TURBO PASCAL TOOLBOX™
NUMERICAL METHODS

A complete collection of Turbo Pascal® routines and programs
Borland’s No-Nonsense License Statement!

This software is protected by both United States copyright law and international treaty provisions. Therefore, you must treat this software just like a book, with the following single exception. Borland International authorizes you to make archival copies of the software for the sole purpose of backing-up our software and protecting your investment from loss.

By saying, “just like a book,” Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is no possibility of it being used at one location while it’s being used at another. Just like a book that can’t be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time. (Unless, of course, Borland’s copyright has been violated.)

Borland International grants you (the licensed owner of the Turbo Pascal Numerical Methods Toolbox) the right to incorporate toolbox routines into your programs. You may distribute your programs that contain Numerical Toolbox routines in executable form without restriction or fee, but you may not give away or sell any part of the actual Numerical Methods Toolbox source code. You are not, of course, restricted from distributing your own source code.

Sample programs are provided on the Numerical Methods Toolbox diskettes as examples of how to use the various toolbox features. You may edit or modify these sample programs and incorporate them into the programs that you write. Use of these sample programs is governed by the same conditions and restrictions as outlined in the first paragraph above.

WARRANTY

With respect to the physical diskette and physical documentation enclosed herein, Borland International, Inc. (“Borland”) warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, Borland will replace the defective diskette or documentation. If you need to return a product, call the Borland Customer Service Department to obtain a return authorization number. The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, and special, incidental, consequential, or other similar claims.

Borland International, Inc. specifically disclaims all other warranties, expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the diskette and documentation, and the program license granted herein in particular, and without limiting operation of the program license with respect to any particular application, use, or purpose. In no event shall Borland be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.

GOVERNING LAW

This statement shall be construed, interpreted, and governed by the laws of the state of California.
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: ___/___/____

Name: last first middle initial Title: __________________________

Company Name: __________________________ Department: ____________

Address: __________________________ Mail Stop: ________________

City: __________________________ State: __________ Zip: __________ Country: ______________

Phone # (_____________________) ____________________

Name: __________________________

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 ☐ full-service retailer ☐ other __________________________

Software was bought for:
☐ self ☐ company paid 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 ☐ other __________________________

Nature of business:
☐ finance/real estate/insurance ☐ computer consulting ☐ transportation/communication/utilities
☐ retail/wholesale ☐ other consulting ☐ mining/construction
☐ legal ☐ software publishing ☐ government
☐ health ☐ other publishing ☐ military
☐ professional services ☐ other services ☐ education
☐ other services ☐ manufacturing ☐ other __________________________

Nature of occupation:
☐ MIS/DP, systems analyst ☐ administration ☐ student
☐ programming ☐ finance/accounting ☐ consulting
☐ engineering/scientific ☐ sales/marketing ☐ teacher/trainer
☐ doctor/lawyer ☐ manufacturing/production ☐ clerical
☐ other professional ☐ other professional ☐ retired
☐ other professional ☐ purchasing ☐ other __________________________

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 9999

Other Borland products owned:

Programming languages
☐ Turbo Pascal ☐ Turbo Prolog ☐ Turbo Prolog TB
☐ Turbo Pascal Toolboxes ☐ Turbo Basic ☐ Turbo Basic Editor TB
☐ Tutor ☐ Turbo Basic Database TB ☐ Turbo Basic Telecm TB
☐ Database ☐ Turbo Basic Editor TB ☐ Turbo C
☐ Editor ☐ GameWorks ☐ Numerical Methods
☐ Graphix

What other software do you use:
☐ spreadsheet ☐ database ☐ languages ☐ desktop publishing
☐ word processor ☐ communications ☐ business graphics
☐ project management ☐ network ☐ CAD/CAM/CAE

What hardware peripherals do you use?
☐ modem ☐ laser printer ☐ hard disk ☐ EGA card
☐ mouse ☐ plotter ☐ other peripheral ☐ other printer

BOR 0045F
Turbo Pascal Numerical Methods Toolbox

Borland's No-Nonsense License Statement!

This software is protected by both United States copyright law and international treaty provisions. Therefore, you must treat this software just like a book, with the following single exception. Borland International, Inc. authorizes you to make archival copies of the software for the sole purpose of backing-up our software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is no possibility of it being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time. (Unless, of course, Borland's copyright has been violated.)

Borland International grants you (the licensed owner of the Turbo Pascal Numerical Methods Toolbox) the right to incorporate toolbox routines into your programs. You may distribute your programs that contain Numerical Toolbox routines in executable form without restriction or fee, but you may not give away or sell any part of the actual Numerical Methods Toolbox source code. You are not, of course, restricted from distributing your own source code.

Sample programs are provided on the Numerical Methods Toolbox diskettes as examples of how to use the various toolbox features. You may edit or modify these sample programs and incorporate them into the programs that you write. Use of these sample programs is governed by the same conditions and restrictions as outlined in the first paragraph above.

WARRANTY

With respect to the physical diskette and physical documentation enclosed herein, Borland International, Inc. ("Borland") warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, Borland will replace the defective diskette or documentation. If you need to return a product, call the Borland Customer Service Department to obtain a return authorization number. The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, and special, incidental, consequential, or other similar claims.

Borland International, Inc. specifically disclaims all other warranties, expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the diskette and documentation, and the program license granted herein in particular, and without limiting operation of the program license with respect to any particular application, use, or purpose. In no event shall Borland be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.

GOVERNING LAW

This statement shall be construed, interpreted, and governed by the laws of the state of California.

BORLAND INTERNATIONAL
4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CALIFORNIA 95066-9987
Turbo Pascal
Numerical Methods Toolbox™

For the Macintosh
# Table of Contents

**Introduction** ............................................................................................................... 1  
Toolbox Functions ........................................................................................................... 1  
About this Manual ......................................................................................................... 2  
On the Distribution Disks ........................................................................................ 3  
System Requirements ................................................................................................ 3  
Acknowledgements .................................................................................................... 3  

**Chapter 1. ROUTINE BEGINNINGS** ......................................................................... 5  
Using the Toolbox: An Example ............................................................................... 5  
The Distribution Disks ................................................................................................. 7  
Installation .................................................................................................................. 8  
Files on Distribution Disks ....................................................................................... 8  
The Graphics Demos ................................................................................................... 10  
Data Types and Defined Constants ....................................................................... 10  
Compiler Directives ................................................................................................. 11  

**Chapter 2. ROOTS TO EQUATIONS IN ONE VARIABLE** ...................................... 13  
Stopping Criteria ........................................................................................................... 15  
Root of a Function Using the Bisection Method (Bisect.pas) ............................. 16  
  Description .............................................................................................................. 16  
  User-Defined Function ...................................................................................... 16  
  Input Parameters ............................................................................................... 16  
  Output Parameters .......................................................................................... 17
Complex Roots of a Complex Polynomial Using Laguerre’s Method and Deflation (Laguerre.pas) ........................................... 35
Description ........................................................................................................ 35
User-Defined Types ......................................................................................... 35
Input Parameters ............................................................................................. 35
Output Parameters .......................................................................................... 36
Syntax of the Procedure Call .......................................................................... 36
Comments ....................................................................................................... 36
Sample Program .............................................................................................. 37
  Input Files ....................................................................................................... 37
  Example .......................................................................................................... 37

Chapter 3. INTERPOLATION ............................................................................ 39
Polynomial Interpolation Using Lagrange’s Method (Lagrange.pas) ............. 41
  Description ..................................................................................................... 41
  User-Defined Types ...................................................................................... 41
  Input Parameters .......................................................................................... 41
  Output Parameters ....................................................................................... 42
  Syntax of the Procedure Call ........................................................................ 42
  Sample Program ........................................................................................... 42
    Input Files .................................................................................................. 42
    Example ...................................................................................................... 42

Interpolation Using Newton’s Interpolary Divided-Difference Method (Divdif.pas) ............................................................................................................. 45
  Description ..................................................................................................... 45
  User-Defined Types ...................................................................................... 45
  Input Parameters .......................................................................................... 45
  Output Parameters ........................................................................................ 46
  Syntax of the Procedure Call ........................................................................ 46
  Sample Program ........................................................................................... 46
    Input Files .................................................................................................. 46
    Example ...................................................................................................... 46

Free Cubic Spline Interpolation (Cube_Fre.pas) ............................................. 48
  Description ..................................................................................................... 48
  User-Defined Types ...................................................................................... 48
  Input Parameters .......................................................................................... 48
  Output Parameters ....................................................................................... 49
  Syntax of the Procedure Call ........................................................................ 49
  Sample Program ........................................................................................... 49
    Input Files .................................................................................................. 50
    Example ...................................................................................................... 50

Clamped Cubic Spline Interpolation (Cube_Cla.pas) ....................................... 53
  Description ..................................................................................................... 53
  User-Defined Types ...................................................................................... 53

Table of Contents iii
Integration Using Adaptive Quadrature and Gaussian Quadrature
(Adapgaus.pas) ................................................................. 94
  Description .............................................................. 94
  User-Defined Function ............................................. 94
  Input Parameters ...................................................... 94
  Output Parameters ................................................... 95
  Syntax of the Procedure Call ...................................... 95
  Comments .............................................................. 95
  Sample Program ....................................................... 97
  Example ................................................................. 97

Integration Using the Romberg Algorithm (Romberg.pas) ............... 98
  Description .............................................................. 98
  User-Defined Function ............................................. 98
  Input Parameters ...................................................... 98
  Output Parameters ................................................... 99
  Syntax of the Procedure Call ...................................... 99
  Sample Program ....................................................... 99
  Example ................................................................. 99

Chapter 6. MATRIX ROUTINES ........................................... 101
Determinant of a Matrix (Det.pas) ............................................. 103
  Description .............................................................. 103
  User-Defined Types .................................................. 103
  Input Parameters ...................................................... 103
  Output Parameters ................................................... 104
  Syntax of the Procedure Call ...................................... 104
  Sample Program ....................................................... 104
    Input File ............................................................. 104
    Example .............................................................. 105

Inverse of a Matrix (Inverse.pas) ............................................ 106
  Description .............................................................. 106
  User-Defined Types .................................................. 106
  Input Parameters ...................................................... 106
  Output Parameters ................................................... 107
  Syntax of the Procedure Call ...................................... 107
  Sample Program ....................................................... 107
    Input Files ............................................................ 107
    Example .............................................................. 108

Solving a System of Linear Equations with Gaussian Elimination
(Gauselim.pas) .............................................................. 109
  Description .............................................................. 109
  User-Defined Types .................................................. 109
  Input Parameters ...................................................... 109
  Output Parameters ................................................... 110
Chapter 7. EIGENVALUES AND EIGENVECTORS ................................................................. 127
Real Dominant Eigenvalue and Eigenvector of a Real Matrix Using the
Power Method (Power.pas) ............................................................................................... 129
  Description .................................................................................................................. 129
  User-Defined Types .................................................................................................. 129
  Input Parameters ....................................................................................................... 129
  Output Parameters ..................................................................................................... 130
Syntax of the Procedure Call ................................................................. 130
Comments ............................................................................................... 130
Sample Program ..................................................................................... 131
  Input File ............................................................................................... 131
  Example .................................................................................................. 131
Real Eigenvalue and Eigenvector of a Real Matrix Using the Inverse
Power Method (InvPower.pas) ................................................................. 133
  Description ............................................................................................ 133
  User-Defined Types ............................................................................... 133
  Input Parameters ................................................................................... 133
  Output Parameters ................................................................................. 134
  Syntax of the Procedure Call ............................................................... 134
  Comments ............................................................................................. 135
  Sample Program .................................................................................... 135
    Input File ............................................................................................. 135
    Example ............................................................................................... 136
Real Eigenvalues and Eigenvectors of a Real Matrix Using
the Power Method and Wielandt's Deflation (Wielandt.pas) ................. 139
  Description ............................................................................................ 139
  User-Defined Types ............................................................................... 139
  Input Parameters ................................................................................... 139
  Output Parameters ................................................................................. 140
  Syntax of the Procedure Call ............................................................... 141
  Comments ............................................................................................. 141
  Sample Program .................................................................................... 141
    Input File ............................................................................................. 141
    Example ............................................................................................... 142
The Complete Eigensystem of a Symmetric Real Matrix Using the Cyclic
Jacobi Method (Jacobi.pas) .................................................................... 144
  Description ............................................................................................ 144
  User-Defined Types ............................................................................... 144
  Input Parameters ................................................................................... 144
  Output Parameters ................................................................................. 145
  Syntax of the Procedure Call ............................................................... 145
  Comments ............................................................................................. 146
  Sample Program .................................................................................... 146
    Input File ............................................................................................. 146
    Example ............................................................................................... 147

Chapter 8. INITIAL VALUE AND BOUNDARY VALUE METHODS .......... 149
Solution to an Initial Value Problem for a First-Order Ordinary Differential
Equation Using the Runge-Kutta Method (Runge_1.pas) ....................... 153
  Description ............................................................................................ 153
  User-Defined Types ............................................................................... 153
User-Defined Function ................................................................. 154
Input Parameters ........................................................................ 154
Output Parameters ...................................................................... 154
Syntax of the Procedure Call ....................................................... 154
Comments .................................................................................. 155
Sample Program ........................................................................ 155
Example ...................................................................................... 155

Solution to an Initial Value Problem for a First-Order Ordinary Differential
  Equation Using the Runge-Kutta-Fehlberg Method (RKF_1.pas) .......... 157
  Description ................................................................................ 157
  User-Defined Types ................................................................... 157
  User-Defined Function ............................................................... 157
  Input Parameters ....................................................................... 158
  Output Parameters ..................................................................... 158
  Syntax of the Procedure Call .................................................... 158
  Comments ............................................................................... 159
  Sample Program ....................................................................... 159
  Example ................................................................................... 159

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) ................................................................. 162
  Description ................................................................................ 162
  User-Defined Types ................................................................... 163
  User-Defined Function ............................................................... 163
  Input Parameters ....................................................................... 163
  Output Parameters ..................................................................... 163
  Syntax of the Procedure Call .................................................... 164
  Comments ............................................................................... 164
  Sample Program ....................................................................... 164
  Example ................................................................................... 164

Solution to an Initial Value Problem for a Second-Order Ordinary Differential
  Equation Using the Runge-Kutta Method (Runge_2.pas) .................. 166
  Description ................................................................................ 166
  User-Defined Types ................................................................... 167
  User-Defined Function ............................................................... 167
  Input Parameters ....................................................................... 167
  Output Parameters ..................................................................... 168
  Syntax of the Procedure Call .................................................... 168
  Comments ............................................................................... 168
  Sample Program ....................................................................... 169
  Example ................................................................................... 169

Solution to an Initial Value Problem for an nth-Order Ordinary Differential
  Equation Using the Runge-Kutta Method (Runge_N.pas) ................. 172
  Description ................................................................................ 172
Solution to an Initial Value Problem for a System of Coupled First-Order Ordinary Differential Equations Using the Runge-Kutta Method (Runge_S1.pas) ................................................................. 180
  Description .............................................................................................. 180
  User-Defined Types .................................................................................. 182
  User-Defined Functions .............................................................................. 182
  Input Parameters ........................................................................................ 183
  Output Parameters ..................................................................................... 184
  Syntax of the Procedure Call ..................................................................... 184
  Comments .................................................................................................. 185
  Sample Program ........................................................................................ 185
    Example .................................................................................................. 185

Solution to an Initial Value Problem for a System of Coupled Second-Order Ordinary Differential Equations Using the Runge-Kutta Method (Runge_S2.pas) .................................................. 190
  Description .............................................................................................. 190
  User-Defined Types .................................................................................. 193
  User-Defined Functions .............................................................................. 193
  Input Parameters ........................................................................................ 194
  Output Parameters ..................................................................................... 195
  Syntax of the Procedure Call ..................................................................... 195
  Comments .................................................................................................. 196
  Sample Program ........................................................................................ 196
    Example .................................................................................................. 197

Solution to Boundary Value Problem for a Second-Order Ordinary Differential Equation Using the Shooting and Runge-Kutta Methods (Shoot2.pas) .................................................... 202
  Description .............................................................................................. 202
  User-Defined Types .................................................................................. 203
  User-Defined Functions .............................................................................. 203
  Input Parameters ........................................................................................ 203
  Output Parameters ..................................................................................... 204
  Syntax of the Procedure Call ..................................................................... 204
  Comments .................................................................................................. 204
  Sample Program ........................................................................................ 205
    Example .................................................................................................. 205
Solution to a Boundary Value Problem for a Second-Order Ordinary Linear Differential Equation Using the Linear Shooting and Runge-Kutta Methods (Linshot2.pas) .................................................................................................... 209
Description ........................................................................................................ 209
User-Defined Types ........................................................................................... 210
User-Defined Functions ................................................................................... 210
Input Parameters ............................................................................................... 210
Output Parameters ............................................................................................ 210
Syntax of the Procedure Call ............................................................................ 211
Comments ........................................................................................................... 211
Sample Program ................................................................................................. 211
Example ............................................................................................................. 212

Chapter 9. LEAST-SQUARES APPROXIMATION ......................................... 215
Least-Squares Approximation (Least.pas) ............................................................ 216
Description ......................................................................................................... 216
User-Defined Types ........................................................................................... 217
Input Parameters ............................................................................................... 217
Output Parameters ............................................................................................ 218
Syntax of the Procedure Call ............................................................................ 218
Comments ........................................................................................................... 218
Poly ................................................................................................................. 218
Fourier ............................................................................................................. 219
Power ............................................................................................................... 219
Expo ............................................................................................................... 219
Log ................................................................................................................ 220
User ................................................................................................................ 220
Sample Program ................................................................................................. 221
Input Files .................................................................................................... 221
Example ............................................................................................................. 221

Chapter 10. FAST FOURIER TRANSFORM ROUTINES .......................... 227
The Application Programs .................................................................................... 228
Data Sampling ....................................................................................................... 232
User-Defined Types ............................................................................................... 233
Fast Fourier Transform Algorithms ..................................................................... 234
Procedure TestInput ......................................................................................... 234
Description ...................................................................................................... 234
Input Parameters ............................................................................................... 234
Output Parameters ............................................................................................ 234
Syntax of the Procedure Call ............................................................................ 234
Procedure MakeSinCosTable ........................................................................... 235
Description ...................................................................................................... 235
Input Parameters ............................................................................................... 235

Table of Contents  xi
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
<th>Input Parameters</th>
<th>Output Parameters</th>
<th>Syntax of the Procedure Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComplexFFT, RealFFT</td>
<td>Fast Fourier Transform Applications</td>
<td>ComplexFFT</td>
<td>Description</td>
<td>Input Parameters</td>
</tr>
<tr>
<td>RealFFT</td>
<td>RealFFT</td>
<td>Description</td>
<td>Input Parameters</td>
<td>Output Parameters</td>
</tr>
<tr>
<td>ComplexConvolution</td>
<td>ComplexConvolution</td>
<td>Description</td>
<td>Input Parameters</td>
<td>Output Parameters</td>
</tr>
<tr>
<td>RealConvolution</td>
<td>RealConvolution</td>
<td>Description</td>
<td>Input Parameters</td>
<td>Output Parameters</td>
</tr>
<tr>
<td>ComplexCrossCorrelation</td>
<td>ComplexCrossCorrelation</td>
<td>Description</td>
<td>Input Parameters</td>
<td>Output Parameters</td>
</tr>
<tr>
<td>RealCrossCorrelation</td>
<td>RealCrossCorrelation</td>
<td>Description</td>
<td>Input Parameters</td>
<td>Output Parameters</td>
</tr>
</tbody>
</table>
Chapter II. GRAPHICS PROGRAMS ............................................................. 255
Function of the Least-Squares Graphics Demonstration Program ............... 256
Function of the Fourier Transform Graphics Demonstration Program ........ 258
Rebuilding the Demonstration Programs ..................................................... 260

REFERENCES ............................................................................................. 261

INDEX .......................................................................................................... 263
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.
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.file
- FFTComplex
- FTDemo
- FTDemo.pas
- FFTMenu.r
- FFTMenu.rsrc
- FFTReal
- FFTRoutines
- LeastSquares
- LeastSquaresDemo
- LSQDemo.pas
- LSQMenu.r
- LSQMenu.rsrc
- Sample11A.dat
- Sample11B.dat
- TurboGraph.unit
NMT Disk 2:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Packed source for Chapter 2</td>
</tr>
<tr>
<td>3</td>
<td>Packed source for Chapter 3</td>
</tr>
<tr>
<td>4</td>
<td>Packed source for Chapter 4</td>
</tr>
<tr>
<td>5</td>
<td>Packed source for Chapter 5</td>
</tr>
<tr>
<td>6</td>
<td>Packed source for Chapter 6</td>
</tr>
<tr>
<td>7</td>
<td>Packed source for Chapter 7</td>
</tr>
<tr>
<td>8</td>
<td>Packed source for Chapter 8</td>
</tr>
<tr>
<td>9</td>
<td>Packed source for Chapter 9</td>
</tr>
<tr>
<td>10</td>
<td>Packed source for Chapter 10</td>
</tr>
<tr>
<td>UnPack</td>
<td>The program to unpack the packed files</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO Selection</td>
</tr>
<tr>
<td>IO Selection.rscre</td>
</tr>
<tr>
<td>IO Selection.unit</td>
</tr>
<tr>
<td>IO Selection.r</td>
</tr>
</tbody>
</table>
Chap2 “Roots to Equations in One Variable”

- Bisect.pas
- Laguerre.pas
- Muller.pas

Chap3 “Interpolation”

- Cube_cla.pas
- Cube_fre.pas
- Divdif.pas
- Interpolation
- Interpolation.unit

Chap4 “Numerical Differentiation”

- Deriv.pas
- Deriv2.pas
- Derivfn.pas

Chap5 “Numerical Integration”

- Adapgaus.pas
- Adapsimp.pas
- Integration

Chap6 “Matrix Routines”

- Det.pas
- Dirfact.pas
- Gauselim.pas
- Gaussidl.pas

Chap7 “Eigenvalues and Eigenvectors”

- EigenRoutines
- EigenRoutines.unit
- Invpower.pas

Chap8 “Initial Value and Boundary Value Methods”

- Adams_1.pas
- DifferentialEquat.unit
- Linshot2.pas
- RKF_1.pas

- Newtdefl.pas
- Raphson.pas
- Raphson2.pas
- Lagrange.pas
- Sample3A.dat
- Sample3B.dat
- Sample3C.dat
- Sample3D.dat
- Sample3E.dat
- Sample3F.dat
- Sample3G.dat
- Sample3H.dat
- Sample3I.dat
- Interdrv.pas
- Sample4A.dat
- Sample4B.dat
- Trapzoid.pas
- Sample6A.dat
- Sample6B.dat
- Sample6C.dat
- Sample6D.dat
- Wielandt.pas

- Roots of Equat
- Roots of Equat.unit
- Secant.pas
- Sample3E.dat
- Sample3F.dat
- Sample3G.dat
- Sample3H.dat
- Sample3I.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, TMatrix 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 dimen-
sion 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.
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 $|f'(x)| \leq TNNearlyZero$.

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.

```plaintext
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) <= TNearlyZero) { 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 (`TNearlyZero` 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`. 

Roots to Equations in One Variable
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

```pascal
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. \( \text{LeftEndpoint} < \text{RightEndpoint} \).
2. \( \text{TNTargetF(LeftEndpoint)} \times \text{TNTargetF(RightEndpoint)} < 0 \); the endpoints must have opposite signs.
3. \( \text{Tol} > 0 \).
4. \( \text{MaxIter} \geq 0 \).
Output Parameters

Answer: Extended;  An approximate root of \( T\!\!\!\!NTargetF \)
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: \( Tol \leq 0 \)
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

\[ \text{Bisect(LeftEndpoint, RightEndpoint, Tol, MaxIter, Answer, yAnswer, Iter, Error, } @TNTargetF); \]

The procedure \textit{Bisect} determines the roots of function \( T\!\!\!\!NTargetF \).

