.

The programs LSQDemo and FFTDemo graphically illustrate the power and utility of the Turbo Pascal Numerical Methods Toolbox.

Function of the Least-Squares Graphics Demonstration Program

The program LSQDemo demonstrates the least-squares capabilities of the Toolbox. A default input file Sample11A.dat contains the x and y values (in ASCII form) separated by carriage returns. Running LSQDemo will provide five different least-squares fits to the input data.

The different fits are based on the function forms: logarithm, exponential, polynomial, power law, and finite Fourier series. The fits are displayed graphically on the screen and can be printed on an ImageWriter or LaserWriter printer.

The first plot shows the input data from Sample11A.dat along with three curves. The three curves are the graphs of the power function

$$Y = aX^b$$

the exponential function

$$Y = a \exp(bX)$$

and the logarithm function

$$Y = a\, ln(bX)$$

The header to the plot tells which curve corresponds to which function. The next plot shows the same input data plotted with a five-term Fourier series:

$$Y = a + b * \cos(x) + c * \sin(x) + d * \cos(2X) + e * \sin(2X)$$

and a five-term polynomial (that is, a polynomial of degree four). The coefficients are found using the routines from Chapter 9, and they give the least-square error among all functions of that form. (In some cases, the problem is transformed into a linear problem, and the error is actually the least for the transformed problem but possibly not exactly the least for the original problem.) Again, the header to the plot tells which curve corresponds to which function.

Finally, a bar chart shows the error for each function. The data is not at all periodic, so the Fourier series model is the worst. The five-degree polynomial gives the best fit, but it is not much better than the fit obtained by using power, exponential, or logarithm functions.

The LSQDemo program offers three pulldown menus—File, Edit and Window. The Edit menu does not offer any executable commands while the File and Window menus offer three and six selections respectively.

The File menu offers:

Print Screen	Prints everything displayed on the screen on an ImageWriter™, ImageWriter™ II, or LaserWriter™
Print Window	Prints the currently selected window on an ImageWriter, ImageWriter II, or LaserWriter
Quit	Terminates program execution

The Window menu gives you control of the various windows displayed on the screen and offers the following window-related commands:

Zoom Windows	Zooms all windows to the largest possible size
Stack Windows	Layers all windows on the screen
Tile Windows	Displays all windows in a row, from the top to the bottom of the screen
Power, Exp, Log	Selects and brings forward the window displaying the power, exponential, and logarithm least-squares fits
Fourier, Polynomial	Selects and brings forward the window displaying the fourier and polynomial least-squares fits
Sum of Squares	Selects and brings forward the window displaying the sum of the squares of the residuals for the five least-squares fits

Function of the Fourier Transform Graphics Demonstration Program

The program FFTDemo demonstrates the Fourier capabilities of the Toolbox.

A default input file Sample11B.dat contains 1024 real values (in ASCII form) separated by carriage returns. These values represent sample points from a two-second signal sampled at a rate of 512 points per second. The program will display four FFT transforms at the following sampling rates: 8 per second (16 points), 32 per second (64 points), 128 per second (256 points), and 512 per second (1,024 points). For the last two samplings, the default data yields the same transforms, demonstrating that a sample rate higher than twice the highest frequency adds no new information (the *Nyquist limit*). The transforms are shown on a scale of −64 to +63 cycles per second.

In addition to the real and imaginary transforms, the program displays the inverse transform over the original data, illustrating the degree to which information is lost at different sampling rates. The header tells which curve is the original data and which is the inverse transform.

A default output data file can easily be arranged by changing the constant *WriteToFile* in FFTDemo.pas and recompiling it.

The FFTDemo program offers five pulldown menus—File, Edit, Sample, Window, and Graph. File and Window additionally provide three and six options respectively.

The File menu offers:

Print Screen	Prints everything displayed on the screen on an ImageWriter, ImageWriter II, or LaserWriter
Print Window	Prints the currently selected window on an ImageWriter, ImageWriter II, or LaserWriter
Quit	Terminates the program

The Edit menu does not offer any executable commands.

The Sample menu allows you to select one of the four sampling rates (mentioned earlier), and indicates the currently selected sample rate with a check mark.