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 \textit{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 \( T\!\!\!\!NTargetF \). Note that the address of \( T\!\!\!\!NTargetF \) is passed into the \textit{Bisect} procedure.
Example

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

1. Write the following code for function \( \text{TNTargetF} \) into Bisect.pas:

   ```pascal
   {----------- 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.

   ```plaintext
   Left endpoint: 0.000000000000000e+0
   Right endpoint: 1.000000000000000e+2
   Tolerance: 1.000000000000000e-6
   Maximum number of iterations: 100
   ```

   ```plaintext
   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):

   ```pascal
   {---------- 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: 0.000000000000000e+0
   Tolerance: 1.000000000000000e-6
   Maximum number of iterations: 100

   Number of iterations: 5
   Calculated root: 7.39085133215161e-1
   Value of the function at the calculated root: 0.000000000000000e+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: 0.00000000000000000e+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

```pascal
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;
  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 \( \text{TNTargetF} \) into Secant.pas:

\[
\text{function TNTargetF}(x : \text{Extended}) : \text{Extended};
\]
\[
\begin{align*}
\text{TNTargetF} & := \cos(x) - x; \\
\end{align*}
\]

\[
\{ \text{function TNTargetF} \}
\]

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. \( \text{InitDegree} > 0 \)
2. \( \text{Tol} > 0 \)
3. \( \text{MaxIter} \geq 0 \)
4. \( \text{InitDegree} \leq \text{TNArraySize} \)

\( \text{TNArraySize} \) fixes an upper bound on the number of elements in each vector. It is used in the type definition of \( \text{TNvector} \). \( \text{TNArraySize} \) 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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree:Integer;</td>
<td>Degree of the deflated polynomial (( &gt; 2 ) if some of the roots are not approximated).</td>
</tr>
<tr>
<td>NumRoots:Integer;</td>
<td>Number of roots found.</td>
</tr>
<tr>
<td>Poly:TNvector;</td>
<td>Coefficients of the deflated polynomial.</td>
</tr>
<tr>
<td>Root:TNvector;</td>
<td>Real part of all roots found.</td>
</tr>
<tr>
<td>Imag:TNvector;</td>
<td>Imaginary part of all roots found (nonzero for 2 at most).</td>
</tr>
<tr>
<td>Value:TNvector;</td>
<td>Value of the polynomial at each approximate root.</td>
</tr>
<tr>
<td>Deriv:TNvector;</td>
<td>Value of the derivative at each found root.</td>
</tr>
<tr>
<td>Iter:TNIntVector;</td>
<td>Number of iterations required to find each root.</td>
</tr>
<tr>
<td>Error:Byte;</td>
<td>0: No error. 1: Maximum number of iterations exceeded. 2: The slope is zero (see “Comments”). 3: ( \text{Degree} \leq 0 ). 4: ( \text{Tol} \leq 0 ). 5: ( \text{MaxIter} &lt; 0 ).</td>
</tr>
</tbody>
</table>

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 \( TNearlyZero \), 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 ($\leq 30$)? 6

Input the coefficients of the polynomial where $\text{Poly}[n]$ is the coefficient of $x^n$

\[
\begin{align*}
\text{Poly}[6] &= 1 \\
\text{Poly}[5] &= 1 \\
\text{Poly}[4] &= -49 \\
\text{Poly}[3] &= 69 \\
\text{Poly}[2] &= 120 \\
\text{Poly}[1] &= 98 \\
\text{Poly}[0] &= -240
\end{align*}
\]

Initial approximation to the root: 0

Tolerance ($>$ 0): 1E-8

Maximum number of iterations ($\geq 0$): 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:

\[
\begin{align*}
\text{Poly}[6]: 1.00000000000000e+0 \\
\text{Poly}[5]: 1.00000000000000e+0 \\
\text{Poly}[4]: -4.90000000000000e+1 \\
\text{Poly}[3]: 6.90000000000000e+1 \\
\text{Poly}[2]: 1.20000000000000e+2 \\
\text{Poly}[1]: 9.80000000000000e+1 \\
\text{Poly}[0]: -2.40000000000000e+2
\end{align*}
\]

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: 3.4694469519536e-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**

```pascal
TComplex = record
    Re, Im: Extended;
end;
```

**User-Defined Procedure**

```pascal
procedure TNTargetF(x: TComplex; var y: TComplex);
```

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

**Input Parameters**

- **Guess**: TComplex; 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. $MaxIter \geq 0$

**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: $Tol \leq 0$
4: $MaxIter < 0$

If $Error \leq 2$, 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:

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

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: \(-4.00000000000000e+0 + 4.00000000000000e+0\) 
Tolerance: \(1.00000000000000e-6\) 
Maximum number of iterations: 100

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

\[
\begin{align*}
\text{TNcomplex} &= \text{record} \\
& \quad \text{Re, Im:Extended;} \\
& \quad \text{end;} \\
\text{TNIntVector} &= \text{array}[0..\text{TNArraySize}] \text{ of Integer;} \\
\text{TNCompVector} &= \text{array}[0..\text{TNArraySize}] \text{ of TNcomplex;}
\end{align*}
\]

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. \( \text{degree} > 0 \)
2. \( \text{Tol} > 0 \)
3. \( \text{MaxIter} \geq 0 \)
4. \( \text{degree} \leq \text{TNArraySize} \)
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 ≥ MaxIter
2: Degree ≤ 0
3: Tol ≤ 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
\[
\text{Re}(\text{Poly}[1]) = 2 \\
\text{Im}(\text{Poly}[1]) = -2 \\
\text{Re}(\text{Poly}[0]) = -1 \\
\text{Re}(\text{Poly}[0]) = 0
\]

Initial approximation to the root:
\[
\text{Re}(\text{Approximation}) = 1 \\
\text{Im}(\text{Approximation}) = 0
\]

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:
\[
\text{InitPoly}[4]: 1.00000000000000e+0 + 0.00000000000000e+0 \\
\text{InitPoly}[3]: -2.00000000000000e+0 + -2.00000000000000e+0 \\
\text{InitPoly}[2]: 0.00000000000000e+0 + 4.00000000000000e+0 \\
\text{InitPoly}[1]: 2.00000000000000e+0 + -2.00000000000000e+0 \\
\text{InitPoly}[0]: -1.00000000000000e+0 + 0.00000000000000e+0
\]

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

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

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

Root 3
Number of iterations: 2
Calculated root: 1.34424834689770e-10 + 9.9999999865575e-11
Value of the function at the calculated root: -1.08420217248550e-19 + 1.44222068471458e-19

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

\[
\text{TNvector} = \text{array}[0..\text{TNArraySize}] \text{ of Extended};
\]
\[
\text{TNmatrix} = \text{array}[0..\text{TNArraySize}] \text{ of TNvector};
\]

*Input Parameters*

The parameters for Lagrange:

- \( \text{NumPoints}: \text{Integer} \); Number of data points
- \( \text{XData}: \text{TNvector} \); The \( x \)-coordinates of the data points
- \( \text{YData}: \text{TNvector} \); The \( y \)-coordinates of the data points
- \( \text{NumInter}: \text{Integer} \); Number of interpolations
- \( \text{XInter}: \text{TNvector} \); 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 \((\text{XInter})\) must be unique.
2. \( \text{NumPoints}, \text{NumInter} \leq \text{TNArraySize} \).
3. \( \text{NumPoints} > 0 \).
**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 \( \text{sqr}(x) \) 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:**

<table>
<thead>
<tr>
<th>(x)</th>
<th>(\cos(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000000</td>
<td>9.99847695156391e-1</td>
</tr>
<tr>
<td>2.000000</td>
<td>9.99390827019096e-1</td>
</tr>
<tr>
<td>3.000000</td>
<td>9.98629534754574e-1</td>
</tr>
<tr>
<td>4.000000</td>
<td>9.97564050259824e-1</td>
</tr>
<tr>
<td>5.000000</td>
<td>9.96194698091746e-1</td>
</tr>
<tr>
<td>6.000000</td>
<td>9.94521895368273e-1</td>
</tr>
<tr>
<td>7.000000</td>
<td>9.92546151641322e-1</td>
</tr>
<tr>
<td>8.000000</td>
<td>9.90268068741570e-1</td>
</tr>
<tr>
<td>9.000000</td>
<td>9.87688340595138e-1</td>
</tr>
<tr>
<td>10.0000</td>
<td>9.84807753012208e-1</td>
</tr>
<tr>
<td>11.0000</td>
<td>9.81627183447664e-1</td>
</tr>
<tr>
<td>12.0000</td>
<td>9.78147600733806e-1</td>
</tr>
<tr>
<td>13.0000</td>
<td>9.74370064785235e-1</td>
</tr>
<tr>
<td>14.0000</td>
<td>9.70295726275996e-1</td>
</tr>
<tr>
<td>15.0000</td>
<td>9.65925826289068e-1</td>
</tr>
<tr>
<td>16.0000</td>
<td>9.61261695938319e-1</td>
</tr>
<tr>
<td>17.0000</td>
<td>9.56304755963035e-1</td>
</tr>
<tr>
<td>18.0000</td>
<td>9.51056516295154e-1</td>
</tr>
<tr>
<td>19.0000</td>
<td>9.45518575599317e-1</td>
</tr>
<tr>
<td>20.0000</td>
<td>9.39692620785908e-1</td>
</tr>
</tbody>
</table>
The polynomial:

Poly[19] = -1.72014247146006e-28
Poly[18] = 3.54986012534706e-26
Poly[16] = 2.02588035084664e-22
Poly[14] = 2.5163094794110e-19
Poly[12] = 1.03284343326638e-16
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[ 2] = -1.52308336619420e-4
Poly[ 1] = -2.49984780967393e-10
Poly[ 0] = 1.00000000000000e+0

<table>
<thead>
<tr>
<th>X</th>
<th>Interpolated Y value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.500</td>
<td>9.99657324975254e-1</td>
</tr>
<tr>
<td>2.500</td>
<td>9.99048221581889e-1</td>
</tr>
<tr>
<td>3.500</td>
<td>9.98134798421861e-1</td>
</tr>
<tr>
<td>4.500</td>
<td>9.969177333733130e-1</td>
</tr>
<tr>
<td>5.500</td>
<td>9.95396198367178e-1</td>
</tr>
<tr>
<td>6.500</td>
<td>9.93571855676588e-1</td>
</tr>
<tr>
<td>7.500</td>
<td>9.9144461373810e-1</td>
</tr>
<tr>
<td>8.500</td>
<td>9.89015863361917e-1</td>
</tr>
<tr>
<td>9.500</td>
<td>9.86285601537232e-1</td>
</tr>
<tr>
<td>10.500</td>
<td>9.83254907563954e-1</td>
</tr>
<tr>
<td>11.500</td>
<td>9.79924704620830e-1</td>
</tr>
<tr>
<td>12.500</td>
<td>9.76296007119933e-1</td>
</tr>
<tr>
<td>13.500</td>
<td>9.72369920397676e-1</td>
</tr>
<tr>
<td>14.500</td>
<td>9.68147640378107e-1</td>
</tr>
<tr>
<td>15.500</td>
<td>9.6363043208623e-1</td>
</tr>
<tr>
<td>16.500</td>
<td>9.58819734868193e-1</td>
</tr>
<tr>
<td>17.500</td>
<td>9.53716950748227e-1</td>
</tr>
<tr>
<td>18.500</td>
<td>9.48323655206198e-1</td>
</tr>
<tr>
<td>19.500</td>
<td>9.42641491092216e-1</td>
</tr>
<tr>
<td>20.500</td>
<td>9.36672189246619e-1</td>
</tr>
</tbody>
</table>

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**

\[
\text{TNvector} = \text{array}[0..\text{TNArraySize}] \text{ of Extended};
\]

\[
\text{TNmatrix} = \text{array}[0..\text{TNArraySize}] \text{ of TNvector};
\]

**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. \(\text{NumPoints}, \text{NumInter} \leq \text{TNArraySize}\).
3. \(\text{NumPoints} > 0\).

\(\text{TNArraySize}\) fixes an upper bound on the number of elements in each vector. It is used in the type definition of \(\text{TNvector}\). \(\text{TNArraySize}\) 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 interpolatory 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 \( \text{sqr}(x) \) could be entered in a text file as

\[
\begin{align*}
  1 & \ 1 \\
  2 & \ 4 \\
  3 & \ 9 \\
  4 & \ 16 \\
  5 & \ 25 \\
\end{align*}
\]

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.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.000</td>
<td>0.9781476</td>
</tr>
<tr>
<td>8.000</td>
<td>0.9902681</td>
</tr>
<tr>
<td>1.000</td>
<td>0.9998477</td>
</tr>
<tr>
<td>10.000</td>
<td>0.9848078</td>
</tr>
<tr>
<td>5.000</td>
<td>0.9961947</td>
</tr>
<tr>
<td>15.000</td>
<td>0.9659258</td>
</tr>
<tr>
<td>4.000</td>
<td>0.9975641</td>
</tr>
<tr>
<td>3.000</td>
<td>0.9986295</td>
</tr>
<tr>
<td>7.000</td>
<td>0.9925462</td>
</tr>
<tr>
<td>14.000</td>
<td>0.9702957</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Interpolated Y value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.500</td>
<td>9.99656668284607e-1</td>
</tr>
<tr>
<td>2.500</td>
<td>9.99047982204853e-1</td>
</tr>
<tr>
<td>3.500</td>
<td>9.98134846782587e-1</td>
</tr>
<tr>
<td>4.500</td>
<td>9.96917355869352e-1</td>
</tr>
<tr>
<td>5.500</td>
<td>9.95396200633579e-1</td>
</tr>
<tr>
<td>6.500</td>
<td>9.93571893532269e-1</td>
</tr>
<tr>
<td>7.500</td>
<td>9.91444906399794e-1</td>
</tr>
<tr>
<td>8.500</td>
<td>9.89015879894104e-1</td>
</tr>
<tr>
<td>9.500</td>
<td>9.86285623948171e-1</td>
</tr>
<tr>
<td>10.500</td>
<td>9.83254980952454e-1</td>
</tr>
<tr>
<td>11.500</td>
<td>9.79924765142406e-1</td>
</tr>
<tr>
<td>12.500</td>
<td>9.76295923083642e-1</td>
</tr>
<tr>
<td>13.500</td>
<td>9.72369781236267e-1</td>
</tr>
<tr>
<td>14.500</td>
<td>9.68147753391414e-1</td>
</tr>
<tr>
<td>15.500</td>
<td>9.63629212784400e-1</td>
</tr>
</tbody>
</table>

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:

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

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 \( i \)th cubic polynomial at \( x \), where \( x[i] \leq x \leq x[i+1] \).

User-Defined Types

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

Input Parameters

\textbf{NumPoints:Integer; } Number of data points
\textbf{XData:TNvector; } The \( x \)-coordinates of the data points
\textbf{YData:TNvector; } The \( y \)-coordinates of the data points
\textbf{NumInter:Integer; } Number of interpolations
\textbf{XInter:TNvector; } \( X \)-coordinates of points at which to interpolate

Turbo Pascal Numerical Methods Toolbox
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. $\text{NumPoints}, \text{NumInter} \leq \text{TNArraystack}$.
4. $\text{NumPoints} > 1$.

$\text{TNArraystack}$ fixes an upper bound on the number of elements in each vector. It is used in the type definition of $\text{TNvector}$. $\text{TNArraystack}$ 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**

- $\text{CoefO:TNvector}$; Coefficient of the constant term
- $\text{Coef1:TNvector}$; Coefficient of the linear term
- $\text{Coef2:TNvector}$; Coefficient of the squared term
- $\text{Coef3:TNvector}$; Coefficient of the cubed term
- $\text{YInter:TNvector}$; Interpolated values at $X\text{Inter}$
- $\text{Error:Byte}$; 0: No error
  - 1: $X$-values of the data points not unique
  - 2: $X$-values of the data points not in ascending order
  - 3: $\text{NumPoints} < 2$

**Syntax of the Procedure Call**

```
CubicSplineFree(NumPoints, XData, YData, NumInter, XInter,
               CoefO, 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 \( \text{sqr}(x) \) 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:

![Figure](image.png)

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:

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000000000</td>
<td>2.8000000000</td>
</tr>
<tr>
<td>2</td>
<td>0.1000000000</td>
<td>2.7000000000</td>
</tr>
<tr>
<td>3</td>
<td>0.2000000000</td>
<td>2.6000000000</td>
</tr>
<tr>
<td>4</td>
<td>0.6000000000</td>
<td>2.2000000000</td>
</tr>
<tr>
<td>5</td>
<td>1.0000000000</td>
<td>1.8000000000</td>
</tr>
<tr>
<td>6</td>
<td>1.4000000000</td>
<td>1.6000000000</td>
</tr>
<tr>
<td>7</td>
<td>1.8000000000</td>
<td>1.4000000000</td>
</tr>
<tr>
<td>8</td>
<td>2.0000000000</td>
<td>1.4200000000</td>
</tr>
<tr>
<td>9</td>
<td>2.2000000000</td>
<td>1.4000000000</td>
</tr>
<tr>
<td>10</td>
<td>2.6000000000</td>
<td>1.5000000000</td>
</tr>
<tr>
<td>11</td>
<td>3.0000000000</td>
<td>1.8000000000</td>
</tr>
<tr>
<td>12</td>
<td>3.4000000000</td>
<td>2.4000000000</td>
</tr>
<tr>
<td>13</td>
<td>3.4500000000</td>
<td>2.6000000000</td>
</tr>
<tr>
<td>14</td>
<td>3.5000000000</td>
<td>2.8000000000</td>
</tr>
<tr>
<td>15</td>
<td>3.5500000000</td>
<td>2.9000000000</td>
</tr>
</tbody>
</table>

Splines:

<table>
<thead>
<tr>
<th>Coef0</th>
<th>Coef1</th>
<th>Coef2</th>
<th>Coef3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>2.8000000000</td>
<td>-0.9988332302</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>2:</td>
<td>2.7000000000</td>
<td>-1.0023335396</td>
<td>-0.0350003094</td>
</tr>
<tr>
<td>3:</td>
<td>2.6000000000</td>
<td>-0.9918326113</td>
<td>0.1400123770</td>
</tr>
<tr>
<td>4:</td>
<td>2.2000000000</td>
<td>-1.0723397281</td>
<td>-0.3412801689</td>
</tr>
<tr>
<td>5:</td>
<td>1.8000000000</td>
<td>-0.7188084763</td>
<td>1.2251082984</td>
</tr>
<tr>
<td>6:</td>
<td>1.6000000000</td>
<td>-0.5524263669</td>
<td>-0.8091530249</td>
</tr>
<tr>
<td>7:</td>
<td>1.4000000000</td>
<td>-0.0714860563</td>
<td>2.0115038012</td>
</tr>
<tr>
<td>8:</td>
<td>1.4200000000</td>
<td>0.0406713524</td>
<td>-1.4507167575</td>
</tr>
<tr>
<td>9:</td>
<td>1.4000000000</td>
<td>-0.0911993534</td>
<td>0.7913632286</td>
</tr>
<tr>
<td>10:</td>
<td>1.5000000000</td>
<td>0.6158534153</td>
<td>0.9762686929</td>
</tr>
<tr>
<td>11:</td>
<td>1.8000000000</td>
<td>0.6277856923</td>
<td>-0.9464380003</td>
</tr>
<tr>
<td>12:</td>
<td>2.4000000000</td>
<td>3.6230038155</td>
<td>8.4344833084</td>
</tr>
<tr>
<td>13:</td>
<td>2.6000000000</td>
<td>4.3322682035</td>
<td>5.7508044511</td>
</tr>
<tr>
<td>14:</td>
<td>2.8000000000</td>
<td>3.0479233704</td>
<td>-31.4377011128</td>
</tr>
</tbody>
</table>

Interpolated Points:

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>0.3000000000</td>
<td>2.5018157855</td>
</tr>
<tr>
<td>2:</td>
<td>0.5000000000</td>
<td>2.3042222482</td>
</tr>
<tr>
<td>3:</td>
<td>1.2000000000</td>
<td>1.6916808945</td>
</tr>
<tr>
<td>4:</td>
<td>1.6000000000</td>
<td>1.4759298485</td>
</tr>
<tr>
<td>5:</td>
<td>2.1000000000</td>
<td>1.4132967676</td>
</tr>
<tr>
<td>6:</td>
<td>2.3000000000</td>
<td>1.3989477848</td>
</tr>
<tr>
<td>7:</td>
<td>2.5000000000</td>
<td>1.4480232575</td>
</tr>
<tr>
<td>8:</td>
<td>2.7000000000</td>
<td>1.5697457729</td>
</tr>
<tr>
<td>9:</td>
<td>2.9000000000</td>
<td>1.7293596306</td>
</tr>
<tr>
<td>10:</td>
<td>3.2000000000</td>
<td>1.9502390938</td>
</tr>
<tr>
<td>11:</td>
<td>3.3000000000</td>
<td>2.1142270171</td>
</tr>
</tbody>
</table>
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**.

<table>
<thead>
<tr>
<th>Data</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.550</td>
<td>2.900</td>
</tr>
<tr>
<td>2</td>
<td>3.600</td>
<td>2.800</td>
</tr>
<tr>
<td>3</td>
<td>3.650</td>
<td>2.650</td>
</tr>
<tr>
<td>4</td>
<td>3.800</td>
<td>2.500</td>
</tr>
<tr>
<td>5</td>
<td>4.000</td>
<td>2.350</td>
</tr>
<tr>
<td>6</td>
<td>4.300</td>
<td>2.200</td>
</tr>
<tr>
<td>7</td>
<td>4.800</td>
<td>1.950</td>
</tr>
<tr>
<td>8</td>
<td>5.300</td>
<td>1.600</td>
</tr>
<tr>
<td>9</td>
<td>5.600</td>
<td>1.300</td>
</tr>
<tr>
<td>10</td>
<td>5.800</td>
<td>1.200</td>
</tr>
<tr>
<td>11</td>
<td>6.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Splines:  
<table>
<thead>
<tr>
<th>CoefO</th>
<th>Coef1</th>
<th>Coef2</th>
<th>Coef3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.900</td>
<td>-1.6719664279</td>
<td>-313.2134288</td>
</tr>
<tr>
<td>2</td>
<td>2.800</td>
<td>-2.6560671441</td>
<td>-19.6820143244</td>
</tr>
<tr>
<td>3</td>
<td>2.650</td>
<td>-2.7037649955</td>
<td>18.7280572976</td>
</tr>
<tr>
<td>4</td>
<td>2.500</td>
<td>-0.4016786037</td>
<td>3.3808146854</td>
</tr>
<tr>
<td>5</td>
<td>2.350</td>
<td>-0.7704798556</td>
<td>1.5368084259</td>
</tr>
<tr>
<td>6</td>
<td>2.200</td>
<td>-0.420082166</td>
<td>-0.3688182960</td>
</tr>
<tr>
<td>7</td>
<td>1.950</td>
<td>-0.4754252188</td>
<td>0.2581334916</td>
</tr>
<tr>
<td>8</td>
<td>1.600</td>
<td>-1.2782163082</td>
<td>-1.8637156703</td>
</tr>
<tr>
<td>9</td>
<td>1.300</td>
<td>0.1155473174</td>
<td>6.5095944222</td>
</tr>
<tr>
<td>10</td>
<td>1.200</td>
<td>-3.0330135193</td>
<td>-22.252396055</td>
</tr>
</tbody>
</table>

Interpolated Points:  
<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.700</td>
</tr>
<tr>
<td>2</td>
<td>3.900</td>
</tr>
<tr>
<td>3</td>
<td>4.100</td>
</tr>
<tr>
<td>4</td>
<td>4.200</td>
</tr>
<tr>
<td>5</td>
<td>4.500</td>
</tr>
<tr>
<td>6</td>
<td>4.600</td>
</tr>
<tr>
<td>7</td>
<td>5.000</td>
</tr>
<tr>
<td>8</td>
<td>5.200</td>
</tr>
<tr>
<td>9</td>
<td>5.500</td>
</tr>
<tr>
<td>10</td>
<td>5.700</td>
</tr>
<tr>
<td>11</td>
<td>5.900</td>
</tr>
</tbody>
</table>
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:

\[ S[i](x) = \text{Coeff0}[i] + \text{Coeff1}[i](x - x[i]) + \text{Coeff2}[i](x - x[i])^2 \\
+ \text{Coeff3}[i](x - x[i])^3 \]

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 \( i \)th cubic polynomial at \( x \), where \( x[i] \leq x \leq x[i + 1] \).

User-Defined Types

\[ \text{TNvector} = \text{array}[0..\text{TNArraySize}] \text{ of Extended}; \]

Input Parameters

- \( \text{NumPoints:Integer} \); Number of data points
- \( \text{XData:TNvector} \); The \( x \)-coordinates of the data points
- \( \text{YData:TNvector} \); The \( y \)-coordinates of the data points
- \( \text{DerivLE:Extended} \); Derivative of the function at the left endpoint
- \( \text{DerivRE:Extended} \); 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 ≤ 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 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:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000000000</td>
<td>2.8000000000</td>
</tr>
<tr>
<td>0.1000000000</td>
<td>2.7000000000</td>
</tr>
<tr>
<td>0.2000000000</td>
<td>2.6000000000</td>
</tr>
<tr>
<td>0.6000000000</td>
<td>2.2000000000</td>
</tr>
<tr>
<td>1.0000000000</td>
<td>1.8000000000</td>
</tr>
<tr>
<td>1.4000000000</td>
<td>1.6000000000</td>
</tr>
<tr>
<td>1.8000000000</td>
<td>1.4000000000</td>
</tr>
<tr>
<td>2.0000000000</td>
<td>1.4200000000</td>
</tr>
<tr>
<td>2.2000000000</td>
<td>1.4000000000</td>
</tr>
<tr>
<td>2.6000000000</td>
<td>1.5000000000</td>
</tr>
<tr>
<td>3.0000000000</td>
<td>1.8000000000</td>
</tr>
<tr>
<td>3.4000000000</td>
<td>2.4000000000</td>
</tr>
<tr>
<td>3.4500000000</td>
<td>2.6000000000</td>
</tr>
<tr>
<td>3.5000000000</td>
<td>2.8000000000</td>
</tr>
<tr>
<td>3.5500000000</td>
<td>2.9000000000</td>
</tr>
</tbody>
</table>

Derivative at \( x = 0.0000000000000000e+0 \) : -1.3333333333333333e+0
Derivative at \( x = 3.5500000000000000e+0 \) : 3.0000000000000000e+0

<table>
<thead>
<tr>
<th>Splines</th>
<th>Coef0</th>
<th>Coef1</th>
<th>Coef2</th>
<th>Coef3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>2.8000000000</td>
<td>-1.3333333333</td>
<td>5.7579845570</td>
<td>-24.246512365</td>
</tr>
<tr>
<td>2:</td>
<td>2.7000000000</td>
<td>-0.9091317890</td>
<td>-1.5159611140</td>
<td>6.0728700429</td>
</tr>
<tr>
<td>3:</td>
<td>2.6000000000</td>
<td>-1.0301395105</td>
<td>0.3058918989</td>
<td>-0.5763578064</td>
</tr>
<tr>
<td>4:</td>
<td>2.2000000000</td>
<td>-1.0620777385</td>
<td>-0.3857374687</td>
<td>1.3523295373</td>
</tr>
<tr>
<td>5:</td>
<td>1.8000000000</td>
<td>-0.7215495356</td>
<td>1.2370579761</td>
<td>-1.7079603429</td>
</tr>
<tr>
<td>6:</td>
<td>1.6000000000</td>
<td>-0.5517241193</td>
<td>-0.8124944355</td>
<td>2.3545118344</td>
</tr>
<tr>
<td>7:</td>
<td>1.4000000000</td>
<td>-0.0715539872</td>
<td>2.0129176588</td>
<td>-5.7757491499</td>
</tr>
<tr>
<td>8:</td>
<td>1.4200000000</td>
<td>0.0405240212</td>
<td>-1.4525297241</td>
<td>3.7495480911</td>
</tr>
<tr>
<td>9:</td>
<td>1.4000000000</td>
<td>-0.0905420975</td>
<td>0.7971991306</td>
<td>0.1353902832</td>
</tr>
<tr>
<td>10:</td>
<td>1.5000000000</td>
<td>0.6122045428</td>
<td>0.9596674704</td>
<td>-1.3579470688</td>
</tr>
<tr>
<td>11:</td>
<td>1.8000000000</td>
<td>0.6417239262</td>
<td>-0.8858690121</td>
<td>7.578979919</td>
</tr>
<tr>
<td>12:</td>
<td>2.4000000000</td>
<td>3.5708997526</td>
<td>8.2088085781</td>
<td>7.4639274157</td>
</tr>
<tr>
<td>14:</td>
<td>2.8000000000</td>
<td>2.6380599835</td>
<td>-45.5223993401</td>
<td>655.2239934014</td>
</tr>
</tbody>
</table>
### Interpolated Points

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3000000000</td>
<td>2.4994686101</td>
</tr>
<tr>
<td>0.5000000000</td>
<td>2.3029267570</td>
</tr>
<tr>
<td>1.2000000000</td>
<td>1.6915087292</td>
</tr>
<tr>
<td>1.6000000000</td>
<td>1.4759914934</td>
</tr>
<tr>
<td>2.1000000000</td>
<td>1.4132766530</td>
</tr>
<tr>
<td>2.3000000000</td>
<td>1.3990531718</td>
</tr>
<tr>
<td>2.5000000000</td>
<td>1.4482408301</td>
</tr>
<tr>
<td>2.7000000000</td>
<td>1.5692791819</td>
</tr>
<tr>
<td>2.9000000000</td>
<td>1.7265068643</td>
</tr>
<tr>
<td>3.2000000000</td>
<td>1.9535412087</td>
</tr>
<tr>
<td>3.3000000000</td>
<td>2.1174192125</td>
</tr>
</tbody>
</table>

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? Sample31.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**.

<table>
<thead>
<tr>
<th>Data :</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>3.5500000000</td>
<td>2.9000000000</td>
</tr>
<tr>
<td>2:</td>
<td>3.6000000000</td>
<td>2.8000000000</td>
</tr>
<tr>
<td>3:</td>
<td>3.6500000000</td>
<td>2.6500000000</td>
</tr>
<tr>
<td>4:</td>
<td>3.8000000000</td>
<td>2.5000000000</td>
</tr>
<tr>
<td>5:</td>
<td>4.0000000000</td>
<td>2.3500000000</td>
</tr>
<tr>
<td>6:</td>
<td>4.3000000000</td>
<td>2.2000000000</td>
</tr>
<tr>
<td>7:</td>
<td>4.8000000000</td>
<td>1.9500000000</td>
</tr>
<tr>
<td>8:</td>
<td>5.3000000000</td>
<td>1.6000000000</td>
</tr>
<tr>
<td>9:</td>
<td>5.6000000000</td>
<td>1.3000000000</td>
</tr>
<tr>
<td>10:</td>
<td>5.8000000000</td>
<td>1.2000000000</td>
</tr>
<tr>
<td>11:</td>
<td>6.0000000000</td>
<td>0.0000000000</td>
</tr>
</tbody>
</table>

**Derivative at X= 3.55000000000000000e+0 : -4.00000000000000000e+0**

**Derivative at X= 6.00000000000000000e+0 : -1.70000000000000000e+1**

### Splines:

<table>
<thead>
<tr>
<th>Splines:</th>
<th>Coef0</th>
<th>Coef1</th>
<th>Coef2</th>
<th>Coef3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>2.9000000000</td>
<td>-4.0000000000</td>
<td>80.2233303937</td>
<td>-804.4666078741</td>
</tr>
<tr>
<td>2:</td>
<td>2.8000000000</td>
<td>-2.011665197</td>
<td>-40.446660784</td>
<td>431.3998236224</td>
</tr>
<tr>
<td>3:</td>
<td>2.6500000000</td>
<td>-2.955339213</td>
<td>21.563127559</td>
<td>-56.851685392</td>
</tr>
<tr>
<td>4:</td>
<td>2.5000000000</td>
<td>-0.3236290709</td>
<td>-4.0199470867</td>
<td>9.4454622054</td>
</tr>
<tr>
<td>5:</td>
<td>2.3500000000</td>
<td>-0.7983524409</td>
<td>1.6473302365</td>
<td>-2.1760736673</td>
</tr>
<tr>
<td>6:</td>
<td>2.2000000000</td>
<td>-0.3974941891</td>
<td>-0.3111360640</td>
<td>0.2122488846</td>
</tr>
<tr>
<td>7:</td>
<td>1.9500000000</td>
<td>-0.5494435897</td>
<td>0.0072372629</td>
<td>-0.6167001671</td>
</tr>
<tr>
<td>8:</td>
<td>1.6000000000</td>
<td>-1.0047314521</td>
<td>-0.9176129877</td>
<td>3.1119483153</td>
</tr>
<tr>
<td>9:</td>
<td>1.3000000000</td>
<td>-0.7151931996</td>
<td>1.8829404961</td>
<td>-4.0348724916</td>
</tr>
<tr>
<td>10:</td>
<td>1.2000000000</td>
<td>-0.4462017001</td>
<td>-0.5379829989</td>
<td>-136.1550425028</td>
</tr>
<tr>
<td>Interpolated Points:</td>
<td>X</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>-----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>3.7000000000</td>
<td>2.5490351248</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>3.9000000000</td>
<td>2.4368630843</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>4.1000000000</td>
<td>2.2844619846</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>4.2000000000</td>
<td>2.2388141319</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td>4.5000000000</td>
<td>2.1097537107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:</td>
<td>4.6000000000</td>
<td>2.0584802174</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7:</td>
<td>5.0000000000</td>
<td>1.8354671712</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8:</td>
<td>5.2000000000</td>
<td>1.6919117155</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:</td>
<td>5.5000000000</td>
<td>1.3872367766</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:</td>
<td>5.7000000000</td>
<td>1.2432752125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:</td>
<td>5.9000000000</td>
<td>1.0138495975</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>$t$</th>
<th>$f(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>45.0</td>
</tr>
<tr>
<td>1.1</td>
<td>49.2</td>
</tr>
<tr>
<td>1.2</td>
<td>54.5</td>
</tr>
<tr>
<td>1.3</td>
<td>59.8</td>
</tr>
<tr>
<td>1.4</td>
<td>65.1</td>
</tr>
<tr>
<td>1.5</td>
<td>70.4</td>
</tr>
</tbody>
</table>

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

\[
\text{TNvector} = \text{array}[1..\text{TArraySize}] \text{ of Extended};
\]

Input Parameters

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

The preceding parameters must satisfy the following conditions:

1. \text{XData} points must be unique.
2. \text{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. \text{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 \( \text{sqr}(x) \) 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 \( f(x) = \text{sqr}(x) * \cos(x) \) 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:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000000</td>
<td>5.40302305868140e-1</td>
</tr>
<tr>
<td>1.1000000</td>
<td>5.48651306924949e-1</td>
</tr>
<tr>
<td>1.2000000</td>
<td>5.21795166446410e-1</td>
</tr>
<tr>
<td>1.3000000</td>
<td>4.52073020375553e-1</td>
</tr>
<tr>
<td>1.4000000</td>
<td>3.33135600084472e-1</td>
</tr>
<tr>
<td>1.5000000</td>
<td>1.59158703752332e-1</td>
</tr>
<tr>
<td>1.6000000</td>
<td>-7.47507770912994e-2</td>
</tr>
<tr>
<td>1.7000000</td>
<td>-3.72360588514066e-1</td>
</tr>
<tr>
<td>1.8000000</td>
<td>-7.36134786805602e-1</td>
</tr>
<tr>
<td>1.9000000</td>
<td>-1.16707533637725e+0</td>
</tr>
<tr>
<td>2.0000000</td>
<td>-1.66458734618857e+0</td>
</tr>
</tbody>
</table>

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

Using 2-point differentiation:

<table>
<thead>
<tr>
<th>X</th>
<th>Derivative at X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.100</td>
<td>8.54900105680900e-2</td>
</tr>
<tr>
<td>1.300</td>
<td>-6.97221460708570e-1</td>
</tr>
<tr>
<td>1.500</td>
<td>-1.73976963321400e+0</td>
</tr>
<tr>
<td>2.000</td>
<td>-4.9751209811320e+0</td>
</tr>
<tr>
<td>2.200</td>
<td>No derivative calculated</td>
</tr>
</tbody>
</table>

Using 3-point differentiation:

<table>
<thead>
<tr>
<th>X</th>
<th>Derivative at X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.100</td>
<td>-9.25356971086500e-2</td>
</tr>
<tr>
<td>1.300</td>
<td>-9.43297831809690e-1</td>
</tr>
<tr>
<td>1.500</td>
<td>-2.03943188587886e+0</td>
</tr>
<tr>
<td>2.000</td>
<td>-5.30797739311566e+0</td>
</tr>
<tr>
<td>2.200</td>
<td>No derivative calculated</td>
</tr>
</tbody>
</table>
Using 5-point differentiation:

<table>
<thead>
<tr>
<th>X</th>
<th>Derivative at X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.100</td>
<td>-8.08749392678308e-2</td>
</tr>
<tr>
<td>1.300</td>
<td>-9.32986606435739e-1</td>
</tr>
<tr>
<td>1.500</td>
<td>-2.03221450709713e0</td>
</tr>
<tr>
<td>2.000</td>
<td>-5.30200229054730e0</td>
</tr>
<tr>
<td>2.200</td>
<td>No derivative calculated</td>
</tr>
</tbody>
</table>

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

\[ \text{TNvector} = \text{array}[1..\text{TNArraySize}] \text{ of Extended;} \]

Input Parameters

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

The preceding parameters must satisfy the following conditions:

1. \text{XData} points must be unique.
2. \text{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. \text{Point} must equal 3 or 5.
5. \textit{XData} points must be equally spaced.

6. \textit{XDeriv} points must be a subset of the \textit{XData} points.

7. \textit{NumPoints}, \textit{NumDeriv} \leq \textit{TNArraySize}.

\textit{TNArraySize} represents the number of elements in each vector. It is used in the \texttt{type} definition of \textit{TNvector}. \textit{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).

\textbf{Output Parameters}

\begin{tabular}{ll}
\textit{YDeriv} : \texttt{TNvector}; & Approximation to the second derivative at the \textit{XDeriv} points \\
\textit{Error} : \texttt{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: \textit{Point} not equal to 3 or 5 \\
 & 6: X-value points not equally spaced
\end{tabular}

\textbf{Syntax of the Procedure Call}

\texttt{Second_Derivative(NumPoints, XData, YData, Point, NumDeriv, XDeriv, YDeriv, Error);} 

\textbf{Comments}

If an x-value at which the second derivative is approximated is not among the data points, the value $-9.9999999\times10^{35}$ is arbitrarily assigned to the derivative at that point and \textit{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 $9.9999999\times10^{35}$ is arbitrarily assigned to the second derivative at that point and \textit{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 \( f(x) = \text{sqr}(x) \times \cos(x) \) 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:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000000</td>
<td>5.40302305868140e-1</td>
</tr>
<tr>
<td>1.1000000</td>
<td>5.48851306924949e-1</td>
</tr>
<tr>
<td>1.2000000</td>
<td>5.21795166446410e-1</td>
</tr>
<tr>
<td>1.3000000</td>
<td>4.52073020375553e-1</td>
</tr>
<tr>
<td>1.4000000</td>
<td>3.33135600084472e-1</td>
</tr>
<tr>
<td>1.5000000</td>
<td>1.59158703752332e-1</td>
</tr>
<tr>
<td>1.6000000</td>
<td>-7.47507770912994e-2</td>
</tr>
<tr>
<td>1.7000000</td>
<td>-3.72360588514066e-1</td>
</tr>
<tr>
<td>1.8000000</td>
<td>-7.36134786805602e-1</td>
</tr>
<tr>
<td>1.9000000</td>
<td>-1.1670753367725e+0</td>
</tr>
<tr>
<td>2.0000000</td>
<td>-1.66458734618857e+0</td>
</tr>
</tbody>
</table>

Using 3-point second differentiation:

<table>
<thead>
<tr>
<th>X</th>
<th>Second Derivative at X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1000</td>
<td>-3.56051415353480e+0</td>
</tr>
<tr>
<td>1.3000</td>
<td>-4.92152742202240e+0</td>
</tr>
<tr>
<td>1.5000</td>
<td>-5.99325845114914e+0</td>
</tr>
<tr>
<td>2.0000</td>
<td>-6.65714602396720e+0</td>
</tr>
<tr>
<td>2.2000</td>
<td>No 2nd derivative calculated.</td>
</tr>
</tbody>
</table>

Using 5-point second differentiation:

<table>
<thead>
<tr>
<th>X</th>
<th>Second Derivative at X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1000</td>
<td>-3.61167369644120e+0</td>
</tr>
<tr>
<td>1.3000</td>
<td>-4.92756964541466e+0</td>
</tr>
<tr>
<td>1.5000</td>
<td>-6.00263647117238e+0</td>
</tr>
<tr>
<td>2.0000</td>
<td>-6.59765891992320e+0</td>
</tr>
<tr>
<td>2.2000</td>
<td>No 2nd derivative calculated.</td>
</tr>
</tbody>
</table>
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
(Interdrc.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 \( x \) is an adequate approximation to \( f'(x) \). 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

\[ \text{TNvector} = \text{array}[1..\text{TNArraySize}] \text{ of Extended}; \]

Input Parameters

- \text{NumPoints} : Integer; Number of data points
- \text{XData} : TNvector; \( X \)-coordinates of data points
- \text{YData} : TNvector; \( Y \)-coordinates of data points
- \text{NumDeriv} : Integer; Number of points at which the derivative is to be approximated
- \text{XDeriv} : TNvector; \( X \)-coordinates of points at which the derivative is to be approximated
The preceding parameters must satisfy the following conditions:

1. \textit{XData} points must be unique.
2. \textit{XData} points must be in ascending order.
3. \textit{NumPoints} \geq 2.
4. \textit{NumPoints}, \textit{NumDeriv} \leq \textit{TNArraySize}.

\textit{TNArraySize} represents the number of elements in each vector. It is used in the \texttt{type} definition of \texttt{TNvector}. \textit{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).

\textbf{Output Parameters}

\begin{itemize}
\item \texttt{YInter} : \texttt{TNvector}; \quad \text{Interpolated } y\text{-values at the } XDeriv \text{ points}
\item \texttt{YDeriv} : \texttt{TNvector}; \quad \text{Approximation to the first derivative at the } x\text{-values in } XDeriv
\item \texttt{YDeriv2} : \texttt{TNvector}; \quad \text{Approximation to the second derivative at the } x\text{-values in } XDeriv
\item \texttt{Error} : \texttt{Byte}; \quad 0: \text{No errors} \\
\quad 1: X\text{-values not unique} \\
\quad 2: X\text{-values not in ascending order} \\
\quad 3: \text{NumPoints < 2}
\end{itemize}

\textbf{Syntax of the Procedure Call}

\begin{verbatim}
Interpolate_Derivative(NumPoints, XData, YData, NumDeriv, 
                      XDeriv, YInter, YDeriv, YDeriv2, Error);
\end{verbatim}

\textbf{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 \( \text{sqr}(x) \) could be entered in a text file as

\[
\begin{array}{c}
1 1 \\
2 4 \\
3 9 \\
4 16 \\
5 25 \\
\end{array}
\]

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:

\[
\begin{array}{c}
1 \\
2 \\
3 \\
4 \\
5 \\
\end{array}
\]

Example

Problem. Determine the first and second derivative of \( f(x) = \text{sqr}(x) \times \cos(x) \) 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? Sample48.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:**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000</td>
<td>0.5403023</td>
</tr>
<tr>
<td>1.100</td>
<td>0.5488513</td>
</tr>
<tr>
<td>1.200</td>
<td>0.5217952</td>
</tr>
<tr>
<td>1.300</td>
<td>0.4520730</td>
</tr>
<tr>
<td>1.400</td>
<td>0.3331356</td>
</tr>
<tr>
<td>1.500</td>
<td>0.1591587</td>
</tr>
<tr>
<td>1.600</td>
<td>-0.0747508</td>
</tr>
<tr>
<td>1.700</td>
<td>-0.3723606</td>
</tr>
<tr>
<td>1.800</td>
<td>-0.7361348</td>
</tr>
<tr>
<td>1.900</td>
<td>-1.1670753</td>
</tr>
<tr>
<td>2.000</td>
<td>-1.6645873</td>
</tr>
</tbody>
</table>

Using free cubic spline interpolation:

<table>
<thead>
<tr>
<th>X</th>
<th>Value at X</th>
<th>1st Deriv at X</th>
<th>2nd Deriv at X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.100</td>
<td>5.48851300000000e-1</td>
<td>-5.8601566681664e-2</td>
<td>-4.32274700</td>
</tr>
<tr>
<td>1.300</td>
<td>4.52073000000000e-1</td>
<td>-9.31377366861403e-1</td>
<td>-4.98862501</td>
</tr>
<tr>
<td>1.550</td>
<td>4.99429267146237e-2</td>
<td>-2.33770918101853e+0</td>
<td>-6.19118137</td>
</tr>
<tr>
<td>1.950</td>
<td>-1.41057141673716e+0</td>
<td>-5.01018588841894e+0</td>
<td>-4.20790661</td>
</tr>
<tr>
<td>2.200</td>
<td>-2.57545316779455e+0</td>
<td>-3.43222090956673e+0</td>
<td>16.83162644</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>X</th>
<th>Value at X</th>
<th>1st Deriv at X</th>
<th>2nd Deriv at X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>0.5488513</td>
<td>-0.0804494</td>
<td>-3.5629715</td>
</tr>
<tr>
<td>1.3</td>
<td>0.4520730</td>
<td>-0.9329164</td>
<td>-4.9275779</td>
</tr>
<tr>
<td>1.55</td>
<td>0.0499596</td>
<td>-2.3375165</td>
<td>-6.2070293</td>
</tr>
<tr>
<td>1.95</td>
<td>-1.4076126</td>
<td>-4.9760746</td>
<td>-6.5786348</td>
</tr>
<tr>
<td>2.20</td>
<td>-2.8483454</td>
<td>-6.5025275</td>
<td>-5.4434252</td>
</tr>
</tbody>
</table>

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

\[
f'(x) = \frac{[f(x + \Delta X) - f(x - \Delta X)]}{2\Delta X}
\]

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**

\[
\text{TNvector} = \text{array}[1..\text{TNArraySize}] \text{ of Extended};
\]

**User-Defined Functions**

\[
\text{function} \ \text{TNTargetF}(X : \text{Extended}) : \text{Extended};
\]

**Input Parameters**

\[
\begin{align*}
\text{NumDeriv} & : \text{Integer}; \quad \text{Number of points at which the derivative is to be approximated} \\
\text{XDeriv} & : \text{TNvector}; \quad \text{X-coordinates of points at which the derivative is to be approximated} \\
\text{Tolerance} & : \text{Extended}; \quad \text{Indicates accuracy of solution}
\end{align*}
\]

The preceding parameters must satisfy the following conditions:

1. \( \text{NumDeriv} \leq \text{TNArraySize} \)
2. \( \text{Tolerance} > \text{TNNearlyZero} \)

\( \text{TNArraySize} \) represents the number of elements in each vector. It is used in the type definition of \( \text{TNvector} \). \( \text{TNArraySize} \) 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**

```pascal
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 \( f(x) = \text{sqr}(x) \times \cos(x) \) 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:

```pascal
{ ----- here is the function to differentiate ------------------- }
function TNTargetF(X: Extended): Extended;
begin
  TNTargetF := Sqr(X)*Cos(X);
end; { function TNTargetF }
{ ------- allow for application ------------------------------- }
```

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 = 1.000000000000000e-4

<table>
<thead>
<tr>
<th>X</th>
<th>Derivative at X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.100</td>
<td>-8.0449385380667e-2</td>
</tr>
<tr>
<td>1.300</td>
<td>-9.32916380187814e-1</td>
</tr>
<tr>
<td>1.550</td>
<td>-2.33751652942971e+0</td>
</tr>
<tr>
<td>1.950</td>
<td>-4.97607456093026e+0</td>
</tr>
<tr>
<td>2.200</td>
<td>-6.50252751007340e+0</td>
</tr>
</tbody>
</table>

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

$$f''(x) = \frac{f(x + \Delta X) - 2f(x) + f(x - \Delta X)}{\Delta X^2}$$

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. $\text{NumDeriv} \leq \text{TNArraySize}$
2. $\text{Tolerance} \geq \text{TNNearlyZero}$
$TNArraySize$ 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 \( f(x) = \text{sqr}(x)^2 \times \cos(x) \) 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:

```pascal
{ ----- 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 = $1.00000000000000e-4$

<table>
<thead>
<tr>
<th>X</th>
<th>Second Derivative at X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.100</td>
<td>$-3.56297144833630e+0$</td>
</tr>
<tr>
<td>1.300</td>
<td>$-4.9275792729853e+0$</td>
</tr>
<tr>
<td>1.550</td>
<td>$-6.207025616294e+0$</td>
</tr>
<tr>
<td>1.950</td>
<td>$-6.57863482851564e+0$</td>
</tr>
<tr>
<td>2.200</td>
<td>$-5.44342518062510e+0$</td>
</tr>
</tbody>
</table>

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

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

The procedure \textit{Simpson} approximates the integral of this function.

Input Parameters

- \texttt{LowerLimit : Extended; Lower limit of integration}
- \texttt{UpperLimit : Extended; Upper limit of integration}
- \texttt{NumIntervals : Integer; Number of subintervals over which to apply Simpson's rule}

The preceding parameters must satisfy the following condition:

\[ \text{NumIntervals} > 0 \]
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(3x) + sqrt(x)/3 from 0 to 5 using Simpson's composite algorithm.

1. Code function TNTargetF:
   
   ```pascal
   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**

```pascal
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:

$\text{NumIntervals} > 0$
**Output Parameters**

Integral : Extended; Approximation to the integral of the function

Error : Byte;  
0: No errors  
1: NumIntervals \( \leq 0 \)

**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 \( \exp(3x) + \frac{sqr(x)}{3} \) from 0 to 5 using the trapezoid composite rule.

1. **Code function** *TNTargetF*:

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

Numerical Integration 89
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: 0.0000000000000000e+0
   Upper Limit: 5.0000000000000000e+0
   Number of intervals: 100

   Integral: 1.0917283832081019e+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

```pascal
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 ≤ 0
2: MaxIntervals ≤ 0
3: NumIntervals ≥ 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 \( \exp(3x) + \sqrt{x}/3 \) from 0 to 5 using adaptive quadrature and Simpson's rule.