The Window menu gives you control of the various windows displayed on the screen and offers the following window-related commands:

Zoom Windows	Zooms all windows to the largest possible size
Stack Windows	Layers all windows on the screen
Tile Windows	Displays all windows in a row, from the top to the bottom of the screen
Real Transform	Selects and brings forward the window displaying the real transformation
Imaginary Transform	Selects and brings forward the window displaying the imaginary transformation
Inverse Transform	Selects and brings forward the window displaying the inverse transformation

The Graph menu offers only one selection, Display new graph, which lets you display a new set of graphs with the currently selected sampling rate.

Rebuilding the Demonstration Programs

This procedure assumes that Turbo Pascal is on your hard disk or in a floppy disk drive.

How to recompile the Demos:

1. Copy Disk 1 to a folder on your hard disk or onto another disk. (You don't need to copy the Read Me program or the file Read.file.)
2. Double click on the TurboGraph.unit file. (This should bring up Turbo Pascal.)
3. Compile this Unit to disk. (Type Command-K to Select "Compile To Disk" in Turbo Pascal.)
4. Open either FFTDemo.pas or LSQDemo.pas.
5. Select Command-R to run the Demos in memory.

References

Abramowitz, Milton, and Irene A. Stegun, eds. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Washington, D.C.: National Bureau of Standards Applied Mathematics Series, 55, 1972.

Atkinson, L.V., and P.J. Harley. *An Introduction to Numerical Methods with Pascal.* Reading: Addison-Wesley Publishing Co., 1983. This is an excellent text for learning numerical methods, with an emphasis on the implementation of various numerical algorithms.

Brigham, E. Oran. *The Fast Fourier Transform.* Englewood Cliffs: Prentice-Hall, Inc., 1974. A very complete, easy-to-read text on the use and implementation of the fast Fourier transform algorithm.

The next three texts are excellent for learning numerical analysis, emphasizing the mathematical theory underlying the algorithms in this toolbox.

Burden, Richard L., and J. Douglas Faires. *Numerical Analysis,* 3rd ed. Boston: Prindle, Weber & Schmidt, 1985.

Cheney, Ward, and David Kincaid. *Numerical Mathematics and Computing,* 2nd ed. Monterey: Brooks/Cole Publishing Co., 1985.

Dahlquist, Germund, and Ake Bjorck. *Numerical Methods,* trans. Ned Anderson. Englewood Cliffs: Prentice-Hall, Inc., 1974.

Gerald, Curtis F., and Patrick O. Wheatley. *Applied Numerical Analysis,* 3rd ed. Reading: Addison-Wesley Publishing Co., 1984. This is another excellent source for learning numerical analysis.

Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* New York: Cambridge University Press, 1986. This book discusses many of the subtle problems encountered when implementing numerical methods, and has program listings in Turbo Pascal.

Ralston, Anthony, and Philip Rabinowitz. *A First Course in Numerical Analysis.* New York: McGraw-Hill Book Co., 1978. A well-written mathematics text that is a step more sophisticated than the preceding ones.

Index

TURBO PASCAL® MACINTOSH™

The ultimate Pascal development environment

Borland's new Turbo Pascal for the Mac is so incredibly fast that it can compile 1,420 lines of source code in the 7.1 seconds it took you to read this!

And reading the rest of this takes about *5 minutes*, which is plenty of time for Turbo Pascal for the Mac to compile at least *60,000 more lines* of source code!

Turbo Pascal for the Mac does both Windows and "Units"
The *separate* compilation of routines offered by Turbo Pascal for the Mac creates modules called "Units," which can be linked to any Turbo Pascal program. This "modular pathway" gives you "pieces" which can then be integrated into larger programs. You get a more efficient use of memory and a reduction in the time it takes to develop large programs.

Turbo Pascal for the Mac is so compatible with Lisa® that they should be living together
Routines from Macintosh Programmer's Workshop Pascal and Inside Macintosh can be compiled and run with only the subtlest changes. Turbo Pascal for the Mac is also compatible with the Hierarchical File System of the Macintosh.