1. Code function \( \text{TNTTargetF} \):

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

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

```pascal
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 recur-
sion.

Gaussian quadrature uses orthogonal polynomials (in this case, Legendre polyno-
mials) 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:

\[ \text{Integral from \(-1\) to \(1\) of } f(x) \, dx \]

equals

\[ \text{Sum from } i = 1 \text{ to } \text{NumLegendreTerms} \text{ of } \]

\[ \text{Legendre}[i].\text{Weight} \times f(\text{Legendre}[i].\text{Root}) \]

for an arbitrary function \(f(x)\).

\[\begin{align*}
\text{Legendre}[1] & \quad \text{Root: } 0.0950125098376370440185 \\
& \quad \text{Weight: } 0.189450610455068496285 \\
\text{Legendre}[2] & \quad \text{Root: } 0.281603550778258913230 \\
& \quad \text{Weight: } 0.18260341504492358867 \\
\text{Legendre}[3] & \quad \text{Root: } 0.4580167776757227386342 \\
& \quad \text{Weight: } 0.169156519395002538189 \\
\text{Legendre}[4] & \quad \text{Root: } 0.617876244402643748447 \\
& \quad \text{Weight: } 0.149595988816576732081 \\
\text{Legendre}[5] & \quad \text{Root: } 0.75540440835500303395 \\
& \quad \text{Weight: } 0.124628971255533872052 \\
\text{Legendre}[6] & \quad \text{Root: } 0.865631202387831743880 \\
& \quad \text{Weight: } 0.09515851168249278410 \\
\text{Legendre}[7] & \quad \text{Root: } 0.944575023073232576078 \\
& \quad \text{Weight: } 0.06225352393864792863 \\
\text{Legendre}[8] & \quad \text{Root: } 0.989400934991649932596 \\
& \quad \text{Weight: } 0.02715245941175409452 \\
\text{Legendre}[9] & \quad \text{Root: } -0.0950125098376370440185 \\
& \quad \text{Weight: } 0.189450610455068496285 \\
\text{Legendre}[10] & \quad \text{Root: } -0.281603550778258913230 \\
& \quad \text{Weight: } 0.18260341504492358867 \\
\text{Legendre}[11] & \quad \text{Root: } -0.4580167776757227386342 \\
& \quad \text{Weight: } 0.169156519395002538189 \\
\text{Legendre}[12] & \quad \text{Root: } -0.617876244402643748447 \\
& \quad \text{Weight: } 0.149595988816576732081 \\
\text{Legendre}[13] & \quad \text{Root: } -0.75540440835500303395 \\
& \quad \text{Weight: } 0.124628971255533872052 \\
\text{Legendre}[14] & \quad \text{Root: } -0.865631202387831743880 \\
& \quad \text{Weight: } 0.09515851168249278410 \\
\text{Legendre}[15] & \quad \text{Root: } -0.944575023073232576078 \\
& \quad \text{Weight: } 0.06225352393864792863 \\
\text{Legendre}[16] & \quad \text{Root: } -0.989400934991649932596 \\
& \quad \text{Weight: } 0.027152459411754094852
\end{align*}\]
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 $\exp(3x) + \sqrt{x}/3$ from 0 to 5 using adaptive quadrature with Gaussian quadrature algorithm.

1. Code function $TNTargetF$:

   ```pascal
   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: 1.08968601304609e+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

```pascal
function TNTargetF(x : Extended) : Extended;
The procedure Romberg approximates the integral of this function.
```

Input Parameters

```pascal
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 ≥ 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(3x) + sqrt(x)/3 from 0 to 5 using the Romberg algorithm.

1. Code function TNTargetF:

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

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

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

\[ \begin{align*}
2X + Y + Z &= 7 \\
X - Y + Z &= 2 \\
X + Y - Z &= 0
\end{align*} \]

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

\[ Ax = b \]

where

\[ A = \begin{bmatrix}
2 & 1 & 1 \\
1 & -1 & 1 \\
1 & 1 & -1
\end{bmatrix} \quad x = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad b = \begin{bmatrix} 7 \\ 2 \\ 0 \end{bmatrix} \]

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

\[ \begin{align*}
3X - 3Y &= 4 \\
-2X + 2Y &= 5
\end{align*} \]
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 accu­
ricy 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 sys­
  tems whose coefficients are all of nearly the same magnitude), Gaussian elimi­
  nation without pivoting may be used. It is somewhat faster than its pivoting
  counterpart.

- When considering sparse coefficient matrices, the Gaussian elimination rou­
  tine 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 cre­
  ates less round-off error than the direct methods. However, the Gauss-Seidel
  algorithm has its own weaknesses (see the section, “Solving a System of Lin­
  ear 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**

\[
\text{TNvector} = \text{array}[1..\text{TNArraySize}] \text{ of Extended};
\]
\[
\text{TNmatrix} = \text{array}[1..\text{TNArraySize}] \text{ of TNvector};
\]

**Input Parameters**

\[\text{Dimen} : \text{Integer}; \quad \text{Dimension of the data matrix}\]
\[\text{Data} : \text{TNmatrix}; \quad \text{The square matrix}\]

The preceding parameters must satisfy the following conditions:

1. \( \text{Dimen} > 0 \)
2. \( \text{Dimen} \leq \text{TNArraySize} \)

\( \text{TNArraySize} \) sets an upper bound on the number of elements in each vector. It is used in the type definition of \( \text{TNvector} \) and \( \text{TNmatrix} \). \( \text{TNArraySize} \) 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,
   
   \[
   \begin{bmatrix}
   1, 1, [1, 2] \ldots [1, N], [2, 1] \ldots [2, N] \ldots [N, N],
   \end{bmatrix}
   \]

   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:

\[
\begin{bmatrix}
1.00000000 & 2.00000000 & 0.00000000 & -1.00000000 \\
-1.00000000 & 4.00000000 & 3.00000000 & -0.50000000 \\
2.00000000 & 2.00000000 & 1.00000000 & -3.00000000 \\
0.00000000 & 0.00000000 & 3.00000000 & -4.00000000
\end{bmatrix}
\]

Determinant = \(-2.100000000000000e+1\)
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

```pascal
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 : TMatrix; 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,
   \[ [1, 1], [1, 2] \ldots [1, N], [2, 1] \ldots [2, N] \ldots [N, N], \]
   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:

\[
\begin{array}{c}
2 \\
2 & 3 \\
-4 & 0
\end{array}
\]
Example

Problem. Invert the following matrix:

\[
\begin{pmatrix}
1 & 2 & 0 & -1.0 \\
-1 & 4 & 3 & -0.5 \\
2 & 2 & 1 & -3.0 \\
0 & 0 & 3 & -4.0
\end{pmatrix}
\]

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:

\[
\begin{pmatrix}
1.000000000 & 2.000000000 & 0.000000000 & -1.000000000 \\
-1.000000000 & 4.000000000 & 3.000000000 & -0.500000000 \\
2.000000000 & 2.000000000 & 1.000000000 & -3.000000000 \\
0.000000000 & 0.000000000 & 3.000000000 & -4.000000000
\end{pmatrix}
\]

Inverse:

\[
\begin{pmatrix}
-1.952380952 & 0.190476190 & 1.571428571 & -0.714285714 \\
0.761904762 & 0.047619048 & -0.357142857 & 0.071428571 \\
-1.904761905 & 0.380952381 & 1.142857143 & -0.428571429 \\
-1.428571429 & 0.285714286 & 0.857142857 & -0.571428571
\end{pmatrix}
\]

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:

\[
\begin{pmatrix}
-1.952380952 & 0.190476190 & 1.571428571 & -0.714285714 \\
0.761904762 & 0.047619048 & -0.357142857 & 0.071428571 \\
-1.904761905 & 0.380952381 & 1.142857143 & -0.428571429 \\
-1.428571429 & 0.285714286 & 0.857142857 & -0.571428571
\end{pmatrix}
\]

Inverse:

\[
\begin{pmatrix}
1.000000000 & 2.000000000 & 0.000000000 & -1.000000000 \\
-1.000000000 & 4.000000000 & 3.000000000 & -0.500000000 \\
2.000000000 & 2.000000000 & 1.000000000 & -3.000000000 \\
-0.000000000 & -0.000000000 & 3.000000000 & -4.000000000
\end{pmatrix}
\]
**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**

\[
\begin{align*}
\text{TNvector} & = \text{array}[1..\text{TNArraySize}] \text{ of Extended}; \\
\text{TNmatrix} & = \text{array}[1..\text{TNArraySize}] \text{ of TNvector};
\end{align*}
\]

**Input Parameters**

\[
\begin{align*}
\text{Dimen} : \text{Integer}; & \quad \text{Dimension of the coefficients matrix} \\
\text{Coefficients} : \text{TNmatrix; } & \quad \text{The square matrix containing the coefficients of the equations} \\
\text{Constants} : \text{TNvector; } & \quad \text{The constant terms of each equation}
\end{align*}
\]

The preceding parameters must satisfy the following conditions:

1. \(\text{Dimen} > 0\)
2. \(\text{Dimen} \leq \text{TNArraySize}\)

\(\text{TNArraySize}\) sets an upper bound on the number of elements in each vector. It is used in the type definition of \(\text{TNvector}\) and \(\text{TNmatrix}\). \(\text{TNArraySize}\) 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,
   
   \begin{align*}
   &1, 1, [1, 2], ..., [1, N], [2, 1], ..., [2, N], ..., [N, N], \\
   \end{align*}
   
   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

\[2x + 3y = 10\]
\[-4x = 10\]

a text file could be created to look like this:

\[2 2 3\]
\[\begin{array}{cc}
-4 & 0 \\
10 & 10 \\
\end{array}\]

**Example**

**Problem.** Solve the following linear system:

\[w + 2x + 0y - z = 10.0\]
\[-w + 4x + 3y - 0.5z = 21.5\]
\[2w + 2x + y - 3z = 26.0\]
\[3y - 4z = 37.0\]

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:

\[\begin{array}{cccc}
1.000000000 & 2.000000000 & 0.000000000 & -1.000000000 \\
-1.000000000 & 4.000000000 & 3.000000000 & -0.500000000 \\
2.000000000 & 2.000000000 & 1.000000000 & -3.000000000 \\
0.000000000 & 0.000000000 & 3.000000000 & -4.000000000 \\
\end{array}\]

The constants:

\[\begin{array}{cccc}
1.00000000000000e+1 \\
2.15000000000000e+1 \\
2.60000000000000e+1 \\
3.70000000000000e+1 \\
\end{array}\]

The solution:

\[\begin{array}{cccc}
-1.00000000000000e+0 \\
2.00000000000000e+0 \\
3.00000000000000e+0 \\
-7.00000000000000e+0 \\
\end{array}\]
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

\[ \text{TNvector} = \text{array}[1..\text{TNArraySize}] \text{ of } \text{Extended}; \]
\[ \text{TNmatrix} = \text{array}[1..\text{TNArraySize}] \text{ of } \text{TNvector}; \]

Input Parameters

\text{Dimen} : \text{Integer}; \quad \text{Dimension of the coefficients matrix}
\text{Coefficients} : \text{TNmatrix}; \quad \text{The square matrix containing the coefficients of the equations}
\text{Constants} : \text{TNvector}; \quad \text{The constant terms of each equation}

The preceding parameters must satisfy the following conditions:

1. \( \text{Dimen} > 0 \)
2. \( \text{Dimen} \leq \text{TNArraySize} \)

\text{TNArraySize} sets an upper bound on the number of elements in each vector. It is used in the type definition of \text{TNvector} and \text{TNmatrix}. \text{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: Tvector; 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

\[ 2x + 3y = 10 \]
\[ -4x = 10 \]

a text file could be created to look like this:

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

**Example**

**Problem.** Solve the following linear system:

\[ w + 2x + 0y - z = 10 \]
\[ -w + 4x + 3y - 0.5z = 21.5 \]
\[ 2w + 2x + y - 3z = 26 \]
\[ 3y - 4z = 37 \]

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

```pascal
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. $\text{Dimen} > 0$
2. $\text{Dimen} \leq \text{TNArraySize}$

$\text{TNArraySize}$ fixes an upper bound on the number of elements in each vector. It is used in the type definition of $\text{TNvector}$ and $\text{TNmatrix}$. $\text{TNArraySize}$ 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 : $\text{TNvector}$; Solution to each system of equations
Error : Byte; 0: No errors
1: $\text{Dimen} < 1$

**Syntax of the Procedure Call**

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

**Sample Program**

The sample program Dirfact.pngas 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

\[
\begin{align*}
2x + 3y &= 10 \\
-4x &= 10 \\
2x + 3y &= 1 \\
-4x &= 2
\end{align*}
\]

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:

\[
\begin{align*}
2w + x + 5y - 8z \\
7w + 6x + 2y + 2z \\
-1w - 3x - 10y + 4z \\
2w + 2w + 2y + z
\end{align*}
\]

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:
\[
\begin{bmatrix}
2.000000000 & 1.000000000 & 5.000000000 & -8.000000000 \\
7.000000000 & 6.000000000 & 2.000000000 & 2.000000000 \\
-1.000000000 & -3.000000000 & -10.000000000 & 4.000000000 \\
2.000000000 & 2.000000000 & 2.000000000 & 1.000000000
\end{bmatrix}
\]

The constants:
\[
\begin{bmatrix}
0.00000000000000 \\
1.70000000000000 \\
-1.00000000000000 \\
7.00000000000000
\end{bmatrix}
\]

The solution:
\[
\begin{bmatrix}
1.00000000000000 \\
1.00000000000000 \\
1.00000000000000 \\
1.00000000000000
\end{bmatrix}
\]

The constants:
\[
\begin{bmatrix}
-1.50000000000000 \\
5.00000000000000 \\
-5.00000000000000 \\
1.70000000000000
\end{bmatrix}
\]

The solution:
\[
\begin{bmatrix}
2.00000000000000 \\
5.00000000000000 \\
-2.26268279475236 \times 10^{-19} \\
3.00000000000000
\end{bmatrix}
\]

The constants:
\[
\begin{bmatrix}
1.40000000000000 \\
1.00000000000000 \\
-1.20000000000000 \\
1.00000000000000
\end{bmatrix}
\]

The solution:
\[
\begin{bmatrix}
1.00000000000000 \\
-1.00000000000000 \\
1.00000000000000 \\
-1.00000000000000
\end{bmatrix}
\]

The constants:
\[
\begin{bmatrix}
-1.30000000000000 \\
8.40000000000000 \\
-5.10000000000000 \\
3.70000000000000
\end{bmatrix}
\]

The solution:
\[
\begin{bmatrix}
4.00000000000000 \\
5.00000000000000 \\
6.00000000000000 \\
7.00000000000000
\end{bmatrix}
\]
The constants:
5.00000000000000e+0
3.00000000000000e+1
-1.50000000000000e+1
1.00000000000000e+1

The solution:
1.9825411540207e-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 \( X_0 \), the sequence \( X_m = TX_{m-1} + C \) is generated with the following formula:

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

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

\[
\text{TNvector} = \text{array}[1..\text{TNArraySize}] \text{ of Extended};
\]

\[
\text{TNmatrix} = \text{array}[1..\text{TNArraySize}] \text{ of TNvector};
\]
**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. $\text{Dimen} > 0$.
2. $\text{Dimen} \leq \text{TNArraySize}$.
3. $\text{Tol} > 0$.
4. $\text{MaxIter} \geq 0$.
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: $\text{Iter} > \text{MaxIter}$ and matrix is not diagonally dominant.
  2: $\text{Iter} > \text{MaxIter}$ and matrix is diagonally dominant.
  3: $\text{Dimen} < 1$.
  4: $\text{Tol} \leq 0$.
  5: $\text{MaxIter} < 0$.
  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,
   
   \[
   [1, 1], [1, 2], \ldots, [1, N], [2, 1], \ldots, [2, N], \ldots, [N, N],
   \]
   where \(N\) is the dimension of the matrix
3. The elements of the first constant vector, in the order \([1], \ldots, [N]\)
For example, to solve the systems

\[
\begin{align*}
20x + 3y &= 10 \\
-4y &= 10
\end{align*}
\]

a text file could be created to look like this:

\[
\begin{array}{ccc}
2 & 20 & 3 \\
0 & -4 & 10 \\
10 & 10
\end{array}
\]

**Example**

**Problem.** Solve the following linear system to within a tolerance of \(1E - 12\):

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

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:**

\[
\begin{array}{cccccccc}
10.000000000 & 1.000000000 & 2.000000000 & -3.000000000 & 2.000000000 \\
4.000000000 & 50.000000000 & 1.000000000 & 0.000000000 & 1.000000000 \\
-2.000000000 & 5.000000000 & -30.000000000 & 1.000000000 & 1.000000000 \\
6.000000000 & 4.000000000 & 0.000000000 & 10.000000000 & 3.000000000 \\
\end{array}
\]

**The constants:**

\[
\begin{array}{c}
-2.90000000000000e+1 \\
3.50000000000000e+1 \\
-2.50000000000000e+1 \\
-4.60000000000000e+1 \\
-1.06000000000000e+2
\end{array}
\]
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
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 $Aw = w$. 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**

```plaintext
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. $\text{Dimen} > 1$
2. $\text{Dimen} \leq \text{TNArraySize}$
3. $\text{Tolerance} > 0$
4. $\text{MaxIter} > 0$

$\text{TNArraySize}$ fixes an upper bound on the number of elements in each vector. It is used in the type definition of $\text{TNvector}$ and $\text{TNmatrix}$. $\text{TNArraySize}$ 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**: $\text{TNvector}$; Approximate eigenvector associated with the dominant eigenvalue
- **Iter**: Integer; Number of iterations required to find the solution
- **Error**: Byte; 
  0: No errors
  1: $\text{Dimen} \leq 1$
  2: $\text{Tolerance} \leq 0$
  3: $\text{MaxIter} \leq 0$
  4: $\text{Iter} \geq \text{MaxIter}$

**Syntax of the Procedure Call**

```
Power(Dimen, Mat, GuessVector, MaxIter, Tolerance, 
      Eigenvalue, Eigenvector, Iter, Error);
```
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
   
   \[
   \begin{bmatrix}
   1 & 1 \\
   1 & 2 \\
   \vdots & \vdots \\
   1 & N \\
   N & 1 \\
   \end{bmatrix},
   \] 
   
   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:

\[
\begin{bmatrix}
2 & 1 & 0 \\
0 & 1 & 0 \\
0 & 2 & 4 \\
\end{bmatrix}
\]

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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

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 $(A - pl)w = v$ (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. $\text{Dimen} > 1$
2. $\text{Dimen} \leq \text{TNArraySize}$
3. $\text{Tolerance} > 0$
4. $\text{MaxIter} > 0$

$\text{TNArraySize}$ sets an upper bound on the number of elements in each vector. It is used in the type definition of $\text{TNvector}$ and $\text{TNmatrix}$. $\text{TNArraySize}$ 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

- $\text{Eigenvalue}$ : Extended; Approximation to the eigenvalue closest to $\text{ClosestVal}$
- $\text{Eigenvector}$ : $\text{TNvector}$; Approximation to the eigenvector associated with $\text{Eigenvalue}$
- $\text{Iter}$ : Integer; Number of iterations required to find the solution
- $\text{Error}$ : Byte;
  - 0: No errors
  - 1: $\text{Dimen} \leq 1$
  - 2: $\text{Tolerance} \leq 0$
  - 3: $\text{MaxIter} \leq 0$
  - 4: $\text{Iter} \geq \text{MaxIter}$
  - 5: $\text{Eigenvalue/Eigenvector}$ not calculated (see “Comments”)

Syntax of the Procedure Call

```pascal
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 \((\text{Mat} - \text{Eigenvalue} \times I)\) 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, \(\text{ClosestVal}\).

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 \((\text{ClosestVal})\). If \(\text{ClosestVal}\) 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

\[
\begin{pmatrix}
1 & 2 \\
3 & 4
\end{pmatrix}
\]

with an initial guess of (11, 10), you could first create the following text file:

\[
\begin{pmatrix}
4 \\
1 \\
2 \\
3 \\
4 \\
11 \\
10
\end{pmatrix}
\]

**Example**

**Problem.** Suppose you know that two of the eigenvalues of the matrix

\[
\begin{pmatrix}
2 & 1 & 0 \\
0 & 1 & 0 \\
0 & 2 & 4
\end{pmatrix}
\]

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:
\[
\begin{pmatrix}
2.00000000000000e+0 & 1.00000000000000e+1 & 0.00000000000000e+0 \\
0.00000000000000e+0 & 1.00000000000000e+0 & 0.00000000000000e+0 \\
0.00000000000000e+0 & 2.00000000000000e+0 & 4.00000000000000e+0
\end{pmatrix}
\]

Approximate eigenvalue: 1.99900000000000e+0
Tolerance: 1.00000000000000e-8
Maximum number of iterations: 200

Number of iterations: 4
The approximate eigenvector:
\[
\begin{pmatrix}
1.00000000000000e+0 \\
9.56200019081920e-14 \\
-5.08756039829010e-14
\end{pmatrix}
\]

The associated eigenvalue: 2.00000000000960e+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:
\[
\begin{array}{ccc}
2.00000000000000e+0 & 1.00000000000000e+1 & 0.00000000000000e+0 \\
0.00000000000000e+0 & 1.00000000000000e+0 & 0.00000000000000e+0 \\
0.00000000000000e+0 & 2.00000000000000e+0 & 4.00000000000000e+0 \\
\end{array}
\]

Approximate eigenvalue: \( 7.00000000000000e-1 \)
Tolerance: \( 1.00000000000000e-8 \)
Maximum number of iterations: 200

Number of iterations: 12
The approximate eigenvector:
\[
\begin{array}{c}
1.00000000000000e+0 \\
-1.00000002395103e-1 \\
6.66666682633328e-2 \\
\end{array}
\]

The associated eigenvalue: \( 9.99999976048973e-1 \)

The exact solutions are
\[
\begin{align*}
\text{Eigenvalue} &= 2; \quad \text{Eigenvector} = (1, 0, 0) \\
\text{Eigenvalue} &= 1; \quad \text{Eigenvector} = (1, -0.1, 2/30)
\end{align*}
\]
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

\[
\begin{align*}
\text{TNvector} & = \text{array}[1..\text{TNArraySize}] \text{ of Extended;} \\
\text{TNmatrix} & = \text{array}[1..\text{TNArraySize}] \text{ of TNvector;} \\
\text{TNIntVector} & = \text{array}[1..\text{TNArraySize}] \text{ of Integer;} \\
\end{align*}
\]

Input Parameters

\[
\begin{align*}
\text{Dimen} : \text{Integer} & \quad \text{Dimension of the matrix } \text{Mat} \\
\text{Mat} : \text{TNmatrix} & \quad \text{The matrix} \\
\text{GuessVector} : \text{TNvector} & \quad \text{Initial approximation (Guess) of an eigenvector}
\end{align*}
\]
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 : TNIintVector;  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 $20 \times 20$ 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 $2 \times 2$, it is easy to determine if the eigenvalues of the $2 \times 2$ 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

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

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

\[
\begin{align*}
\text{Matrix}[1, 1] & : 2 \\
\text{Matrix}[1, 2] & : 10 \\
\text{Matrix}[1, 3] & : 0 \\
\text{Matrix}[2, 1] & : 0 \\
\text{Matrix}[2, 2] & : 1 \\
\text{Matrix}[2, 3] & : 0 \\
\text{Matrix}[3, 1] & : 0 \\
\text{Matrix}[3, 2] & : 2 \\
\text{Matrix}[3, 3] & : 4
\end{align*}
\]
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:
\[
\begin{array}{ccc}
2.00000000000000 & 1.00000000000000 & 0.00000000000000 \\
0.00000000000000 & 1.00000000000000 & 0.00000000000000 \\
0.00000000000000 & 2.00000000000000 & 4.00000000000000 \\
\end{array}
\]

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
-9.99999999999991e-2
6.66666592646069e-2

The associated eigenvalue: 9.99999999999991e-1

The exact solution is

Eigenvalue = 4; Eigenvector = (0, 0, 1)
Eigenvalue = 2; Eigenvector = (1, 0, 0)
Eigenvalue = 1; Eigenvector = (1, -0.1, 2/30)
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

\[
\text{TNvector} = \text{array}[1..\text{TNArraySize}] \text{ of Extended;}
\]
\[
\text{TNmatrix} = \text{array}[1..\text{TNArraySize}] \text{ of TNvector;}
\]

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. $\text{Dimen} > 1$.
2. $\text{Dimen} \leq \text{TNArraySize}$.
3. $\text{Tolerance} > 0$.
4. $\text{MaxIter} > 0$.
5. $\text{Mat}$ must be symmetric.

$\text{TNArraySize}$ sets an upper bound on the number of elements in each vector. It is used in the type definition of $\text{TNvector}$ and $\text{TNmatrix}$. $\text{TNArraySize}$ 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**: $\text{TNvector}$; Approximation to the eigenvalues of the matrix
- **Eigenvectors**: $\text{TNmatrix}$; Approximation to the eigenvectors associated with the eigenvalues
- **Iter**: $\text{Integer}$; Number of iterations required to find eigenvalues/vectors
- **Error**: $\text{Byte}$; 0: No errors 1: $\text{Dimen} \leq 1$ 2: $\text{Tolerance} \leq 0$ 3: $\text{MaxIter} \leq 0$ 4: $\text{Mat}$ not symmetric 5: $\text{Iter} \geq \text{MaxIter}$

**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
   \[1, 1, \ldots, [1, 2, \ldots, [1, N], \ldots, [N, 1], \ldots, [N, N],\]
   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:

\[
\begin{bmatrix}
1.000000000 & 2.000000000 & -3.000000000 & -1.000000000 \\
2.000000000 & 1.000000000 & -1.000000000 & -3.000000000 \\
-3.000000000 & -1.000000000 & 1.000000000 & 2.000000000 \\
-1.000000000 & -3.000000000 & 2.000000000 & 1.000000000
\end{bmatrix}
\]

Tolerance: 1.000000000000000e-8
Maximum number of iterations: 200
Number of iterations: 4

The approximate eigenvector:
-1.000000000000000e+0
-1.000000000000000e+0
1.000000000000000e+0
1.000000000000000e+0

The associated eigenvalue: 7.000000000000000e+0

The approximate eigenvector:
9.9999999997805e-1
-9.9999999997804e-1
-1.000000000000000e+0
1.000000000000000e+0

The associated eigenvalue: 1.000000000000000e+0
The approximate eigenvector:
1.00000000000000e+0
-9.99999556935429e-1
9.99999999999999e-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: -1.00000000000000e+0

The exact solution is

Eigenvalue =  7; Eigenvector = (1, 1, -1, -1)
Eigenvalue = -3; Eigenvector = (1, -1, 1, -1)
Eigenvalue =  1; Eigenvector = (-1, 1, 1, -1)
Eigenvalue = -1; Eigenvector = (1, 1, 1, 1)
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

\[
\begin{align*}
    f(0) &= 3 \\
    f'(0) &= 4
\end{align*}
\]

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 \( n \)th-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 involv-
ing 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 infor-
mation obtained from the preceding mesh point. The Runge-Kutta method em-
ploys 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 trunca-
tion 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 approx-
imate the value of the dependent variable at the current mesh point. The proce-
dure 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 equa-
tion 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 **Lipschitz condition**. The Lipschitz 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 Lipschitz condition. For a second-order differential equation, the Lipschitz condition states that there exists a positive number \(L\) such that

\[
|f(t, x_2, x'_2) - f(t, x_1, x'_1)| \leq L(|x_2 - x_1| + |x'_2 - x'_1|)
\]

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 \(n\)th-order differential equation with the fourth-order Runge-Kutta formulas. The Lipschitz 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 Lipschitz 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
\[ \frac{dx}{dt} = TNTargetF(t, x) \]
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:

\[
egin{align*}
F1 &= h \cdot TNTargetF(t, x[t]) \\
F2 &= h \cdot TNTargetF(t + h/2, x[t] + F1/2) \\
F3 &= h \cdot TNTargetF(t + h/2, x[t] + F2/2) \\
F4 &= h \cdot TNTargetF(t + h, x[t] + F3) \\
x[t + 1] &= x[t] + (F1 + 2 \cdot F2 + 2 \cdot F3 + F4)/6
\end{align*}
\]

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

\[ \text{TNvector} = \text{array}[1..TNArraySize] \text{ of Extended;} \]
User-Defined Function

\[ \frac{dx}{dt} = TNTargetF(t, x) \]

The function \( TNTargetF(t, x) \) is a user-defined function that calculates the derivative \( \frac{dx}{dt} \).

Input Parameters

\begin{itemize}
  \item LowerLimit: Extended; Lower limit of interval
  \item UpperLimit: Extended; Upper limit of interval
  \item XInitial: Extended; Value of \( X \) at \( \text{LowerLimit} \)
  \item NumReturn: Integer; Number of \((t, x)\) pairs returned from the procedure
  \item NumIntervals: Integer; Number of subintervals used in calculations
\end{itemize}

The preceding parameters must satisfy the following conditions:

1. \( \text{NumReturn} > 0 \)
2. \( \text{NumIntervals} \geq \text{NumReturn} \)
3. \( \text{LowerLimit} \neq \text{UpperLimit} \)

Output Parameters

\begin{itemize}
  \item TValues: TNvector; Values of \( t \) between the limits
  \item XValues: TNvector; Values of \( X \) approximated at the values in \( \text{TValues} \)
  \item Error: Byte;
    0: No errors
    1: \( \text{NumReturn} < 1 \)
    2: \( \text{NumIntervals} < \text{NumReturn} \)
    3: \( \text{LowerLimit} = \text{UpperLimit} \)
\end{itemize}

Syntax of the Procedure Call

\begin{verbatim}
InitialCondition1stOrder(LowerLimit, UpperLimit, XInitial, NumReturn,
  NumIntervals, TValues, XValues, Error, @TNTargetF);
\end{verbatim}

The procedure \( \text{InitialCondition1stOrder} \) 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, \( y = x + e^{-100t} \). 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:

\[
\begin{align*}
    x' &= x/t + t - 1, \quad 1 \leq t \leq 2 \\
    x(1) &= 1
\end{align*}
\]

1. Code the equation into the program Runge_1.pas:

```pascal
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 (\( \geq 10 \))? 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

<table>
<thead>
<tr>
<th>( t )</th>
<th>( X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000</td>
<td>1.000000000000000e+0</td>
</tr>
<tr>
<td>1.1000</td>
<td>1.10515880220649e+0</td>
</tr>
<tr>
<td>1.2000</td>
<td>1.22121413182916e+0</td>
</tr>
<tr>
<td>1.3000</td>
<td>1.34892645616477e+0</td>
</tr>
<tr>
<td>1.4000</td>
<td>1.48893866869362e+0</td>
</tr>
<tr>
<td>1.5000</td>
<td>1.64180233779216e+0</td>
</tr>
<tr>
<td>1.6000</td>
<td>1.80799419315265e+0</td>
</tr>
<tr>
<td>1.7000</td>
<td>1.98793197313186e+0</td>
</tr>
<tr>
<td>1.8000</td>
<td>2.18198400310574e+0</td>
</tr>
<tr>
<td>1.9000</td>
<td>2.39047761619428e+0</td>
</tr>
<tr>
<td>2.0000</td>
<td>2.61370563879444e+0</td>
</tr>
</tbody>
</table>

The exact solution is

   \[ X = t^2 - t \cdot \log(t) \]
   \[ X(2) = 2.6137056 \]
Solution to an Initial Value Problem for a First-Order Ordinary Differential Equation Using the Runge-Kutta-Fehlberg Method (RKF_I.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_I.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** \(\leq 0\)
  2: **NumReturn** \(\leq 0\)
  3: **LowerLimit** = **UpperLimit**
  4: **Tolerance** not reached

**Syntax of the Procedure Call**

```pascal
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, \( y = x + e^{-100t} \). 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:

\[
x' = \frac{x}{t} + t - 1 \quad 1 \leq t \leq 2 \\
x(1) = 1
\]
1. Code the differential equation into the program RKF_1.pas:

```pascal
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 = 1.00000000: 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

<table>
<thead>
<tr>
<th>t</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00000000</td>
<td>1.00000000000000E+0</td>
</tr>
<tr>
<td>1.10000000</td>
<td>1.10515881708653E+0</td>
</tr>
<tr>
<td>1.20000000</td>
<td>1.22121416069278E+0</td>
</tr>
<tr>
<td>1.30000000</td>
<td>1.34892649817459E+0</td>
</tr>
<tr>
<td>1.40000000</td>
<td>1.489893892310351E+0</td>
</tr>
<tr>
<td>1.50000000</td>
<td>1.64180240395245E+0</td>
</tr>
<tr>
<td>1.60000000</td>
<td>1.80799427050390E+0</td>
</tr>
<tr>
<td>1.70000000</td>
<td>1.98793206119471e+0</td>
</tr>
<tr>
<td>1.80000000</td>
<td>2.18198410146987e+0</td>
</tr>
<tr>
<td>1.90000000</td>
<td>2.39047772450816e+0</td>
</tr>
<tr>
<td>2.00000000</td>
<td>2.61370575675625e+0</td>
</tr>
</tbody>
</table>

   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 = 1.00000000: 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.

<table>
<thead>
<tr>
<th>T</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>1.1220</td>
<td>1.1298</td>
</tr>
<tr>
<td>1.2058</td>
<td>1.2283</td>
</tr>
<tr>
<td>1.2927</td>
<td>1.3392</td>
</tr>
<tr>
<td>1.3829</td>
<td>1.4640</td>
</tr>
<tr>
<td>1.4764</td>
<td>1.6046</td>
</tr>
<tr>
<td>1.5737</td>
<td>1.7630</td>
</tr>
<tr>
<td>1.6747</td>
<td>1.9412</td>
</tr>
<tr>
<td>1.7797</td>
<td>2.1414</td>
</tr>
<tr>
<td>1.8887</td>
<td>2.3663</td>
</tr>
<tr>
<td>2.0019</td>
<td>2.6182</td>
</tr>
</tbody>
</table>

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

$$\frac{dx}{dt} = TNTargetF(t, x)$$

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:

$$x_o[i+1] = x[i] + h/24 \times \{ 55 \times TNTargetF(t[i], x[i])$$
- $$59 \times TNTargetF(t[i-1], x[i-1])$$
+ $$37 \times TNTargetF(t[i-2], x[i-2])$$
- $$9 \times TNTargetF(t[i-3], x[i-3]) \}$$

and the following implicit Adams-Moulton formula:

$$x[i+1] = x[i] + h/24 \times \{ 9 \times TNTargetF(t[i+1], x_o[i+1])$$
+ $$19 \times TNTargetF(t[i], x[i])$$
- $$5 \times TNTargetF(t[i-1], x[i-1])$$
+ $$TNTargetF(t[i-2], x[i-2]) \}$$

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. \( \text{NumReturn} > 0 \)
2. \( \text{NumIntervals} \geq \text{NumReturn} \)
3. \( \text{LowerLimit} \neq \text{UpperLimit} \)

**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: \( \text{NumReturn} < 1 \)  
2: \( \text{NumIntervals} < \text{NumReturn} \)  
3: \( \text{LowerLimit} = \text{UpperLimit} \)
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:

\[
\begin{align*}
    x' &= x/t + t - 1 & 1 \leq t \leq 2 \\
    x(1) &= 1
\end{align*}
\]
1. Code the differential equation into the program Adams_1.pas:

```pascal
function TNTargetF(t, X : Extended) : Extended;

{----------------------------------------------------------------------}
{--- THIS IS THE FIRST-ORDER DIFFERENTIAL EQUATION ---}
{----------------------------------------------------------------------}

begin
    TNTargetF := x/t + t - 1;
end;
```

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

<table>
<thead>
<tr>
<th>t</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000000</td>
<td>1.00000000000000e+0</td>
</tr>
<tr>
<td>1.1000000</td>
<td>1.10515880229293e+0</td>
</tr>
<tr>
<td>1.2000000</td>
<td>1.22121413201736e+0</td>
</tr>
<tr>
<td>1.3000000</td>
<td>1.34892645643801e+0</td>
</tr>
<tr>
<td>1.4000000</td>
<td>1.4889386904034e+0</td>
</tr>
<tr>
<td>1.5000000</td>
<td>1.64180233820416e+0</td>
</tr>
<tr>
<td>1.6000000</td>
<td>1.80799419362396e+0</td>
</tr>
<tr>
<td>1.7000000</td>
<td>1.98793197365806e+0</td>
</tr>
<tr>
<td>1.8000000</td>
<td>2.1819840368348e+0</td>
</tr>
<tr>
<td>1.9000000</td>
<td>2.39047761682098e+0</td>
</tr>
<tr>
<td>2.0000000</td>
<td>2.61370563946811e+0</td>
</tr>
</tbody>
</table>

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
\[ \frac{d^2x}{dt^2} = T\text{NTargetF}(t, x, x') \]
where \( x' \) indicates \( dx/dt \) (which satisfies the Lipshitz condition given at the beginning of this chapter), the initial conditions
\[
\begin{align*}
x[\text{LowerLimit}] &= \text{InitialValue} \\
x'[\text{LowerLimit}] &= \text{InitialDeriv}
\end{align*}
\]
and spacing
\[ h = (\text{UpperLimit} - \text{LowerLimit})/\text{NumIntervals} \]
rewrite the second-order differential equation as two, first-order differential equations:
\[
\begin{align*}
x' &= y \\
y' &= T\text{NTargetF}(t, x, y)
\end{align*}
\]
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:
\[
\begin{align*}
F1x &= h \ast y[t] \\
F1y &= h \ast T\text{NTargetF}(t, x[t], y[t]) \\
F2x &= h \ast (y[t] + F1y/2) \\
F2y &= h \ast T\text{NTargetF}(t + h/2, x[t] + F1x/2, y[t] + F1y/2) \\
F3x &= h \ast (y[t] + F2y/2) \\
F3y &= h \ast T\text{NTargetF}(t + h/2, x[t] + F2x/2, y[t] + F2y/2) \\
F4x &= h \ast (y[t] + F3y) \\
F4y &= h \ast T\text{NTargetF}(t + h, x[t] + F3x, y[t] + F3y)
\end{align*}
\]
\[ x[t+1] = x[t] + (F1x + 2 * F2x + 2 * F3x + F4x)/6 \]
\[ y[t+1] = y[t] + (F1y + 2 * F2y + 2 * F3y + F4y)/6 \]

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

InitialCondition2ndOrder(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 $F \sin(\omega t)$ ($\omega$ represents the frequency of the driving force and $t$ is time), the equation of motion of the mass is as follows:

$$m \frac{d^2x}{dt^2} + kx = F \sin(\omega t)$$

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:

$$\frac{d^2x}{dt^2} = \frac{F}{m} \sin(\omega t) - \frac{k}{m} x$$
2. Code this second-order differential equation into the program Runge-2.pas:

```pascal
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 = 0.000000e+0: 0
- Enter derivative of X at t = 0.000000e+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.

<table>
<thead>
<tr>
<th>T</th>
<th>Value of X</th>
<th>Derivative of X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000000</td>
<td>0.00000000</td>
<td>-2.50000000</td>
</tr>
<tr>
<td>0.2000000</td>
<td>-4.2073528</td>
<td>-1.3507564</td>
</tr>
<tr>
<td>0.4000000</td>
<td>-4.5464872</td>
<td>1.0403653</td>
</tr>
<tr>
<td>0.6000000</td>
<td>-7.0560578</td>
<td>2.4749799</td>
</tr>
<tr>
<td>0.8000000</td>
<td>3.7840037</td>
<td>1.6341103</td>
</tr>
<tr>
<td>1.0000000</td>
<td>4.7946176</td>
<td>-7.0915129</td>
</tr>
<tr>
<td>1.2000000</td>
<td>1.3970846</td>
<td>-2.4004215</td>
</tr>
<tr>
<td>1.4000000</td>
<td>-3.2849179</td>
<td>-1.8847552</td>
</tr>
<tr>
<td>1.6000000</td>
<td>-4.9477947</td>
<td>3.6374522</td>
</tr>
<tr>
<td>1.8000000</td>
<td>-2.0605951</td>
<td>2.2778186</td>
</tr>
<tr>
<td>2.0000000</td>
<td>2.7200842</td>
<td>2.0976751</td>
</tr>
</tbody>
</table>
The exact solution is

\[ x = \frac{F \sin(\omega t)}{m (\omega_0^2 - \omega^2)} \]

\[ \frac{dx}{dt} = \frac{F_0 \cos(\omega t)}{m (\omega_0^2 - \omega^2)} \]

where \( \omega_0 \) is the natural frequency of the system

\[ \omega_0^2 = \frac{k}{m} \]

The period of oscillation is given by

\[ t = \frac{2 \pi}{\omega} = 1.257 \text{ sec} \]

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

<table>
<thead>
<tr>
<th>t</th>
<th>Values of X</th>
<th>Derivative of X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0000000000000E + 000</td>
<td>-2.500000000000E + 000</td>
</tr>
<tr>
<td>0.2</td>
<td>-4.207354924039E - 001</td>
<td>-1.350755764670E + 000</td>
</tr>
<tr>
<td>0.4</td>
<td>-4.546487134128E - 001</td>
<td>1.040367091367E + 000</td>
</tr>
<tr>
<td>0.6</td>
<td>-7.05600402993E - 002</td>
<td>2.474981241501E + 000</td>
</tr>
<tr>
<td>0.8</td>
<td>3.784012476539E - 001</td>
<td>1.634109052159E + 000</td>
</tr>
<tr>
<td>1.0</td>
<td>4.794621373315E - 001</td>
<td>-7.091554636580E - 001</td>
</tr>
<tr>
<td>1.2</td>
<td>1.397077490994E - 001</td>
<td>-2.400425716625E + 000</td>
</tr>
<tr>
<td>1.4</td>
<td>-3.284932993593E - 001</td>
<td>-1.884755635858E + 000</td>
</tr>
<tr>
<td>1.6</td>
<td>-4.946791233116E - 001</td>
<td>3.637500845215E - 001</td>
</tr>
<tr>
<td>1.8</td>
<td>-2.60592426208E - 001</td>
<td>2.277825654711E + 000</td>
</tr>
<tr>
<td>2.0</td>
<td>2.720105554446E - 001</td>
<td>2.097678822691E + 000</td>
</tr>
</tbody>
</table>
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 nth-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
\[ d^n x/dt^n = TNTargetF(t, x, x^{(1)}, ..., x^{(n-1)}) \]

where \( x^{(0)} \) indicates \( d^n x/dt^n \), 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

\[ x[\text{LowerLimit}] = a_1 \]
\[ x^{(1)}[\text{LowerLimit}] = a_2 \]
\[ \vdots \]
\[ x^{(n-1)}[\text{LowerLimit}] = a_n \]

and spacing

\[ h = (\text{UpperLimit} - \text{LowerLimit})/\text{NumIntervals} \]

rewrite the nth-order differential equation as \( n \) first-order differential equations:

\[ x^{(1)} = y_1 \]
\[ x^{(2)} = y^{(1)}_1 = y_2 \]
\[ x^{(3)} = y^{(1)}_2 = y_3 \]
\[ \vdots \]
\[ x^{(n-1)} = y^{(1)}_{n-2} = y_{n-1} \]
\[ x^{(n)} = y^{(1)}_{n-1} = TNTargetF(t, x, y_1, y_2, ..., y_{n-1}) \]

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:

\[
F1x = h \cdot y_1[t]
\]
\[
F1y_1 = h \cdot y_2[t]
\]
\[
F1y_{n-2} = h \cdot y_{n-1}[t]
\]
\[
F1y_{n-1} = h \cdot \text{TNTargetF}(t, x[t], y_1[t], ..., y_{n-1}[t])
\]
\[
F2x = h \cdot (y_1[t] + F1y_1/2)
\]
\[
F2y_1 = h \cdot (y_2[t] + F1y_2/2)
\]
\[
F2y_{n-2} = h \cdot (y_{n-1}[t] + F1y_{n-1}/2)
\]
\[
F2y_{n-1} = h \cdot \text{TNTargetF}(t + h/2, x[t] + F1x/2, y_1[t] + F1y_1/2, ..., y_{n-1}[t] + F1y_{n-1}/2)
\]
\[
F3x = h \cdot (y_1[t] + F2y_1)
\]
\[
F3y_1 = h \cdot (y_2[t] + F2y_2)
\]
\[
F3y_{n-2} = h \cdot (y_{n-1}[t] + F2y_{n-1}/2)
\]
\[
F3y_{n-1} = h \cdot \text{TNTargetF}(t + h/2, x[t] + F2x/2, y_1[t] + F2y_1/2, ..., y_{n-1}[t] + F2y_{n-1}/2)
\]
\[
F4x = h \cdot (y_1[t] + F3y_1)
\]
\[
F4y_1 = h \cdot (y_2[t] + F3y_2)
\]
\[
F4y_{n-2} = h \cdot (y_{n-1}[t] + F3y_{n-1})
\]
\[
F4y_{n-1} = h \cdot \text{TNTargetF}(t + h, x[t] + F3x, y_1[t] + F3y_1, ..., y_{n-1}[t] + F3y_{n-1})
\]
\[
x[t+1] = x[t] + (F1x + 2 \cdot F2x + 2 \cdot F3x + F4x)/6
\]
\[
y_1[t+1] = y_1[t] + (F1y_1 + 2 \cdot F2y_1 + 2 \cdot F3y_1 + F4y_1)/6
\]
\[
y_2[t+1] = y_2[t] + (F1y_2 + 2 \cdot F2y_2 + 2 \cdot F3y_2 + F4y_2)/6
\]
\[
y_{n-2}[t+1] = y_{n-2}[t] + (F1y_{n-2} + 2 \cdot F2y_{n-2} + 2 \cdot F3y_{n-2} + F4y_{n-2})/6
\]
\[
y_{n-1}[t+1] = y_{n-1}[t] + (F1y_{n-1} + 2 \cdot F2y_{n-1} + 2 \cdot F3y_{n-1} + F4y_{n-1})/6
\]

where \( t \) ranges from \( \text{LowerLimit} \) to \( \text{UpperLimit} \) 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**

\[
TNvector = \text{array}[0..TNRowSize] \text{ of } \text{Extended};
\]

\[
TNmatrix = \text{array}[0..TNColumnSize] \text{ of } TNvector;
\]

\( TNRowSize \) 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**

\[
TNTargetF(V : TNvector) : \text{Extended};
\]

The elements of \( V \) are defined as

\[
V[0] \text{ corresponds to } t
\]
\[
V[1] \text{ corresponds to } x
\]
\[
V[2] \text{ corresponds to first derivative of } x
\]
\[
V[3] \text{ corresponds to second derivative of } x
\]

\[\vdots\]

This is the differential equation:

\[
d^n x/dt^n = TNTargetF(t, x, x^{(1)}, \ldots, x^{(n-1)}) \text{ where } n \text{ is the order of the equation.}
\]

The procedure \textit{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 \textit{LowerLimit}
- **NumReturn**: Integer; Number of \((t, x, x^{(1)}, \ldots, x^{(n)})\) 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 ≥ NumReturn
3. Order > 0
4. LowerLimit ≠ 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<sup>1</sup> 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, \( y = x + e^{-100t} \). The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differen-
tial 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 \( t = 1 \):

\[
\frac{d^4x(t)}{dt^4} = -4x(t) \frac{d^3x(t)}{dt^3}
\]

\[
x(0) = 1 \\
dx(0)/dt = -1 \\
d^2x(0)/dt^2 = 2 \\
d^3x(0)/dt^3 = -6
\]
1. Code the equation into the program Runge_JJ.pas:

   function TNTargetF(V: TNvector): Extended;
   begin
     TNTargetF := -4 * V[1] * V[4];
   end;

2. Run Runge_JJ.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.
<table>
<thead>
<tr>
<th>t</th>
<th>Value X[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.00000000000000e+00</td>
</tr>
<tr>
<td>0.01</td>
<td>9.09090909737517e-01</td>
</tr>
<tr>
<td>0.20</td>
<td>8.33333334189336e-01</td>
</tr>
<tr>
<td>0.30</td>
<td>7.69230770157394e-01</td>
</tr>
<tr>
<td>0.40</td>
<td>7.14285715280102e-01</td>
</tr>
<tr>
<td>0.50</td>
<td>6.66666667788519e-01</td>
</tr>
<tr>
<td>0.60</td>
<td>6.25000001337168e-01</td>
</tr>
<tr>
<td>0.70</td>
<td>5.88235295769619e-01</td>
</tr>
<tr>
<td>0.80</td>
<td>5.55555557625526e-01</td>
</tr>
<tr>
<td>0.90</td>
<td>5.26315792064849e-01</td>
</tr>
<tr>
<td>1.00</td>
<td>5.00000000321398e-01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t</th>
<th>Value X[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>-1.00000000000000e+00</td>
</tr>
<tr>
<td>0.10</td>
<td>-8.26446283273189e-01</td>
</tr>
<tr>
<td>0.20</td>
<td>-6.9444446826215e-01</td>
</tr>
<tr>
<td>0.30</td>
<td>-5.9171597923112e-01</td>
</tr>
<tr>
<td>0.40</td>
<td>-5.10204082090465e-01</td>
</tr>
<tr>
<td>0.50</td>
<td>-4.4444443661452e-01</td>
</tr>
<tr>
<td>0.60</td>
<td>-3.90624997971428e-01</td>
</tr>
<tr>
<td>0.70</td>
<td>-3.46020758007956e-01</td>
</tr>
<tr>
<td>0.80</td>
<td>-3.08641970911504e-01</td>
</tr>
<tr>
<td>0.90</td>
<td>-2.7708304743045e-01</td>
</tr>
<tr>
<td>1.00</td>
<td>-2.49999993429933e-01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t</th>
<th>Value X[3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>2.00000000000000e+00</td>
</tr>
<tr>
<td>0.10</td>
<td>1.50262961438149e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>1.15740742373768e+00</td>
</tr>
<tr>
<td>0.30</td>
<td>9.10332288053840e-01</td>
</tr>
<tr>
<td>0.40</td>
<td>7.2862989793594e-01</td>
</tr>
<tr>
<td>0.50</td>
<td>5.92592607536866e-01</td>
</tr>
<tr>
<td>0.60</td>
<td>4.88281263842229e-01</td>
</tr>
<tr>
<td>0.70</td>
<td>4.07083261374879e-01</td>
</tr>
<tr>
<td>0.80</td>
<td>3.42935540127152e-01</td>
</tr>
<tr>
<td>0.90</td>
<td>2.91587706310718e-01</td>
</tr>
<tr>
<td>1.00</td>
<td>2.50000001075353e-01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t</th>
<th>Value X[4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>-6.00000000000000e+00</td>
</tr>
<tr>
<td>0.10</td>
<td>-4.09080760565627e+00</td>
</tr>
<tr>
<td>0.20</td>
<td>-2.89351855059016e+00</td>
</tr>
<tr>
<td>0.30</td>
<td>-2.10076680857258e+00</td>
</tr>
<tr>
<td>0.40</td>
<td>-1.56184925333600e+00</td>
</tr>
<tr>
<td>0.50</td>
<td>-1.185185204443061e+00</td>
</tr>
<tr>
<td>0.60</td>
<td>-0.9357390788898e+00</td>
</tr>
<tr>
<td>0.70</td>
<td>-0.7138221540418e-01</td>
</tr>
<tr>
<td>0.80</td>
<td>-0.571559223064178e-01</td>
</tr>
<tr>
<td>0.90</td>
<td>-0.460401631119694e-01</td>
</tr>
<tr>
<td>1.00</td>
<td>-0.37500005740566e-01</td>
</tr>
</tbody>
</table>
$X[1]$ 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

\[
x(t) = (t+1)^{-1}
\]
\[
dx(t)/dt = -(t+1)^{-2}
\]
\[
d^2x(t)/dt^2 = 2(t+1)^{-3}
\]
\[
d^3x(t)/dt^3 = -6(t+1)^{-4}
\]

\[
x(1) = 0.5
\]
\[
dx(1)/dt = -0.25
\]
\[
d^2x(1)/dt^2 = 0.25
\]
\[
d^3x(1)/dt^3 = -0.375
\]
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

$$\frac{dx_1}{dt} = TNTargetF1(t, x_1, x_2, ..., x_m)$$
$$\frac{dx_2}{dt} = TNTargetF2(t, x_1, x_2, ..., x_m)$$
.$$...
.$$...
.$$\frac{dx_m}{dt} = TNTargetFm(t, x_1, x_2, ..., x_m)$$

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

$$x_1[LowerLimit] = a_1$$
$$x_2[LowerLimit] = a_2$$
.$$...
.$$...
.$$x_m[LowerLimit] = a_m$$

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:

\[ F_{1x_1} = h \cdot TNSourceFunction(t, x_1[t], x_2[t], \ldots, x_m[t]) \]
\[ F_{1x_2} = h \cdot TNSourceFunction(t, x_1[t], x_2[t], \ldots, x_m[t]) \]
\[ \quad \vdots \]
\[ F_{1x_m} = h \cdot TNSourceFunction(t, x_1[t], x_2[t], \ldots, x_m[t]) \]
\[ F_{2x_1} = h \cdot TNSourceFunction(t + h/2, x_1[t] + F_{1x_1}/2, x_2[t] + F_{1x_2}/2, \ldots, x_m[t] + F_{1x_m}/2) \]
\[ F_{2x_2} = h \cdot TNSourceFunction(t + h/2, x_1[t] + F_{1x_1}/2, x_2[t] + F_{1x_2}/2, \ldots, x_m[t] + F_{1x_m}/2) \]
\[ \quad \vdots \]
\[ F_{2x_m} = h \cdot TNSourceFunction(t + h/2, x_1[t] + F_{1x_1}/2, x_2[t] + F_{1x_2}/2, \ldots, x_m[t] + F_{1x_m}/2) \]
\[ F_{3x_1} = h \cdot TNSourceFunction(t + h/2, x_1[t] + F_{2x_1}/2, x_2[t] + F_{2x_2}/2, \ldots, x_m[t] + F_{2x_m}/2) \]
\[ F_{3x_2} = h \cdot TNSourceFunction(t + h/2, x_1[t] + F_{2x_1}/2, x_2[t] + F_{2x_2}/2, \ldots, x_m[t] + F_{2x_m}/2) \]
\[ \quad \vdots \]
\[ F_{3x_m} = h \cdot TNSourceFunction(t + h/2, x_1[t] + F_{2x_1}/2, x_2[t] + F_{2x_2}/2, \ldots, x_m[t] + F_{2x_m}/2) \]
\[ F_{4x_1} = h \cdot TNSourceFunction(t + h, x_1[t] + F_{3x_1}, x_2[t] + F_{3x_2}, \ldots, x_m[t] + F_{3x_m}) \]
\[ F_{4x_2} = h \cdot TNSourceFunction(t + h, x_1[t] + F_{3x_1}, x_2[t] + F_{3x_2}, \ldots, x_m[t] + F_{3x_m}) \]
\[ \quad \vdots \]
\[ F_{4x_m} = h \cdot TNSourceFunction(t + h, x_1[t] + F_{3x_1}, x_2[t] + F_{3x_2}, \ldots, x_m[t] + F_{3x_m}) \]
\[ x_1[t+1] = x_1[t] + (F_{1x_1} + 2F_{2x_1} + 2F_{3x_1} + F_{4x_1})/6 \]
\[ x_2[t+1] = x_2[t] + (F_{1x_2} + 2F_{2x_2} + 2F_{3x_2} + F_{4x_2})/6 \]
\[ \quad \vdots \]
\[ x_m[t+1] = x_m[t] + (F_{1x_m} + 2F_{2x_m} + 2F_{3x_m} + F_{4x_m})/6 \]
where \( t \) ranges from \( \text{LowerLimit} \) to \( \text{UpperLimit} \) 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 \( \text{TNTargetF}'s. \)

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**

\[
\text{TNvector} = \text{array}[0..\text{TNRowSize}] \text{ of Extended};
\]
\[
\text{TNmatrix} = \text{array}[0..\text{TNColumnSize}] \text{ of TNvector};
\]

\( \text{TNRowSize} \) is an upper bound for the number of values returned for a particular variable (\( \text{NumReturn} \)). \( \text{TNColumnSize} \) is an upper bound for the number of differential equations (\( \text{NumEquations} \)).

**User-Defined Functions**

\[
\text{function TNTargetF1(V : TNvector) : Extended;}
\]
\[
\text{function TNTargetF2(V : TNvector) : Extended;}
\]
\[
\text{function TNTargetF3(V : TNvector) : Extended;}
\]
\[
\text{function TNTargetF4(V : TNvector) : Extended;}
\]
\[
\text{function TNTargetF5(V : TNvector) : Extended;}
\]
\[
\text{function TNTargetF6(V : TNvector) : Extended;}
\]
\[
\text{function TNTargetF7(V : TNvector) : Extended;}
\]
\[
\text{function TNTargetF8(V : TNvector) : Extended;}
\]
\[
\text{function TNTargetF9(V : TNvector) : Extended;}
\]
\[
\text{function TNTargetF10(V : TNvector) : Extended;}
\]

These are the differential equations:

\[
\frac{dx_j}{dt} = \text{TNTargetF}_j(t, x_1, x_2, ..., x_{10})
\]

where \( j \) ranges from 1 to 10.
The elements of the vector V are defined as follows:

\[ V[0] = t \]
\[ V[1] = x_1 \]
\[ V[2] = x_2 \]
\[ \ldots \]
\[ V[10] = x_{10} \]

The procedure \textit{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.

\textbf{Input Parameters}

- \texttt{NumEquations} : Integer; Number of first-order differential equations
- \texttt{LowerLimit} : Extended; Lower limit of interval
- \texttt{UpperLimit} : Extended; Upper limit of interval
- \texttt{InitialValues} : TNvector; Values of \( x_1, x_2, \ldots, x_m \) at \textit{LowerLimit}
- \texttt{NumReturn} : Integer; Number of \((t, x_1, x_2, \ldots, x_m)\) values returned from the procedure
- \texttt{NumIntervals} : Integer; Number of subintervals used in the calculations
- \texttt{FuncVect} : array[1..10] of ProcPtr; Pointers to the ten equations

The preceding parameters must satisfy the following conditions:

1. \texttt{NumReturn} > 0
2. \texttt{NumIntervals} \( \geq \) \texttt{NumReturn}
3. \texttt{NumEquations} > 0
4. \texttt{LowerLimit} \( \neq \) \texttt{UpperLimit}
Output Parameters

SolutionValues : TNmatrix; Values of \( t, x_1, x_2, \ldots x_m \) 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 (TNTargetFlO is used as an example):