The 27-second Guide to Turbo Pascal for the Mac

- Compilation speed of more than 12,000 lines per minute
- "Unit" structure lets you create programs in modular form
- Multiple editing windows—up to 8 at once
- Compilation options include compiling to disk or memory, or compile and run
- No need to switch between programs to compile or run a program
- Streamlined development and debugging
- Compatibility with Macintosh Programmer's Workshop Pascal (with minimal changes)
- Compatibility with Hierarchical File System of your Mac
- Ability to define default volume and folder names used in compiler directives
- Search and change features in the editor speed up and simplify alteration of routines
- Ability to use all available Macintosh memory without limit
- "Units" included to call all the routines provided by Macintosh Toolbox

Suggested Retail Price: $99.95* (not copy protected)

3 MacWinners from Borland!
First there was SideKick for the Mac, then Reflex for the Mac, and now Turbo Pascal for the Mac!

Minimum system configuration: Macintosh 512K or Macintosh Plus with one disk drive.

BORLAND
INTERNATIONAL

TURBO PASCAL® TUTOR

From the folks who created Turbo Pascal. Borland's new Turbo Pascal Tutor is everything you need to start programming in Turbo Pascal on the Macintosh!™ It takes you from the bare basics to advanced programming in a simple, easy-to-understand fashion.

No gimmicks. It's all here.

The manual, the Tutor application, and 30 sample programs provide a step-by-step tutorial in three phases: programming in Pascal, programming on the Macintosh, and programming in Turbo Pascal on the Macintosh. Here's how the manual is set up:

Turbo Pascal for the Absolute Novice
delivers the basics—a concise history of Pascal, key terminology, your first program.

A Programmer's Guide to Turbo Pascal
covers Pascal specifics—program structure, procedures and functions, arrays, strings, and so on. We've also included Turbo Typist, a textbook sample program.

Advanced Programming
takes you a step higher into stacks, queues, binary trees, linked structures, writing large programs, and more.

Using the Power of the Macintosh
discusses the revolutionary hardware and software features of this machine. It introduces the 600-plus utility routines in the Apple Toolbox.

Programming the Macintosh in Turbo Pascal
shows you how to create true·Macintosh programs that use graphics, pull-down menus, dialog boxes, and so on. Finally, MacTypist, a complete stand-alone application featuring animated graphics, builds on Turbo Typist and demonstrates what you can do with all the knowledge you've just acquired.

The disk contains the source code for all the sample programs, including Turbo Typist, MacTypist, and Turbo Tutor. The Tutor's split screen lets you run a procedure and view its source code simultaneously. After running it, you can take a test on the procedure. If you're stuck for an answer, a Hint option steers you in the right direction.

Macintosh topics included are

- ☑ memory management
- ☑ resources and resource files
- ☑ QuickDraw
- ☑ events
- ☑ windows
- ☑ controls
- ☑ menus
- ☑ desk accessory support
- ☑ dialogs
- ☑ File Manager
- ☑ debugging

Suggested Retail Price: $69.95

Minimum system requirements: Any Macintosh with at least 512K of RAM. Requires Turbo Pascal.

BORLAND
INTERNATIONAL

SIDEKICK®: THE DESKTOP ORGANIZER Release 2.0

Macintosh™

The most complete and comprehensive collection of desk accessories available for your Macintosh!

Thousands of users already know that SideKick is the best collection of desk accessories available for the Macintosh. With our new Release 2.0, the best just got better.

We've just added two powerful high-performance tools to SideKick—Outlook™: The Outliner and MacPlan™: The Spreadsheet. They work in perfect harmony with each other and *while* you run other programs!

Outlook: The Outliner

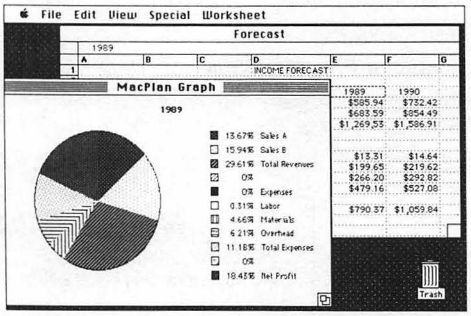
- It's the desk accessory with more power than a stand-alone outliner
- A great desktop publishing tool, Outlook lets you incorporate both text and graphics into your outlines
- Works hand-in-hand with MacPlan
- Allows you to work on several outlines at the same time

MacPlan: The Spreadsheet

- Integrates spreadsheets and graphs
- Does both formulas and straight numbers
- Graph types include bar charts, stacked bar charts, pie charts and line graphs
- Includes 12 example templates free!
- Pastes graphics and data right into Outlook creating professional memos and reports, complete with headers and footers.

SideKick: The Desktop Organizer, Release 2.0 now includes

- ☑ Outlook: The Outliner
- ☑ MacPlan: The Spreadsheet
- ☑ Mini word processor
- ☑ Calendar
- ☑ PhoneLog
- ☑ Analog clock
- ☑ Alarm system
- ☑ Calculator
- ☑ Report generator
- ☑ Telecommunications (new version now supports XModem file transfer protocol)



	A	B	C	D	E	F	G
1				INCOME FORECAST			
					1989	1990	
					$585.94	$732.42	
					$683.59	$854.49	
					$1,269.53	$1,586.91	
					$13.31	$14.64	
					$199.65	$219.62	
					$266.20	$292.82	
					$479.16	$527.08	
					$790.37	$1,059.84	

MacPlan does both spreadsheets and business graphs. Paste them into your Outlook files and generate professional reports.

Suggested Retail Price: $99.95 (not copy protected)

Minimum system configurations: Macintosh 512K or Macintosh Plus with one disk drive. One 800K or two 400K drives are recommended. With one 400K drive, a limited number of desk accessories will be installable per disk.

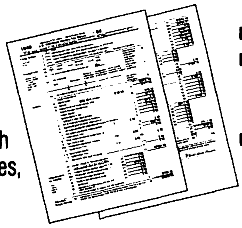
BORLAND INTERNATIONAL

BOR 0069D

REFLEX®: THE DATABASE MANAGER

The easy-to-use relational database that thinks like a spreadsheet. Reflex for the Mac lets you crunch numbers by entering formulas and link databases by drawing on-screen lines.

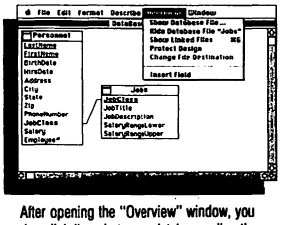
5 free ready-to-use templates are included on the examples disk:



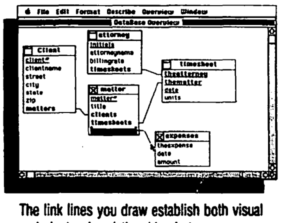
- A sample 1040 tax application with Schedule A, Schedule B, and Schedule D, each contained in a separate report document.
- A portfolio analysis application with linked databases of stock purchases, sales, and dividend payments.
- A checkbook application.
- A client billing application set up for a law office, but easily customized by any professional who bills time.
- A parts explosion application that breaks down an object into its component parts for cost analysis.

Reflex for the Mac accomplishes all of these tasks without programming—using spreadsheet-like formulas. Some other Reflex for the Mac features are:

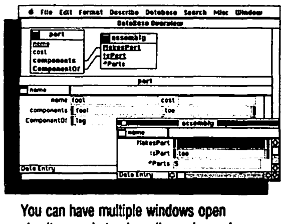
- Visual database design.
- "What you see is what you get" report and form layout with pictures.
- Automatic restructuring of database files when data types are changed, or fields are added and deleted.
- Display formats which include General, Decimal, Scientific, Dollars, Percent.
- Data types which include variable length text, number, integer, automatically incremented sequence number, date, time, and logical.
- Up to 255 fields per record.
- Up to 16 files simultaneously open.
- Up to 16 Mac fonts and styles are selectable for individual fields and labels.



After opening the "Overview" window, you draw link lines between databases directly onto your Macintosh screen.



The link lines you draw establish both visual and electronic relationships between your databases.



You can have multiple windows open simultaneously to view all members of a linked set—which are interactive and truly relational.

Critic's Choice

". . . a powerful relational database . . . uses a visual approach to information management." ***InfoWorld***

". . . gives you a lot of freedom in report design; you can even import graphics." ***A+ Magazine***

". . . bridges the gap between the pretty programs and the power programs." **Stewart Alsop, *PC Letter***

BORLAND
INTERNATIONAL

Suggested Retail Price: $99.95*
(not copy protected)

***Minimum system configuration:** Macintosh 512K or Macintosh Plus with one disk drive. Second external drive recommended.*