```pascal
function TNTargetFlO(V : TNvector) : Extended;
begin
    TNTargetFlO := 0.0;
end; { function TNTargetFlO }
```

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:

```pascal
F[11] := Spacing * TNTargetF11(CurrentValues);
```

More statements (for \( F[12] \), and so on) may be added as necessary. All new functions (for example, TNTargetF11) 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 $\text{NumIntervals}$ values in its calculations; however, you will rarely need to use these values. The rows of $\text{SolutionValues}$ will contain only $\text{NumReturn}$ values at roughly equally spaced $t$-values between the lower and upper limits. (They will be equally spaced only when $\text{NumIntervals}$ is a multiple of $\text{NumReturn}$.) Thus, you can ensure a highly accurate solution (by making $\text{NumIntervals}$ large) without generating an excessive amount of output (by making $\text{NumReturn}$ 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 ($\text{NumIntervals}$).

**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 $\text{InitialConditionSystem}$ in the variable $\text{FuncVect}$.

**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 \( F \sin(\omega t) \) (\( \omega \) represents the frequency of the driving force and \( t \) is time), the equation of motion of the mass is as follows:

\[
m \frac{d^2x}{dt^2} + k x = F \sin(\omega t)
\]

Given

\[
m = 2 \text{ kg} \\
F = 9 \text{ N} \\
k = 32 \text{ N/m} \\
\omega = 5 \text{ cycles/sec} \\
x(0) = 0 \text{ m} \\
\frac{dx(0)}{dt} = -2.5 \text{ m/sec}
\]

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:

\[
\frac{dx_1}{dt} = x_2 \\
\frac{dx_2}{dt} = \left( \frac{F}{m} \right) \sin(\omega t) - \left( \frac{k}{m} \right) x_1
\]

2. Code these equations into the program Runge_S1.pas:

```pascal
function TNTargetF1(V : TNvector) : Extended;
{
THIS IS THE FIRST DIFFERENTIAL EQUATION
}
{
\{ dx[l] \\
\{ ----- = TNTargetF1(t, x[l], x[2], ... x[m]) \}
\{ dt \}
\{ The vector V is defined: \}
\{ V[0] = t \}
\{ \ldots \}
\{ V[m] = x[m] \}
\{ where m is the number of coupled equations. \}
{
begin
TNTargetF1 := V[2];
end; \{ function TNTargetF1 \}
```
function TNTargetF2(V : TNvector) : Extended;

begin
    TNTargetF2 := 9/2 * Sin(5 * V[0]) - 32/2 * V[1];
end;

function TNTargetF3(V : TNvector) : Extended;

begin
    TNTargetF3 := 0.0;
end;

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 = 0.000000e+0: 0
Enter X[2] value at t = 0.000000e+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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000000</td>
<td>0.000000000000000e+0</td>
<td>-2.500000000000000e+0</td>
</tr>
<tr>
<td>0.20000000</td>
<td>-4.20735284275848e-1</td>
<td>-1.35075642830665e+0</td>
</tr>
<tr>
<td>0.40000000</td>
<td>-4.54648724216734e-1</td>
<td>1.04036531118478e+0</td>
</tr>
<tr>
<td>0.60000000</td>
<td>-7.05605786993375e-2</td>
<td>2.47497991717220e+0</td>
</tr>
<tr>
<td>0.80000000</td>
<td>3.78400378699554e-1</td>
<td>1.63411037473655e+0</td>
</tr>
<tr>
<td>1.00000000</td>
<td>4.79461767300631e-1</td>
<td>-7.09151289407566e-1</td>
</tr>
<tr>
<td>1.20000000</td>
<td>1.39708469016312e-1</td>
<td>-2.4004215228323e+0</td>
</tr>
<tr>
<td>1.40000000</td>
<td>-3.28491796183335e-1</td>
<td>1.4004215228323e+0</td>
</tr>
<tr>
<td>1.60000000</td>
<td>-4.94677974769031e-1</td>
<td>2.27781864414105e+0</td>
</tr>
<tr>
<td>1.80000000</td>
<td>-2.06059519715177e-1</td>
<td>2.09767516082022e+0</td>
</tr>
<tr>
<td>2.00000000</td>
<td>2.7200842396951e-1</td>
<td>2.09767516082022e+0</td>
</tr>
</tbody>
</table>

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

\[ x = \frac{F \sin(\omega t)}{m (\omega_0^2 - \omega^2)} \]

\[ \frac{dx}{dt} = \frac{F \omega \cos(\omega t)}{m (\omega_0^2 - \omega^2)} \]

where \( \omega_0 \) is the natural frequency of the system:

\[ \omega_0^2 = \frac{k}{m} \]

The period of oscillation is given by

\[ T = \frac{2\pi}{\omega} = 1.257 \text{ sec} \]

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

<table>
<thead>
<tr>
<th>t</th>
<th>Values of X</th>
<th>Derivative of X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.00000000000000E + 000</td>
<td>-2.50000000000000E + 000</td>
</tr>
<tr>
<td>0.2</td>
<td>-4.207354924039E - 001</td>
<td>-1.350755764670E + 000</td>
</tr>
<tr>
<td>0.4</td>
<td>-4.546487134128E - 001</td>
<td>1.040367091367E + 000</td>
</tr>
<tr>
<td>0.6</td>
<td>-7.056000402993E - 002</td>
<td>2.474981241501E + 000</td>
</tr>
<tr>
<td>0.8</td>
<td>3.784012476539E - 001</td>
<td>1.634109052159E + 000</td>
</tr>
<tr>
<td>1.0</td>
<td>4.794621373315E - 001</td>
<td>-7.091554636580E - 001</td>
</tr>
<tr>
<td>1.2</td>
<td>1.397077490994E - 001</td>
<td>-2.400425716625E + 000</td>
</tr>
<tr>
<td>1.4</td>
<td>-3.284932935935E - 001</td>
<td>-1.884755635858E + 000</td>
</tr>
<tr>
<td>1.6</td>
<td>-4.946791233116E - 001</td>
<td>3.637500845215E - 001</td>
</tr>
<tr>
<td>1.8</td>
<td>-2.060592426208E - 001</td>
<td>2.277825654711E + 000</td>
</tr>
<tr>
<td>2.0</td>
<td>2.720105544446E - 001</td>
<td>2.097678822691E + 000</td>
</tr>
</tbody>
</table>
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 $x_1'$ indicates $dx_1/dt$, 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

\[
x_1[LowerLimit] = a_1 \quad x_1'[LowerLimit] = b_1
\]
\[
x_2[LowerLimit] = a_2 \quad x_2'[LowerLimit] = b_2
\]
\[
\vdots \]
\[
x_m[LowerLimit] = a_m \quad x_m'[LowerLimit] = b_m
\]

and spacing

\[
h = (UpperLimit - LowerLimit)/NumIntervals
\]
rewrite each of the second-order differential equations as two, first-order differential equations:

\[
\frac{dx}{dt} = y_1 \\
\frac{dy}{dt} = TNTargetF1(t, x_1, y_1, x_2, y_2, \ldots, x_m, y_m) \\
\frac{dx_2}{dt} = y_2 \\
\frac{dx_3}{dt} = TNTargetF2(t, x_1, y_1, x_2, y_2, \ldots, x_m, y_m) \\
\vdots \\
\frac{dx_m}{dt} = y_m \\
\frac{dx_m}{dt} = TNTargetFm(t, x_1, y_1, x_2, y_2, \ldots, x_m, y_m)
\]

Then the fourth-order general Runge-Kutta method can be used to approximate the \(x_i\)'s and the \(y_j\)'s simultaneously.

The general Runge-Kutta formulas for these equations are as follows:

\[
F1x_1 = h \cdot y_1 \\
F1y_1 = h \cdot TNTargetF1(t, x_1[t], y_1[t], x_2[t], y_2[t], \ldots, x_m[t], y_m[t]) \\
F1x_2 = h \cdot y_2 \\
F1y_2 = h \cdot TNTargetF2(t, x_1[t], y_1[t], x_2[t], y_2[t], \ldots, x_m[t], y_m[t]) \\
\vdots \\
F1x_m = h \cdot y_m \\
F1y_m = h \cdot TNTargetFm(t, x_1[t], y_1[t], x_2[t], y_2[t], \ldots, x_m[t], y_m[t])
\]

\[
F2x_1 = h \cdot (y_1 + F1y_1/2) \\
F2y_1 = h \cdot NTTargetF1(t + h/2, x_1[t] + F1x_1/2, y_1[t] + F1y_1/2, x_2[t] + F1x_2/2, y_2[t] + F1y_2/2, \ldots, x_m[t] + F1x_m/2, y_m[t] + F1y_m/2) \\
F2x_2 = h \cdot (y_2 + F1y_2/2) \\
F2y_2 = h \cdot NTTargetF2(t + h/2, x_1[t] + F1x_1/2, y_1[t] + F1y_1/2, x_2[t] + F1x_2/2, y_2[t] + F1y_2/2, \ldots, x_m[t] + F1x_m/2, y_m[t] + F1y_m/2) \\
\vdots \\
F2x_m = h \cdot (y_m + F1y_m/2) \\
F2y_m = h \cdot NTTargetFm(t + h/2, x_1[t] + F1x_1/2, y_1[t] + F1y_1/2, x_2[t] + F1x_2/2, y_2[t] + F1y_2/2, \ldots, x_m[t] + F1x_m/2, y_m[t] + F1y_m/2)
\]
\[
F3x_1 = h \ast (y_1 + F2y_1/2)
\]
\[
F3y_1 = h \ast \text{TNTargetF1}(t + h/2, x_1[t] + F2x_1/2, y_1[t] + F2y_1/2, x_2[t] + F2x_2/2, y_2[t] + F2y_2/2, ..., x_m[t] + F2x_m/2, y_m[t] + F2y_m/2)
\]
\[
F3x_2 = h \ast (y_2 + F2y_2/2)
\]
\[
F3y_2 = h \ast \text{TNTargetF2}(t + h/2, x_1[t] + F2x_1/2, y_1[t] + F2y_1/2, x_2[t] + F2x_2/2, y_2[t] + F2y_2/2, ..., x_m[t] + F2x_m/2, y_m[t] + F2y_m/2)
\]
\[
\cdots
\]
\[
F3x_m = h \ast (y_m + F2y_m/2)
\]
\[
F3y_m = h \ast \text{TNTargetFm}(t + h/2, x_1[t] + F2x_1/2, y_1[t] + F2y_1/2, x_2[t] + F2x_2/2, y_2[t] + F2y_2/2, ..., x_m[t] + F2x_m/2, y_m[t] + F2y_m/2)
\]
\[
F4x_1 = h \ast (y_1 + F3y_1)
\]
\[
F4y_1 = h \ast \text{TNTargetF1}(t + h, x_1[t] + F3x_1, y_1[t] + F3y_1, x_2[t] + F3x_2, y_2[t] + F3y_2, ..., x_m[t] + F3x_m, y_m[t] + F3y_m)
\]
\[
F4x_2 = h \ast (y_2 + F3y_2)
\]
\[
F4y_2 = h \ast \text{TNTargetF2}(t + h, x_1[t] + F3x_1, y_1[t] + F3y_1, x_2[t] + F3x_2, y_2[t] + F3y_2, ..., x_m[t] + F3x_m, y_m[t] + F3y_m)
\]
\[
\cdots
\]
\[
F4x_m = h \ast (y_m + F3y_m)
\]
\[
F4y_m = h \ast \text{TNTargetFm}(t + h, x_1[t] + F3x_1, y_1[t] + F3y_1, x_2[t] + F3x_2, y_2[t] + F3y_2, ..., x_m[t] + F3x_m, y_m[t] + F3y_m)
\]
\[
x_1[t+1] = x_1[t] + (F1x_1 + 2 \ast F2x_1 + 2 \ast F3x_1 + F4x_1)/6
\]
\[
y_1[t+1] = y_1[t] + (F1y_1 + 2 \ast F2y_1 + 2 \ast F3y_1 + F4y_1)/6
\]
\[
x_2[t+1] = x_2[t] + (F1x_2 + 2 \ast F2x_2 + 2 \ast F3x_2 + F4x_2)/6
\]
\[
y_2[t+1] = y_2[t] + (F1y_2 + 2 \ast F2y_2 + 2 \ast F3y_2 + F4y_2)/6
\]
\[
\cdots
\]
\[
x_m[t+1] = x_m[t] + (F1x_m + 2 \ast F2x_m + 2 \ast F3x_m + F4x_m)/6
\]
\[
y_m[t+1] = y_m[t] + (F1y_m + 2 \ast F2y_m + 2 \ast F3y_m + F4y_m)/6
\]

where \( t \) ranges from \( \text{LowerLimit} \) to \( \text{UpperLimit} \) 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**

```plaintext
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**

```plaintext
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:
\[
d^2 x_j / dt^2 = TNTargetFj(t, x_1, x'_1, x_2, x'_2, \ldots, x_{10}, x'_{10})
\]
where \( j \) ranges from 1 to 10.
```
The elements of the vector $V$ are defined as follows:

$$
\begin{align*}
  V[0].x &= t \\
  V[1].x &= x[1] \\
  V[1].xDeriv &= x'[1] \\
  V[2].xDeriv &= x'[2] \\
  \vdots \\
  V[10].x &= x[10] \\
  V[10].xDeriv &= x'[10]
\end{align*}
$$

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumEquations : Integer;</td>
<td>Number of second-order differential equations</td>
</tr>
<tr>
<td>LowerLimit : Extended;</td>
<td>Lower limit of interval</td>
</tr>
<tr>
<td>UpperLimit : Extended;</td>
<td>Upper limit of interval</td>
</tr>
<tr>
<td>InitialValues : TNvector2;</td>
<td>Values of $x_j$'s and $x_j'$'s at $LowerLimit$</td>
</tr>
<tr>
<td>NumReturn : Integer;</td>
<td>Number of $(t, x, x', x_2, x'_2, \ldots, x_m, x'_m)$ values returned from the procedure</td>
</tr>
<tr>
<td>NumIntervals : Integer;</td>
<td>Number of subintervals used in the calculations</td>
</tr>
<tr>
<td>FuncVect : array[1..10] of ProcPtr;</td>
<td>Pointers to the ten equations</td>
</tr>
</tbody>
</table>

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 $t$, $x'$, and $x''$ 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 $x'$ and $x''$ 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 (TNTargetFlO is used as an example):

```plaintext
function TNTargetFlO(V : TNvector) : Extended;
begin
    TNTargetFlO := 0.0
end; { function TNTargetFlO }
```

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:

```plaintext
F[11].x := Spacing * TNTargetFlO1(CurrentValues);
```

More statements (for $F[12]$, and so on) may be added as necessary. All new functions (for example, TNTargetFlO1) 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 \textit{NumIntervals} values in its calculations; however, you will rarely need to use these values. The rows of \textit{SolutionValues} will contain only \textit{NumReturn} values at roughly equally spaced \(t\)-values between the lower and upper limits. (They will be equally spaced only when \textit{NumIntervals} is a multiple of \textit{NumReturn}.) Thus, you can ensure a highly accurate solution (by making \textit{NumIntervals} large) without generating an excessive amount of output (by making \textit{NumReturn} small).