BOR0149A

Borland Software ORDER TODAY

BORLAND
INTERNATIONAL

4585 Scotts Valley Drive Scotts Valley, California 95066

To Order By Credit Card, Call (800) 255-8008

In California call (800) 742-1133

In Canada call (800) 237-1136

TURBO PASCAL MACINTOSH BORLAND
4
SIDEKICK: THE DESKTOP ORGANIZER MACINTOSH BORLAND
3
REFLEX FOR THE MAC
1
EUREKA: THE SOLVER MACINTOSH BORLAND
2

For The Apple® Macintosh™

1 Reflex®: The Database Manager

Combines the analytic capability of a relational database with the number-crunching power of a spreadsheet to let you organize, analyze and report information faster than ever before. Creating database designs, forms, and reports is as easy as drawing them on your screen. Comes with sample applications which can be customized to your needs. Minimum memory: 512K.

2 Eureka: The Solver™*

Anyone who routinely works with equations needs Eureka: The Solver. Using a mouse, Macintosh pull-down menus and online help screens, you can solve complex equations while interacting with your computer in an almost conversational way. Eureka also lets you plot graphs of the functions and generate reports showing the equation file, the solutions and the graphs. Support for the 68881 math co-processor chip is also provided. Minimum memory: 512K.

*Available Third Quarter 1987.

3 SideKick®: The Desktop Organizer, Release 2.0

A complete and comprehensive collection of desk accessories. Includes Outlook,™ a powerful outline processor, and MacPlan,™ a 20x50 cell spreadsheet that supports 30 mathematical functions and standard business graphics. Also includes notepad, calculator, modem communications package, phone directory, autodialer, appointment scheduler, alarm system and quicksheet reminder templates. Minimum memory: 512K.

4 Turbo Pascal®

The fastest, most efficient and easy-to-use Pascal compiler! Compiles and runs over 12,000 lines of source code per minute. Multiple editing windows let you work with up to 8 programs at a time. Compatible with your Macintosh's Hierarchical File System, Macintosh Programmer's Workshop Pascal and Inside Macintosh. Minimum memory: 512K.

BORLAND
INTERNATIONAL

4584 SCOTTS VALLEY DRIVE SCOTTS VALLEY, CA 95066 PHONE (408) 438-8400 TELEX: 172373

Scientific &

11 Turbo Pascal Numerical Methods Toolbox™

Implements the latest high-level mathematical methods to solve the most common problems. An essential programming tool for mathematicians, engineers, statisticians, or physicists. Supports the 8087 chip and comes complete with source code. Minimum memory: 256K. Requires Turbo Pascal 2.0 or later.

Business P

13 Sprint®: The Professional Word Processor*

The most powerful, easy-to-use word processor ever written. Can be used "as is" or told to function like WordPerfect,™ WordStar® or Microsoft® Word. Includes pop-up menus, incremental saving, multiple windows and files, and Autospell (with 100,000-word dictionary and 300,000-word thesaurus). Drives practically every printer. Minimum memory: 256K.

*Available Second Half 1987.

14 Reflex®: The Database Manager

No matter what business you're in, Reflex is the database management system for you. With its Form, List, Graph, Crosstab and Report views that give you instant graphic analyses of your data, Reflex shows you patterns and relationships otherwise hidden in data and numbers. Minimum memory: 384K.

15 Reflex: The Workshop™

Taps Reflex's powerful analytical capabilities and makes them work for your business. Comes with 22 models and five samples on disk that you can adapt to your needs. It can also generate form letters, help you through common analysis problems, and explain advanced reporting and graphing techniques. Minimum memory: 384K.

16 SuperKey®: The Productivity Booster

With SuperKey you can turn a thousand keystrokes into the one keystroke of your choice! You can encrypt your confidential files in seconds. And SuperKey is RAM-resident, so you can encrypt files or create macros while you're running another program. Minimum memory: 128K.

Engineering

12 **Eureka: The Solver™**

Any solvable problem that can be expressed as a linear or non-linear equation can be solved using Eureka. Its pull-down menus and context-sensitive help screens make it easy to use and learn. Eureka can also plot graphs of functions and print them out. Minimum memory: 384K.

'roductivity

17 **SideKick®: The Desktop Organizer**

The #1 best-seller for the IBM PC and true compatibles, SideKick is a powerful, RAM-resident desktop management program. Comes with notepad, calendar, calculator, appointment scheduler, telephone directory, and autodialer. Can be called up at the touch of a key, even while you run other programs. Minimum memory: 128K.

18 **Traveling SideKick®**

Your SideKick's sidekick and the organizer for the Computer Age! It's both a notebook that travels with you and a software program. The software lets you organize, format and print your address book, phone list, mailing labels and calendar engagements in daily, weekly, monthly or yearly formats from its own files or your SideKick files. So you can stay up-to-date, at home and on the road. Minimum memory: 256K.

19 **Turbo Lightning®: The Spell-Checker & Thesaurus**

Gives you a RAM-resident spell-checker and thesaurus. Beeps every time you make a mistake and lets you correct a misspelled word instantly. Synonyms are available—at the touch of a key! Minimum memory: 256K.

20 **Lightning Word Wizard™**

With the help of Turbo Lightning, you can incorporate Lightning Word Wizard's procedures and functions into your own word programs. Includes source code for Turbo Lightning. Comes with seven games and solvers to give you ideas on how to implement the routines in your own applications—or play the games just for fun! Minimum memory: 256K.

But we haven't stopped there. We are constantly developing new products. Improving our existing products. And exploring new and better ways to make your computer's potential more accessible, your software more friendly, more affordable.

Our commitment extends over several categories of software development. From programming languages and Artificial Intelligence to business productivity and scientific and engineering products. And our products in every category are faster, more powerful and technically more advanced.

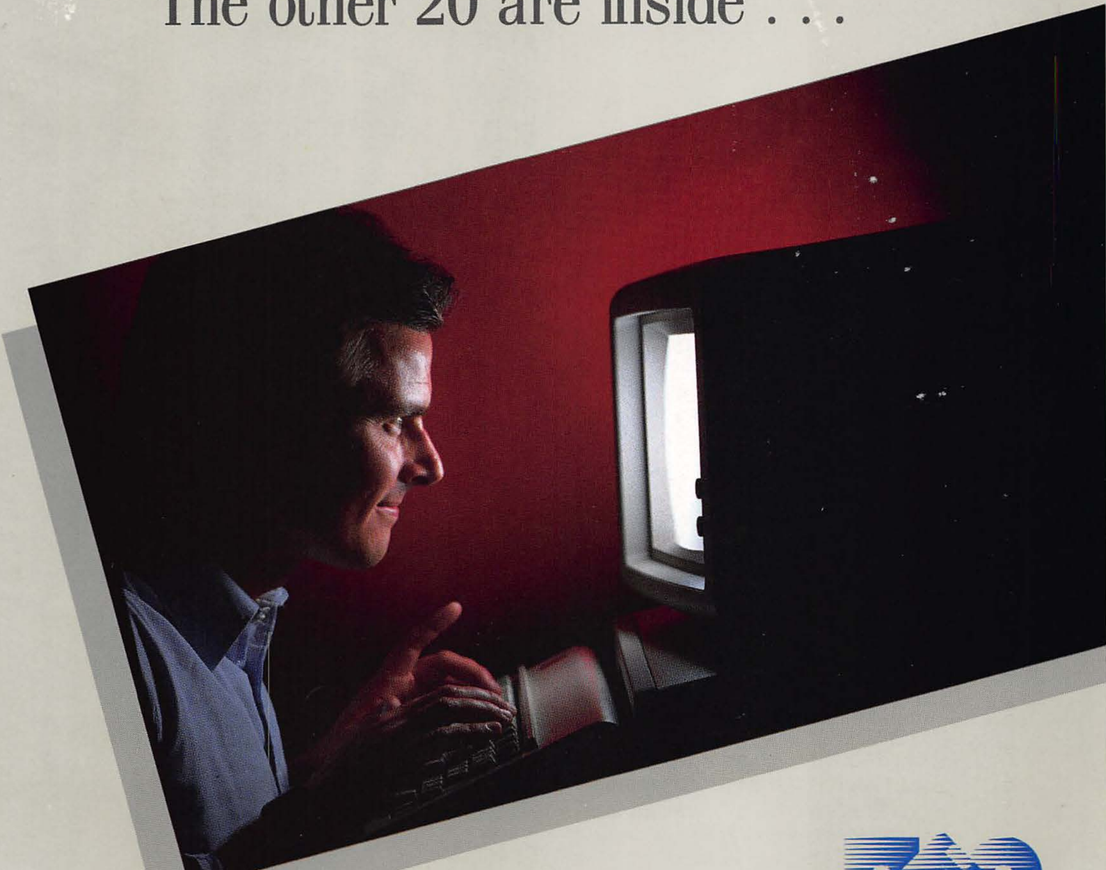
So whether you're a PC user or a Macintosh user; whether you're an expert programmer or a beginner; a business user or someone who just likes tapping at a keyboard, you can be sure that Borland has the software to match your needs. At a price to match your pocket. And a performance level that's unmatched.