\textbf{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 (\textit{NumIntervals}).

\textit{Sample Program}

The sample program \texttt{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 \texttt{InitialConditionSystem2} procedure in the variable \texttt{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 $k$:

![Diagram of two weights and a spring](image)

The equations of motion of these two masses are as follows:

\[
\begin{align*}
    m \frac{d^2 x}{dt^2} &= -mg \frac{x}{l} - k(x - y) \\
    m \frac{d^2 y}{dt^2} &= -mg \frac{y}{l} + k(x - y)
\end{align*}
\]

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
- $\frac{dx(0)}{dt} = 0$ m/sec
- $\frac{dy(0)}{dt} = 0$ 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:

\[
\begin{align*}
    \frac{d^2 x}{dt^2} &= -g \frac{x}{l} - k \frac{x}{m} \frac{x - y}{m} \\
    \frac{d^2 y}{dt^2} &= -g \frac{y}{l} + k \frac{x}{m} \frac{x - y}{m}
\end{align*}
\]
2. Code these equations into the program Runge_S2.pas:

```
function TNTargetFl(V : TNvector) : Extended;
{-----------------------------------------------------}
{ THIS IS THE FIRST DIFFERENTIAL EQUATION }
{-----------------------------------------------------}
{ }
{ \frac{d^2 x[1]}{dt^2} = TNTargetFl(t, x[1], x'[1], x[2], x'[2], ... x[m], x'[m] }
{ }
{ 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;
  TNTargetFl := -9.8 * V[1].x/0.6125 - 32/2 * (V[1].x - V[2].x);
end;
```
function TNTargetF2(V : TNvector) : Extended;

{ THIS IS THE SECOND DIFFERENTIAL EQUATION }

\[ \frac{d^2x[2]}{dt^2} = TNTargetF2(t, x[1], x'[1], x[2], x'[2], \ldots, x[m], x'[m]) \]

The elements of the vector \( V \) are defined:
- \( V[0].x = t \)
- \( V[1].x = X[1] \)
- \( V[1].xDeriv = X'[1] \)
- \( V[2].x = X[2] \)
- \( V[2].xDeriv = X'[2] \)
  
  \vdots

- \( V[m].x = X[m] \)
- \( V[m].xDeriv = X'[m] \)

{ where \( m \) is the number of coupled equations. }

var
  t : Extended;

begin
  t := v[0].x;
  TNTargetF2 := -9.8 \times V[2].x/0.6125 + 32/2 \times (V[1].x - V[2].x);
end;

{ function TNTargetF2 }
function TNTargetF3(V : TNvector) : Extended;

{ THIS IS THE THIRD DIFFERENTIAL EQUATION }

\[ \frac{d^2 x[3]}{dt^2} = TNTargetF3(t, x[1], x'[1], x[2], x'[2], \ldots, x[m], x'[m]) \]

The elements of the vector V are defined:
- \( V[0].x = t \)
- \( V[1].x = X[1] \)
- \( V[1].xDeriv = X'[1] \)
- \( V[2].x = X[2] \)
- \( V[2].xDeriv = X'[2] \)

\[ \ldots \]
- \( V[m].x = X[m] \)
- \( V[m].xDeriv = X'[m] \)

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 \( t = 0 \times 10^0 \): 0.01
Enter \( X'[1] \) value at \( t = 0 \times 10^0 \): 0.00
Enter \( X[2] \) value at \( t = 0 \times 10^0 \): -0.01
Enter \( X'[2] \) value at \( t = 0 \times 10^0 \): 0.00

Number of values to return (1-70)? 10

Number of intervals (\( \geq 10 \))? 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:
\[ X[1] = 1.00000000000000e-2 \]
\[ X'[1] = 0.00000000000000e+0 \]
\[ X[2] = -1.00000000000000e-2 \]
\[ X'[2] = 0.00000000000000e+0 \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000000</td>
<td>1.00000000000000e-2</td>
<td>0.00000000000000e+0</td>
</tr>
<tr>
<td>0.10000000</td>
<td>7.69447788485895e-3</td>
<td>-4.42511063153028e-2</td>
</tr>
<tr>
<td>0.20000000</td>
<td>1.84099813762452e-3</td>
<td>-6.80978317847279e-2</td>
</tr>
<tr>
<td>0.30000000</td>
<td>-4.86137387553900e-3</td>
<td>-6.05443464988731e-2</td>
</tr>
<tr>
<td>0.40000000</td>
<td>-9.32214486443693e-3</td>
<td>-2.50735962938904e-2</td>
</tr>
<tr>
<td>0.50000000</td>
<td>-9.48443369885918e-3</td>
<td>2.1958691271007e-2</td>
</tr>
<tr>
<td>0.60000000</td>
<td>-5.27340834792187e-3</td>
<td>5.88657408762406e-2</td>
</tr>
<tr>
<td>0.70000000</td>
<td>1.36920877108260e-3</td>
<td>6.86295294795967e-2</td>
</tr>
<tr>
<td>0.80000000</td>
<td>7.38047758874091e-3</td>
<td>4.6747939392010e-2</td>
</tr>
<tr>
<td>0.90000000</td>
<td>9.98857556718864e-3</td>
<td>3.31066873866278e-3</td>
</tr>
<tr>
<td>1.00000000</td>
<td>7.99089728515028e-3</td>
<td>-4.16531651968366e-2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000000</td>
<td>-1.00000000000000e-2</td>
<td>0.00000000000000e+0</td>
</tr>
<tr>
<td>0.10000000</td>
<td>-7.69447788485895e-3</td>
<td>4.42511063153028e-2</td>
</tr>
<tr>
<td>0.20000000</td>
<td>-1.84099813762452e-3</td>
<td>6.80978317847279e-2</td>
</tr>
<tr>
<td>0.30000000</td>
<td>4.86137387553900e-3</td>
<td>6.05443464988731e-2</td>
</tr>
<tr>
<td>0.40000000</td>
<td>9.32214486443693e-3</td>
<td>2.50735962938904e-2</td>
</tr>
<tr>
<td>0.50000000</td>
<td>9.48443369885918e-3</td>
<td>-2.1958691271007e-2</td>
</tr>
<tr>
<td>0.60000000</td>
<td>5.27340834792187e-3</td>
<td>-5.88657408762406e-2</td>
</tr>
<tr>
<td>0.70000000</td>
<td>-1.36920877108260e-3</td>
<td>-6.86295294795967e-2</td>
</tr>
<tr>
<td>0.80000000</td>
<td>-7.38047758874091e-3</td>
<td>-4.6747939392010e-2</td>
</tr>
<tr>
<td>0.90000000</td>
<td>-9.98857556718864e-3</td>
<td>-3.31066873866278e-3</td>
</tr>
<tr>
<td>1.00000000</td>
<td>-7.99089728515028e-3</td>
<td>4.16531651968366e-2</td>
</tr>
</tbody>
</table>

The weights move in opposite directions; the system is in one of its normal modes. The natural frequency \( \omega_0 \) is given by the following:

\[
\omega_0^2 = g/l + 2k/m
\]

\( \omega_0 = 6.928 \) cycles/sec

Thus the period of oscillation, \( t \), is

\[
t = 2\pi/\omega_0 = 0.9069 \text{ sec}
\]
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

\[ \frac{d^2y}{dx^2} = TNTargetF(x, y, y') \]

where \( y' \) represents \( \frac{dy}{dx} \), 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;

\[ \frac{d^2 y}{dx^2} = TNTargetF(x, y, dy/dx) \]

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. \( \text{NumReturn} > 0 \)
2. \( \text{NumIntervals} \geq \text{NumReturn} \)
3. \( \text{LowerLimit} \neq \text{UpperLimit} \)
4. \( \text{Tolerance} > 0 \)
5. \( \text{MaxIter} > 0 \)
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 $\leq 0$
  - 5: MaxIter $\leq 0$
  - 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:

\[
y'' = 192 \frac{\sqrt{y}}{y'} \\
0 \leq x \leq 1
\]

\[
y(1) = 1 \\
y(2) = 16
\]

1. Code the differential equation into the program:

```pascal
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 \( X = 0.000000e+0 \): 1
   Enter Y value at \( X = 1.000000e+0 \): 16
   Enter a guess for the slope at \( X = 0.000000e+0 \): 15
   Number of points returned (1-40)? 10
   Number of intervals (\( \geq 10 \))? 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

<table>
<thead>
<tr>
<th>X</th>
<th>Y Value</th>
<th>Derivative of Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000000000000e+0</td>
<td>1.00000000000000e+0</td>
<td>4.00053795390884e+0</td>
</tr>
<tr>
<td>1.00000000000000e-1</td>
<td>1.46417721408153e+0</td>
<td>5.32386904044879e+0</td>
</tr>
<tr>
<td>2.00000000000000e-1</td>
<td>2.07370562259973e+0</td>
<td>6.91162114244397e+0</td>
</tr>
<tr>
<td>3.00000000000000e-1</td>
<td>2.85621262766442e+0</td>
<td>8.78752756627335e+0</td>
</tr>
<tr>
<td>4.00000000000000e-1</td>
<td>3.84170902091389e+0</td>
<td>1.09754927855527e+1</td>
</tr>
<tr>
<td>5.00000000000000e-1</td>
<td>5.06259931530967e+0</td>
<td>1.34994802016423e+1</td>
</tr>
<tr>
<td>6.00000000000000e-1</td>
<td>6.55368547624580e+0</td>
<td>1.63834750611955e+1</td>
</tr>
<tr>
<td>7.00000000000000e-1</td>
<td>8.35216836918581e+0</td>
<td>1.96514712240017e+1</td>
</tr>
<tr>
<td>8.00000000000000e-1</td>
<td>1.04976483580762e+1</td>
<td>2.33274661179548e+1</td>
</tr>
<tr>
<td>9.00000000000000e-1</td>
<td>1.30321255669365e+1</td>
<td>2.7435487043772e+1</td>
</tr>
<tr>
<td>1.00000000000000e+0</td>
<td>1.60000000000094e+1</td>
<td>3.1999486182108e+1</td>
</tr>
</tbody>
</table>
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 \( X = 0.000000e+0 \): 1
Enter Y value at \( X = 1.000000e+0 \): 16
Enter a guess for the slope at \( X = 0.000000e+0 \): 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: 0.0000000000000000e+0
Upper Limit: 1.0000000000000000e+0
Value of Y at 0.0000: 1.0000000000000000e+0
Value of Y at 1.0000: 1.6000000000000000e+1
Initial slope at 0.0000: 1.5000000000000000e+1
NumIntervals: 100
Tolerance: 1.0000000000000000e-6
Maximum number of iterations: 100

Number of iterations: 5

<table>
<thead>
<tr>
<th>(X)</th>
<th>(Y) Value</th>
<th>Derivative of (Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00000000000000e+0</td>
<td>1.0000000000000000e+0</td>
<td>4.000000062625639e+0</td>
</tr>
<tr>
<td>1.00000000000000e+0</td>
<td>1.46410005120828e+0</td>
<td>5.32400035609946e+0</td>
</tr>
<tr>
<td>2.00000000000000e+0</td>
<td>2.07360008576235e+0</td>
<td>6.91200027103432e+0</td>
</tr>
<tr>
<td>3.00000000000000e+0</td>
<td>2.85610011557157e+0</td>
<td>8.78800025750412e+0</td>
</tr>
<tr>
<td>4.00000000000000e+0</td>
<td>3.84160014547825e+0</td>
<td>1.09760002747783e+1</td>
</tr>
<tr>
<td>5.00000000000000e+0</td>
<td>5.06250017769403e+0</td>
<td>1.35000003070170e+1</td>
</tr>
<tr>
<td>6.00000000000000e+0</td>
<td>6.5360021337285e+0</td>
<td>1.63840003476283e+1</td>
</tr>
<tr>
<td>7.00000000000000e+0</td>
<td>8.35210025321452e+0</td>
<td>1.96520003937080e+1</td>
</tr>
<tr>
<td>8.00000000000000e+0</td>
<td>1.04976002977125e+1</td>
<td>2.33280004439140e+1</td>
</tr>
<tr>
<td>9.00000000000000e+0</td>
<td>1.30321003472617e+1</td>
<td>2.74360004976014e+1</td>
</tr>
<tr>
<td>1.00000000000000e+0</td>
<td>1.60000004022081e+1</td>
<td>3.20000005544507e+1</td>
</tr>
</tbody>
</table>
The exact solution is

\[ y = (x + 1)^4 \]

<table>
<thead>
<tr>
<th>X</th>
<th>Y Value</th>
<th>Derivative of Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.0000000000</td>
<td>4.0000000000</td>
</tr>
<tr>
<td>0.1</td>
<td>1.4641000000</td>
<td>5.3240000000</td>
</tr>
<tr>
<td>0.2</td>
<td>2.0736000000</td>
<td>6.9120000000</td>
</tr>
<tr>
<td>0.3</td>
<td>2.8561000000</td>
<td>8.7880000000</td>
</tr>
<tr>
<td>0.4</td>
<td>3.8416000000</td>
<td>1.0976000000</td>
</tr>
<tr>
<td>0.5</td>
<td>5.0625000000</td>
<td>1.3500000000</td>
</tr>
<tr>
<td>0.6</td>
<td>6.5536000000</td>
<td>1.6384000000</td>
</tr>
<tr>
<td>0.7</td>
<td>8.3521000000</td>
<td>1.9652000000</td>
</tr>
<tr>
<td>0.8</td>
<td>1.0497600000</td>
<td>2.3328000000</td>
</tr>
<tr>
<td>0.9</td>
<td>1.3032100000</td>
<td>2.7436000000</td>
</tr>
<tr>
<td>1.0</td>
<td>1.6000000000</td>
<td>3.2000000000</td>
</tr>
</tbody>
</table>

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 condi­
tions (Burden and Faires 1985, 519–524).

Given a second-order differential equation (Burden and Faires 1985, 261–264) of
the form

\[ \frac{d^2y}{dx^2} = TNTargetF(x, y, y') \]

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):

\[ y'[LowerLimit] = 0 \quad y[LowerLimit] = LowerInitial \]
\[ y'[LowerLimit] = 1 \quad y[LowerLimit] = LowerInitial \]

(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 \( LowerLimit, UpperLimit, LowerInitial, UpperInitial, NumInt-
ervals, \) and \( TNTargetF \).
**User-Defined Types**

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

**User-Defined Functions**

TNTargetF(x, y, yPrime : Extended) : Extended;

\[ \frac{d^2y}{dx^2} = TNTargetF(x, y, dy/dx) \]

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** ≥ **NumReturn**
3. **LowerLimit** ≠ **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

210 Turbo Pascal Numerical Methods Toolbox
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:

\[
y'' = y'/x - y/\sqrt{x} + 1 \quad 1 = x \leq 10
\]

\[y(1) = 1\]

\[y(10) = 76.974149\]

1. Code the differential equation into the program Linshot2.pas:

```pascal
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.

<table>
<thead>
<tr>
<th>X</th>
<th>Y Value</th>
<th>Derivative of Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00000000000000e+0</td>
<td>1.00000000000000e+0</td>
<td>1.00000000000000e+0</td>
</tr>
<tr>
<td>2.00000000000000e+0</td>
<td>2.61170356138588e+0</td>
<td>2.30627678512124e+0</td>
</tr>
<tr>
<td>3.00000000000000e+0</td>
<td>5.70207271413620e+0</td>
<td>3.90115296191831e+0</td>
</tr>
<tr>
<td>4.00000000000000e+0</td>
<td>1.04528257144295e+1</td>
<td>5.61367861126495e+0</td>
</tr>
<tr>
<td>5.00000000000000e+0</td>
<td>1.69509879730537e+1</td>
<td>7.39067355864438e+0</td>
</tr>
<tr>
<td>6.00000000000000e+0</td>
<td>2.52478687612139e+1</td>
<td>9.20845513089500e+0</td>
</tr>
<tr>
<td>7.00000000000000e+0</td>
<td>3.53773649984557e+1</td>
<td>1.10543869346579e+1</td>
</tr>
<tr>
<td>8.00000000000000e+0</td>
<td>4.73635728977226e+1</td>
<td>1.29209245920937e+1</td>
</tr>
<tr>
<td>9.00000000000000e+0</td>
<td>6.12245068576119e+1</td>
<td>1.48030211472994e+1</td>
</tr>
<tr>
<td>1.00000000000000e+1</td>
<td>7.69741490000000e+1</td>
<td>1.66978931711222e+1</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>X</th>
<th>Y Value</th>
<th>Derivative of Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.e+0</td>
<td>1.e+0</td>
<td>1.e+0</td>
</tr>
<tr>
<td>2.e+0</td>
<td>2.61370547174514e+0</td>
<td>2.30685275847028e+0</td>
</tr>
<tr>
<td>3.e+0</td>
<td>5.70416298088411e+0</td>
<td>3.90138768358927e+0</td>
</tr>
<tr>
<td>4.e+0</td>
<td>1.04548224122436e+1</td>
<td>5.61370562650429e+0</td>
</tr>
<tr>
<td>5.e+0</td>
<td>1.69528103026793e+1</td>
<td>7.3905620842440e+0</td>
</tr>
<tr>
<td>6.e+0</td>
<td>2.52494430584438e+1</td>
<td>9.2082405332639e+0</td>
</tr>
<tr>
<td>7.e+0</td>
<td>3.53786288412165e+1</td>
<td>1.10540898579729e+1</td>
</tr>
<tr>
<td>8.e+0</td>
<td>4.73644675641047e+1</td>
<td>1.29205584690303e+1</td>
</tr>
<tr>
<td>9.e+0</td>
<td>6.12249767166508e+1</td>
<td>1.48027754364805e+1</td>
</tr>
<tr>
<td>1.e+1</td>
<td>7.69741490000000e+1</td>
<td>1.66974149235206e+1</td>
</tr>
</tbody>
</table>

The exact solution is

\[ y = x \cdot x - x \cdot \ln(x) \]

\[ y(1) = 1 \quad y'(1) = 1 \]
\[ y(10) = 7.6974149 \quad y'(10) = 16.6974149 \]

The second approximation is more accurate.
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

$$ Y = a \times X + b $$

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 $(a \times X + b - Y)$ 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 \(j\)th 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:

\[
\text{S.D.} = (\langle Y[i] - F(X[i]) \rangle^2 / (m - n))^{1/2}
\]

where \(F(X[i])\) is the least-squares solution at the point \(X[i]\), \(\langle Y[i] - F(X[i]) \rangle\) 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:

\[ \text{Solution}[j] = a_j \quad 1 \leq j \leq \text{NumTerms} \]

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:

\[ Solution[j] = F_{j-1} \quad 1 \leq j \leq \text{NumTerms} \]

where \( F_{j-1} \) is the \((j-1)\)th term in the Fourier series. Following are the first few terms in the Fourier series:

\[
\begin{align*}
F[0] &= 1 \\
F[1] &= \cos(x) \\
F[2] &= \sin(x) \\
F[3] &= \cos(2x) \\
F[4] &= \sin(2x) \\
F[5] &= \cos(3x) \\
F[6] &= \sin(3x)
\end{align*}
\]

**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:

\[ \ln(y) = \ln(a) + b \cdot \ln(x) \]

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:

\[
\begin{align*}
\text{Solution}[1] &= a \\
\text{Solution}[2] &= b
\end{align*}
\]

**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 $\text{NumTerms}$ (unless $\text{NumTerms} > \text{NumPoints}$, in which case Error 3 will occur). The elements of the $\text{Solution}$ vector will be as follows:

\[
\text{Solution}[1] = a \\
\text{Solution}[2] = b
\]

**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 $\text{NumTerms}$ (unless $\text{NumTerms} > \text{NumPoints}$, in which case Error 3 will occur). The elements of the $\text{Solution}$ vector will be as follows:

\[
\text{Solution}[1] = a \\
\text{Solution}[2] = b
\]

**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 $\text{sqr}(x)$ 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 5.3990665131879e-1
0.30000000000000e+0 2.41970724519143e+0
0.40000000000000e+0 3.98942280401433e+0
0.02000000000000e+0 2.91946925791461e-3
0.04000000000000e+0 6.1190130113775e-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 7.8950158008939e-1
0.24000000000000e+0 1.10920834679455e0
0.26000000000000e+0 1.49727465653745e0
0.28000000000000e+0 1.94186054983213e0
0.32000000000000e+0 2.89691552761483e0
0.34000000000000e+0 3.33224602891800e0
0.36000000000000e+0 3.68270140303323e0
0.38000000000000e+0 3.91042693975456e0
```

(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:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.200</td>
<td>0.0443185</td>
</tr>
<tr>
<td>0.300</td>
<td>0.5399097</td>
</tr>
<tr>
<td>0.400</td>
<td>2.4197072</td>
</tr>
<tr>
<td>0.500</td>
<td>3.9894228</td>
</tr>
<tr>
<td>0.100</td>
<td>0.0013383</td>
</tr>
<tr>
<td>0.120</td>
<td>0.0029195</td>
</tr>
<tr>
<td>0.140</td>
<td>0.0061190</td>
</tr>
<tr>
<td>0.160</td>
<td>0.0123222</td>
</tr>
<tr>
<td>0.180</td>
<td>0.0238409</td>
</tr>
<tr>
<td>0.220</td>
<td>0.0791545</td>
</tr>
<tr>
<td>0.240</td>
<td>0.1358297</td>
</tr>
<tr>
<td>0.260</td>
<td>0.2239453</td>
</tr>
<tr>
<td>0.280</td>
<td>0.3547459</td>
</tr>
<tr>
<td>0.320</td>
<td>0.7895016</td>
</tr>
<tr>
<td>0.340</td>
<td>1.1092083</td>
</tr>
<tr>
<td>0.360</td>
<td>1.4972747</td>
</tr>
<tr>
<td>0.380</td>
<td>1.9418605</td>
</tr>
<tr>
<td>0.420</td>
<td>2.8969155</td>
</tr>
<tr>
<td>0.440</td>
<td>3.3322460</td>
</tr>
<tr>
<td>0.460</td>
<td>3.6827014</td>
</tr>
<tr>
<td>0.480</td>
<td>3.9104269</td>
</tr>
</tbody>
</table>

*----------------------------------------------------------*

Polyomial 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
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 \textit{FitType} parameter to \texttt{Log} in the call to procedure \texttt{LeastSquares}.

Run Least.pas:

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

File name? Sample9A.dat

Number of terms in the least squares fit (\(\leq 21\))? 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 \texttt{OK}.

Least-Squares Approximation
The Data Points:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.200</td>
<td>0.0443185</td>
</tr>
<tr>
<td>0.300</td>
<td>0.5399097</td>
</tr>
<tr>
<td>0.400</td>
<td>2.4197072</td>
</tr>
<tr>
<td>0.500</td>
<td>3.9894228</td>
</tr>
<tr>
<td>0.100</td>
<td>0.0013383</td>
</tr>
<tr>
<td>0.120</td>
<td>0.0029195</td>
</tr>
<tr>
<td>0.140</td>
<td>0.0061190</td>
</tr>
<tr>
<td>0.160</td>
<td>0.0123222</td>
</tr>
<tr>
<td>0.180</td>
<td>0.0238409</td>
</tr>
<tr>
<td>0.220</td>
<td>0.0791545</td>
</tr>
<tr>
<td>0.240</td>
<td>0.1358297</td>
</tr>
<tr>
<td>0.260</td>
<td>0.2239453</td>
</tr>
<tr>
<td>0.280</td>
<td>0.3547459</td>
</tr>
<tr>
<td>0.320</td>
<td>0.789016</td>
</tr>
<tr>
<td>0.340</td>
<td>1.1092083</td>
</tr>
<tr>
<td>0.360</td>
<td>1.4972747</td>
</tr>
<tr>
<td>0.380</td>
<td>1.9416005</td>
</tr>
<tr>
<td>0.420</td>
<td>2.8969155</td>
</tr>
<tr>
<td>0.440</td>
<td>3.322460</td>
</tr>
<tr>
<td>0.460</td>
<td>3.6827014</td>
</tr>
<tr>
<td>0.480</td>
<td>3.9104269</td>
</tr>
</tbody>
</table>

Logarithmic Least Squares Fit

Coefficients in least squares approximation:
- Coefficient 0: 2.5984092388e+0
- Coefficient 1: 6.0253489684e+0

<table>
<thead>
<tr>
<th>X</th>
<th>Least Squares Fit</th>
<th>Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.200</td>
<td>4.8470072527e-1</td>
<td>4.4038224115e-1</td>
</tr>
<tr>
<td>0.300</td>
<td>1.5382650082e+0</td>
<td>9.9835534307e-1</td>
</tr>
<tr>
<td>0.400</td>
<td>2.2857807631e+0</td>
<td>-1.3392648209e-1</td>
</tr>
<tr>
<td>0.500</td>
<td>2.865590284e+0</td>
<td>-1.1238237756e+0</td>
</tr>
<tr>
<td>0.100</td>
<td>-1.3163793126e+0</td>
<td>-1.3177176148e+0</td>
</tr>
<tr>
<td>0.120</td>
<td>-8.426329495e-1</td>
<td>-8.455276421e-1</td>
</tr>
<tr>
<td>0.140</td>
<td>-4.4208674432e-1</td>
<td>-4.4820576362e-1</td>
</tr>
<tr>
<td>0.160</td>
<td>-9.5117540049e-2</td>
<td>-1.043973173e-1</td>
</tr>
<tr>
<td>0.180</td>
<td>2.1093098798e-1</td>
<td>1.8709010596e-1</td>
</tr>
<tr>
<td>0.220</td>
<td>7.323557703e-1</td>
<td>6.5320106120e-1</td>
</tr>
<tr>
<td>0.240</td>
<td>9.584674288e-1</td>
<td>8.2261705055e-1</td>
</tr>
<tr>
<td>0.260</td>
<td>1.1664304540e-0</td>
<td>9.4248515105e-1</td>
</tr>
<tr>
<td>0.280</td>
<td>1.3589932935e-0</td>
<td>1.0042473651e+0</td>
</tr>
<tr>
<td>0.320</td>
<td>1.7059624978e-0</td>
<td>9.1646091478e-1</td>
</tr>
<tr>
<td>0.340</td>
<td>1.8634900752e-0</td>
<td>7.5428172842e-1</td>
</tr>
<tr>
<td>0.360</td>
<td>2.0120110250e-0</td>
<td>5.1473636946e-1</td>
</tr>
<tr>
<td>0.380</td>
<td>2.1524997931e-0</td>
<td>2.1063924325e-1</td>
</tr>
<tr>
<td>0.420</td>
<td>2.4125575764e-0</td>
<td>-4.8435795117e-1</td>
</tr>
<tr>
<td>0.440</td>
<td>2.5334356149e-0</td>
<td>-7.9881041405e-1</td>
</tr>
<tr>
<td>0.460</td>
<td>2.6893985845e-0</td>
<td>-1.0337619176e+0</td>
</tr>
<tr>
<td>0.480</td>
<td>2.7595267807e-0</td>
<td>-1.1509001590e+0</td>
</tr>
</tbody>
</table>

Standard Deviation: 8.320742e-1
The logarithmic function that best fits this data is as follows:

\[ y = 2.59841 \times \ln(6.02535x) \]

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.
Fourier transforms are used to analyze periodic phenomena such as waves. A continuous function \( f \) that has period \( 2\pi (= 2 \times 3.14159265...) \); 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 nth 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

$$x(n) = f(nT/N) \quad n = 0, 1, ..., N - 1$$

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 $t = nT/N, n = 0, 1, ..., N - 1$. 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

\[ X_f = N^{-1/2} \sum_{n=0}^{N-1} x_n \exp \left(2\pi i n f/N\right) \quad f = 0..N - 1 \]

where \( i \) is the square root of \(-1\). The inverse Fourier transform (which may also be calculated with **ComplexFFT**) is defined as

\[ \bar{x}_n = N^{-1/2} \sum_{f=0}^{N-1} \bar{X}_f \exp \left(2\pi i f n/N\right) \quad n = 0..N - 1 \]

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 \( 2 \times 4^k \).

**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

\[ y_m = \sum_{n=0}^{N-1} x_n h_{m-n} \quad m = 0, 1, ... N - 1 \]

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:

\[ Y_m = X_m H_m \quad m = 0, 1, ... N - 1 \]

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 $X_{\text{Real}}$ and $H_{\text{Real}}$, combine them into a complex vector $X_{\text{Real}} + iH_{\text{Real}}$, 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

$$c_m = \sum_{n=0}^{N-1} x_n h_{n+m} \quad m = 0, 1, \ldots, N - 1$$

where subscripts are taken modulo $N$ (circular convolution). This can be computed using FFTs with a method analogous to that used in **ComplexConvolution**:

$$C_m = X_m H_{N-m} \quad m = 0, 1, \ldots, N - 1$$

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 per-
formed. The procedure is as follows:

1. Given two real vectors \(X\text{Real}\) and \(H\text{Real}\), combine them into a complex vec-
tor \(X\text{Real} + iH\text{Real}\), 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 \(X_j = \overline{X_{-j}}\), where the bar stands for
complex conjugation).

4. Multiply each element of the transform of the first data set by the appropri-
ate 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 men-
tion 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 trans-
formed. The high frequencies will falsely contribute to the transform. This con-
tribution 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
ture, 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 bound-
ary is created because the sine of 0 does not equal the sine of 270. High frequen-
cies 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**