Take a look inside. Make your choice. Then, if you have any further questions, call us at (408) 438-8400.

For the dealer nearest you or to order by phone call
(800) 255-8008
in CA (800) 742-1133 in Canada (800) 237-1136

Turbo Pascal,
Turbo C, Sprint,
and Reflex
are 4 of our
famous products.

The other 20 are inside . . .



opment Languages

6 Turbo Pascal Tutor® 2.0

This interactive tutorial for Turbo Pascal takes you from "What's a computer?" through complex data structures, assembly languages, trees and tips on writing long Turbo Pascal programs. Includes a 400-page, quick-study tutorial and 10,000 lines of fully commented source code. Minimum memory: 192K. Requires Turbo Pascal 3.0 (CP/M-80 version available.)

7 Turbo Pascal Graphix Toolbox®

A library of graphics routines for Turbo Pascal programs. Lets even beginning programmers create high-resolution graphics on the IBM® PC, true compatibles, and the Zenith Z-100.® Gives you a set of programming tools for complex business graphics, easy windowing and storing screen images to disk and to memory. Minimum memory: 192K. Requires Turbo Pascal 3.0.

8 Turbo Pascal Database Toolbox®

A perfect companion to Turbo Pascal, it contains a complete library of Pascal procedures that allows you to search and sort data and build powerful database applications. Comes with source code for a free sample database—right on the disk. Minimum memory: 128K. Requires Turbo Pascal 2.0 or later. (CP/M-80 version available.)

9 Turbo Pascal Editor Toolbox®

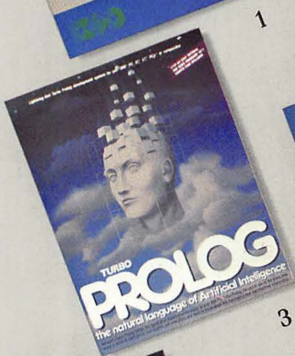
It's the only tool you need to build your own text editor or word processor. Comes with two sample editors—Simple Editor and MicroStar™—and their complete source code, plus information on how to install the features you need into your programs. Minimum memory: 192K. Requires Turbo Pascal 3.0.

10 Turbo Pascal GameWorks®

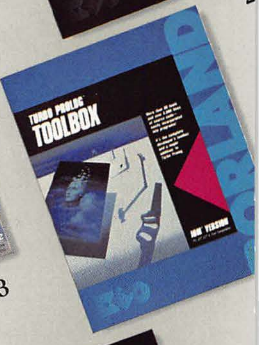
Teaches you techniques to quickly create your own computer games using Turbo Pascal. The secrets and strategies of the Masters are revealed for the first time in three classic games of strategy—Chess, Bridge and Go-Moku. Complete source code is included. You can play them "as is," customize them for greater challenge or *build a whole new set of games!* Minimum memory: 192K. Requires Turbo Pascal 3.0.

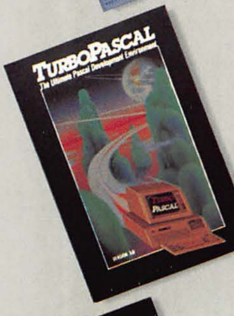
1

2

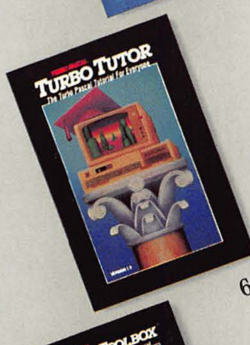


3

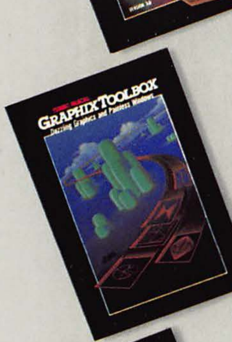




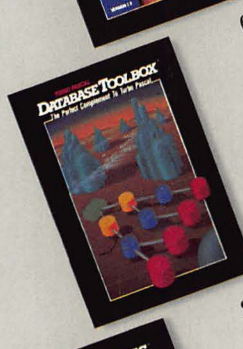
5



6

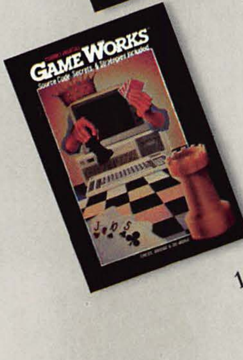


7



8

9



10

Programming Devel

1

Turbo Basic®

With a compilation speed of up to 12,000* lines per minute, Turbo Basic combines an interactive editor, fast memory-to-memory compiler and a trace debugging system. Program size not limited by 64K. Compatible with BASICA. Offers 8087 math support and true recursion. Comes with a free MicroCalc™ spreadsheet and source code. Minimum memory: 320K.

*Run on a 4.77 MHz IBM PC with 20MB hard disk using Turbo Basic version 1.0.

2

Turbo C®

With its RAM-based compiler and high-performance linker, Turbo C offers a compilation speed of up to 7,000* lines per minute. Fully compatible with the ANSI C standard. Generates native in-line code and linkable object modules. Supports tiny, small, compact, medium, large and huge memory model libraries. Minimum memory: 384K.

*Run on a 6 MHz IBM AT using Turbo C version 1.0 and Turbo Linker version 1.0.

3

Turbo Prolog®

The high-speed Prolog compiler. Brings 5th-generation programming language and supercomputer power to your IBM PC and compatibles. Both amateurs and professionals can build powerful expert systems, customized knowledge bases, natural language interfaces, and smart information-management systems. Minimum memory: 384K.

4

Turbo Prolog Toolbox™

A professional developer's toolbox to help you build powerful commercial applications in Prolog. Enhances Turbo Prolog with over 80 tools, 40 sample programs and 8,000 lines of source code that can easily be incorporated into your programs. Minimum memory: 512K.

5

Turbo Pascal® 3.0

The worldwide standard in high-speed Pascal compilers. Gives you a high-performance development tool featuring a completely integrated programming environment, a compiler which instantly locates programming errors, a full-screen editor, BCD reals, 8087 support and much more. Minimum memory: 128K. (CP/M-80® version available.)

BORLAND

TURBO PASCAL

NUMERICAL METHODS TOOLBOX™

Provides state-of-the-art math tools to solve scientific and engineering problems—fast!

Turbo Pascal Numerical Methods Toolbox for the Macintosh implements the latest high-level mathematical methods t solve common scientific and engineering problems. Fast.

So every time you need to calculate an integral, work with Fourier transforms, or incorporate any of the classical numerical analysis tools into your programs, you don't have to reinvent the wheel, because the Numerical Methods Toolbox is a complete collection of Turbo Pascal routines and programs that gives you applied state-of-the-art math tools. It also includes two graphics demo programs that use least-square and Fast Fourier Transform routines to give you the picture along with the numbers.

The Turbo Pascal Numerical Methods Toolbox is a must if you're involved with any type of scientific or engineering computing on the Macintosh. Because it comes with complete source code, you have total control of your application at all times.

What Numerical Methods Toolbox will do for you:

- Find solutions to equations
- Interpolations
- Calculus: numerical derivatives and integrals
- Matrix operations: inversions, dete minants, and eigenvalues
- Differential equations
- Least-squares approximations
- Fourier transforms
- Graphics

Five free ways to look at Least-Squares Fit!

As well as a free demo of Fast Fourier Transforms, you also get the Least-Squares Fit in five different forms—which gives you five different methods of fitting curves to a collection of data points. You instantly get the picture! The five different forms are

1. Power	4. 5-term Fourier
2. Exponential	5. 5-term Polynomial
3. Logarithm	

They're all ready to compile and run as is.

MACINTOSH™

Minimum system requirements: Macintosh 512K, Macintosh Plus, SE, or II, with one 800K disk drive (or two 400K).

BORLAND INTERNATIONAL

4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CA 95066

Les Postillons des Bruyeres - 65, rue de la Garenne
92318 SEVRES CEDEX - FRANCE

One Great Cumberland Place
London W1H 7AL - United Kingdom

ISBN 0-87524-166-2