```pascal
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. \(\text{NumPoints} \geq 2\).
2. \(\text{NumPoints}\) 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: \(\text{NumPoints} < 2\)
2: \(\text{NumPoints}\) not a power of two [four]

**Syntax of the Procedure Call**

\[
\text{TestInput}(\text{NumPoints}, \text{NumberOfBits}, \text{Error});
\]
Procedure MakeSinCosTable

Description

This example creates a look-up table of \( \frac{\text{NumPoints}}{2} \) [3/4 \( \text{NumPoints} \)] roots of unity. The roots of unity are defined as follows:

\[
\text{Root}_n = \exp\left(-i \frac{2\pi n}{\text{NumPoints}}\right) \quad n = 0..\frac{\text{NumPoints}}{2} [3/4 \text{NumPoints}]
\]

where \( i \) is the square root of \(-1\). These values are stored in two tables: \( \text{SinTable} \), containing the imaginary parts of the roots of unity, and \( \text{CosTable} \), 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

\( \text{NumPoints} : \text{Integer}; \) Number of data points

The preceding parameter must satisfy the following conditions:

1. \( \text{NumPoints} \geq 2 \).
2. \( \text{NumPoints} \) must be a power of two [four].

Output Parameters

\( \text{SinTable} : \text{TVectorPtr}; \) Table of sine values
\( \text{CosTable} : \text{TVectorPtr}; \) Table of cosine values

Syntax of the Procedure Call

\[
\text{MakeSinCosTable(NumPoints, SinTable, CosTable);}
\]

Procedure ComplexFFT, RealFFT

Description

This example implements the particular variation of the Cooley-Tukey algorithm. The Fast Fourier Transform of the data \( \text{XReal}, \text{XImag} \) is made in place and is thus returned in the vectors \( \text{XReal}, \text{XImag} \). 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**: TVectorPtr; Pointer to real values of the data points
- **XImag**: TVectorPtr; Pointer to imaginary values of the data points
- **SinTable**: TVectorPtr; Table of sine values
- **CosTable**: TVectorPtr; 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**: TVectorPtr; Pointer to real values of the discrete Fourier transform of the input data
- **XImag**: TVectorPtr; 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 ≥ 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

\[
y_m = \sum_{n=0}^{N-1} x_n h_{m-n} \quad m = 0, 1, \ldots, N - 1
\]

where subscripts are taken modulo \( N \) (circular convolution). The basic theorem that allows us to calculate these effectively using FFTs is as follows:

\[
Y_m = X_m H_m \quad m = 0, 1, \ldots, N - 1
\]

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 ≥ 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 $X_f = X_{-f}$, 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**

```c
RealConvolution(NumPoints, XReal, XImag, HReal, Error, Radix);
```
**ComplexCrossCorrelation**

**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

\[
y_m = \sum_{n=0}^{N-1} x_n h_{m+n} \quad m = 0, 1, \ldots, N-1
\]

where subscripts are taken modulo \( N \) (circular correlation). The basic theorem that allows us to calculate these effectively using FFTs is as follows:

\[
Y_m = X_m H_{N-m} \quad m = 0, 1, \ldots, N-1
\]

where capital letters indicate the transforms of the functions represented by lowercase 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 \( C_m = X_m H_m \), 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; TRUE equals autocorrelation; FALSE equals crosscorrelation
- **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 cross-correlation)
- **HImag**: TNvectorPtr; Pointer to imaginary values of the second set of data points (for cross-correlation)
- **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 $X_f = X_{-f}$, 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 \( X_{Real} \) and \( H_{Real} \) (or the autocorrelation of \( X_{Real} \) if \( Auto = \) TRUE)

**XImag** : TNvectorPtr;  
Pointer to imaginary values of the correlation of \( X_{Real} \) and \( H_{Real} \) (or the autocorrelation of \( X_{Real} \) if \( Auto = \) TRUE)

**Error** : Byte;  
0: No errors  
1: \( NumPoints < 2 \)  
2: \( NumPoints \) not a power of two [four]

### Syntax of the Procedure Call

\[
\text{RealCorrelation}(\text{NumPoints}, \text{Auto}, \text{XReal}, \text{XImag}, \text{HReal}, \text{Error}, \text{Radix});
\]

### Comments

If you are performing an autocorrelation of the vector \( X_{Real} \), then set \( Auto \) equal to TRUE. In this case, the vector \( H_{Real} \) 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 \textit{ComplexFFT}, \textit{ComplexConvolution}, and \textit{ComplexCrossCorrelation} expect data to be in complex form. A data file containing a four-point complex square wave could look like this:

\[
\begin{align*}
0 & \ 0 \\
1 & \ 1 \\
1 & \ 1 \\
0 & \ 0
\end{align*}
\]
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
0
0
0
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
0
0
0
0
0
0
0
0
0
0
```

Turbo Pascal Numerical Methods Toolbox
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 | -3.5668238105970e-17 |
| 7.53417436515731e-1 | 4.55292313916419e-17 |
| 5.96901852132470e-2 | -2.6928947846945e-16 |
| -4.2677695926637e-1 | -2.87492515628133e-20 |
| 2.89883706652938e-1 | -3.48529572466949e-16 |
| 6.207520331860e-2 | 1.09874847931145e-16 |
| -2.6665595906343e-1 | 2.77190700484791e-18 |
| 1.7677695926637e-1 | 0.00000000000000e+0 |
| 6.63840517512571e-2 | 9.35764180659936e-17 |
| -2.08522329739913e-1 | 1.09798183260311e-16 |
| 1.27160952826887e-1 | -1.28663728588383e-16 |
| 7.32230407033631e-2 | 1.05413922396982e-19 |
| -1.83841625879619e-1 | -4.96595405328328e-17 |
| 1.00135954077543e-1 | 4.53375697145565e-17 |
| 8.37351650164211e-2 | 5.53423092584155e-17 |
| -1.7677695926637e-1 | 0.00000000000000e+0 |
| 8.37351650164211e-2 | -5.53423092584155e-17 |
| 1.00135954077543e-1 | -4.53375697145565e-17 |
| -1.83841625879619e-1 | 4.96595405328328e-17 |
| 7.32230407033631e-2 | -1.05413922396982e-19 |
| 1.27160952826887e-1 | 1.28663728588383e-16 |
| -2.08522329739913e-1 | -1.09798183260311e-16 |
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 1.68143873236862e-16
9.23818927547367e-16 2.37407332761055e-16
1.00487796352727e-15 -0.00000000000000e+0
3.31468256684932e-16 2.37751564484846e-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

Turbo Pascal Numerical Methods Toolbox
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:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.9445364826301e+0</td>
<td>-1.86068547103024e-18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.76776695296637e+0</td>
<td>-7.31524706015784e-17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.59099025766973e+0</td>
<td>-1.20291519030913e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.41421356237310e+0</td>
<td>-2.28393637498843e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.23743686707646e+0</td>
<td>-1.08420217248551e-18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.06066017177982e+0</td>
<td>-3.11917400601706e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.83883476483185e-1</td>
<td>-2.67794627815503e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.01706781186548e-1</td>
<td>-3.24279577877371e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3033008589911e-1</td>
<td>1.54428189954649e-19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5355390593273e-1</td>
<td>-2.64227183800926e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.7677695296636e-1</td>
<td>-2.67190893532684e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-7.13134768099437e-16</td>
<td>-1.92763731728663e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-8.91610121801382e-16</td>
<td>1.08420217248551e-18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6.53642983532122e-16</td>
<td>-1.0877980600795e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6.24203749931802e-16</td>
<td>-1.18700727111104e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-7.13441426782774e-16</td>
<td>-1.28557069905047e-17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4.75627617855183e-16</td>
<td>1.55182909112094e-18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-7.13364762111940e-16</td>
<td>1.34977736087408e-17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6.24054020590133e-16</td>
<td>1.2012888705040e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6.54026306886293e-16</td>
<td>1.09467566867339e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-8.91610121801382e-16</td>
<td>1.08420217248551e-18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-7.1374800546611e-16</td>
<td>1.92996121512129e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.7677695296636e-1</td>
<td>2.6795258141376e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.53555390593273e-1</td>
<td>2.64586549445461e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3033008589911e-1</td>
<td>1.54428189954649e-19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.01706781186548e-1</td>
<td>3.2388180793764e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.83883476483185e-1</td>
<td>2.67353523858557e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.06066017177982e+0</td>
<td>3.11689802360167e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.23743686707646e+0</td>
<td>-1.08420217248551e-18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.41421356237310e+0</td>
<td>2.27701259690372e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.59099025766973e+0</td>
<td>1.1853096785231e-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.7677695296637e+0</td>
<td>7.25487363187593e-17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
1
2
3
```
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:**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.16672618895780e+1</td>
<td>-0.00000000000000e+0</td>
<td></td>
</tr>
<tr>
<td>1.14904851942814e+1</td>
<td>5.3975947343382e-16</td>
<td></td>
</tr>
<tr>
<td>1.11369318036881e+1</td>
<td>-1.17756934401283e-16</td>
<td></td>
</tr>
<tr>
<td>1.06066017177982e+1</td>
<td>-6.2347103223460e-16</td>
<td></td>
</tr>
<tr>
<td>9.8999493661167e+0</td>
<td>-0.00000000000000e+0</td>
<td></td>
</tr>
<tr>
<td>9.01561146012848e+0</td>
<td>-1.63431828667510e-15</td>
<td></td>
</tr>
<tr>
<td>7.95495128834866e+0</td>
<td>-1.33473191922288e-15</td>
<td></td>
</tr>
<tr>
<td>6.71751442127220e+0</td>
<td>-2.27837818635295e-15</td>
<td></td>
</tr>
<tr>
<td>5.30330085989911e+0</td>
<td>-0.00000000000000e+0</td>
<td></td>
</tr>
<tr>
<td>3.71231060122937e+0</td>
<td>-2.0562661547234e-15</td>
<td></td>
</tr>
<tr>
<td>1.94454364826300e+0</td>
<td>-1.87280291147249e-15</td>
<td></td>
</tr>
<tr>
<td>-4.4896365339059e-15</td>
<td>-1.68784939308506e-15</td>
<td></td>
</tr>
<tr>
<td>-4.65630544778407e-15</td>
<td>-0.00000000000000e+0</td>
<td></td>
</tr>
<tr>
<td>-3.65659814010651e-15</td>
<td>-1.09893055790468e-15</td>
<td></td>
</tr>
<tr>
<td>-3.47045631932115e-15</td>
<td>-1.31403245809765e-15</td>
<td></td>
</tr>
<tr>
<td>-3.85439299085867e-15</td>
<td>-2.51249292491279e-16</td>
<td></td>
</tr>
<tr>
<td>-2.87891171916470e-15</td>
<td>-0.00000000000000e+0</td>
<td></td>
</tr>
<tr>
<td>-4.57044101644980e-15</td>
<td>-9.3300944051833e-17</td>
<td></td>
</tr>
<tr>
<td>-3.77067517030775e-15</td>
<td>1.17756934401283e-16</td>
<td></td>
</tr>
<tr>
<td>-4.08070709916113e-15</td>
<td>2.09754539402286e-16</td>
<td></td>
</tr>
<tr>
<td>-5.8476432354705e-15</td>
<td>-0.00000000000000e+0</td>
<td></td>
</tr>
<tr>
<td>-3.69997138562321e-15</td>
<td>6.25142892149520e-16</td>
<td></td>
</tr>
<tr>
<td>1.76776695256633e-1</td>
<td>1.33473191922288e-15</td>
<td></td>
</tr>
<tr>
<td>5.30330085899907e-1</td>
<td>1.11725341423405e-15</td>
<td></td>
</tr>
<tr>
<td>1.00666017177982e0</td>
<td>-0.00000000000000e+0</td>
<td></td>
</tr>
<tr>
<td>1.76776695296637e0</td>
<td>1.60413157253415e-15</td>
<td></td>
</tr>
<tr>
<td>2.65165042944955e0</td>
<td>1.87280291147249e-15</td>
<td></td>
</tr>
<tr>
<td>3.71231060122937e0</td>
<td>2.10153195690623e-15</td>
<td></td>
</tr>
<tr>
<td>4.9497474830584e0</td>
<td>-0.00000000000000e+0</td>
<td></td>
</tr>
<tr>
<td>6.36396103067893e0</td>
<td>2.10810595243026e-15</td>
<td></td>
</tr>
<tr>
<td>7.95495128834866e0</td>
<td>1.31403245809765e-15</td>
<td></td>
</tr>
<tr>
<td>9.72271824131503e0</td>
<td>1.4123746461018e-15</td>
<td></td>
</tr>
</tbody>
</table>

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:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.16672618895780e+1</td>
<td>-0.00000000000000e+0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.72271824131503e+0</td>
<td>-1.39794194032565e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.95495128834866e+0</td>
<td>-1.30705597305174e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.36396103067893e+0</td>
<td>-2.10114863335206e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.94974746830583e+0</td>
<td>-0.00000000000000e+0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.71231060122937e+0</td>
<td>-2.09953867546454e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.65165042949455e+0</td>
<td>-1.87232375727978e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.7677695296637e+0</td>
<td>-1.6100558602067e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.06066017177982e+0</td>
<td>-0.00000000000000e+0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.30330085889907e-1</td>
<td>-1.12643400856644e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.7677695296632e-1</td>
<td>-1.34178506893962e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4.54836159124956e-15</td>
<td>-6.36393432594434e-16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6.31870217015218e-15</td>
<td>-0.00000000000000e+0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3.84059335010852e-15</td>
<td>-2.0870916673240e-16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2.9141774677842e-15</td>
<td>-1.27416682926388e-16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3.80256767337477e-15</td>
<td>8.09195600654651e-17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2.82248652143076e-15</td>
<td>-0.00000000000000e+0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3.79950108654140e-15</td>
<td>2.39002111325521e-16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3.67714427189007e-15</td>
<td>1.30705597305174e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3.7789549575784e-15</td>
<td>1.08963496666604e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4.4158504004812e-15</td>
<td>-0.00000000000000e+0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4.31192774639698e-15</td>
<td>1.67956960863497e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1.94454364826300e+0</td>
<td>1.87232375727978e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-3.71231060122937e+0</td>
<td>2.05122076667131e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-5.30330085889911e+0</td>
<td>-0.00000000000000e+0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6.71751442127220e+0</td>
<td>2.28537383756575e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-7.95495128834866e+0</td>
<td>1.34178506893962e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-9.01561146012849e+0</td>
<td>1.6479079958046e-15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-9.8994936611674e+0</td>
<td>-0.00000000000000e+0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1.06066017177982e+0</td>
<td>6.40839983502816e-16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1.11369318036881e+0</td>
<td>1.27416682926388e-16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1.14904851942814e+0</td>
<td>-5.22124740716106e-16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 (1024 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


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.


The next three texts are excellent for learning numerical analysis, emphasizing the mathematical theory underlying the algorithms in this toolbox.


Index
Adams-I.pas, 164–165
Adams-Bashforth/Adams-Moulton predictor-corrector method, 150, 162–165
Adams-Bashforth formula, 162
Adams-Moulton formula, 162
Adapgaus.pas, 97
Adapsimp.pas, 92–93
Adaptive schemes, 83
quadrature, 91–97
Aliasing, 232
Backward substitution, 109, 115
Basis vectors, 216
Bisection method, 13
root of a function using, 16–18
Bisect.pas, 17–18
Boundary value problems, 149–152
using Linear Shooting/Runge-Kutta methods, 209–213
using Shooting/Runge-Kutta methods, 202–208
Chebyshev polynomials, 218
ComplexConvolution, 227, 230–231
application, 239–240
ComplexCrossCorrelation, 227, 231
application, 242–243
ComplexFFT, 227, 230
application, 237–238
Compiler directives, 11
Convergence, rate of, 13–14
Cooley-Tukey method, 227–228
Cube_Cla.pas, 55–58
Cube_Fre.pas, 49–52
Cubic spline methods clamped, 40, 53–58
free, 40, 48–52, 60, 71–74
Cyclic Jacobi method, 128, 144–148
and Newton-Horner, 26–30
of a matrix, 128, 139–143
Deriv2fn.pas, 79–81
Deriv2.pas, 68–70
Derivative, 59
approximation of, 60–81
Derivfn.pas, 76–77
Deriv.pas, 63–65
Determinant of a matrix, 101–105
Det.pas, 104–105
Diagonally dominant, 121
Diagonal matrix, 127
Differential equations
first-order, 153–165
coupled, 180–187
linear, 149
nth order, 151, 172–179
ordinary, 149
second-order, 152, 166–171, 186, 202–213
coupled, 190–201
stiff, 155
systems of, 151
Direct factorization of matrices, 102, 115–120
Dirfact.pas, 117–120
Distribution disks, 7–10
Divdif.pas, 46–47
Eigensystem, 144
Eigenvalue, 127–128
Eigenvector, 127
Expo, 219–220
Fast Fourier Transform, 227
algorithms, 234–236
applications, 237–253
sample program, 245–253
FTTDemo, 255, 258
FFTProgs.pas, 245–253
Forward substitution, 115
Forward transform, 228
Fourier, 219
Fourier series, 219
in graphics programs, 256
Fourier transform, 227–253
in graphics demo, 258–259
G
Gauselim.pas, 110–111
Gaussian elimination, 102
with partial pivoting, 102, 112–114
Gaussian quadrature, 83
using Legendre polynomials, 94–97
Gaussidl.pas, 123–125
Gauss-Jordan elimination, 106
Gauss-Seidel iterative method, 102, 121–125
Goodness of fit, 216
Graphics
demo programs, 10, 256–260
rebuilding, 260
I
Initial value problems, 149–151
Adams-Bashforth/Adams-Moulton, 162–165
Runge-Kutta order five, 157–161
Runge-Kutta order four, 153–161, 166–201
Installation, 8–10
Integration, 83–100
Interdrv.pas, 72–74
Interpolation, 39, 152
cubic splines, 48–58
Lagrange polynomials, 41–44
Newton’s divided-difference method, 45–47
Inverse of a matrix, 101–102, 106–108
Inverse.pas, 107–108
Inverse power method, 127–128, 133–138
Inverse transform, 228
InvPower.pas, 135–138
Iterative methods, 13
cyclic Jacobi, 144–148
Gauss-Seidel, 102
J
Jacobi.pas, 146–148
L
Lagrange method, 39, 41–44
Lagrange.pas, 42–44
Laguerre.pas, 37–38
Laguerre’s method, 14
finding roots of complex polynomial, 35–38
Least.pas, 221–225
Least-squares approximation, 216–225
Least-squares solution
graphics demo, 256–257
linear regression, 215
multiple regression, 215
Linear equations, 101–102
differential, 149
with direct factoring, 115–120
with Gaussian elimination, 109–114
Linshot2.pas, 211–213
Lipshitz condition, 151
Log, 220
LSQDemo, 255–256
LU_Decompose, 115–116
LU_Solve, 115–117
M
MakeSinCosTable, 235–236
Matrix
algebra, 101
diagonal, 127
direct factorization, 102, 115
identity, 135
nonsingular, 102, 115–120
orthogonal, 130
permutation, 115
rotation, 144
square, 102, 127–129, 133
symmetric, 128, 144–148
Mesh points, 150
Muller.pas, 33–34
Muller’s method, 14
finding roots of complex function, 31–34
N
Newtdefl.pas, 28–30
Newton-Horner method, 13–14
with deflation, 26–30
Newton-Raphson method, 13–14
root of a function using, 19–22
Newton’s general divided-difference algorithm, 39, 45–47
Nonlinear shooting method, 152, 202–208
Numerical differentiation, 59–60
five-point formulas, 60, 61–70
three-point formulas, 60, 61–70
two-point formulas, 60, 61–65
Nyquist frequency, 232, 258
Partial pivoting, 102, 112
   and direct factoring, 115
Partpivt.pas, 113–114
Poly, 218
Polynomials
   Lagrange, 85
   Legendre, 94–96
   methods to approximate roots of, 14, 23–38
Power, 219
Power method, 127–132
   and Wielandt’s deflation, 139–143
Power.pas, 131–132
Powers-of-four, 228
Powers-of-two, 228

R
Raphson.pas, 10, 20–22
RealConvolution, 227, 231
   application, 240–241
RealCrossCorrelation, 227, 231–232
   application, 244–245
RealFFT, 227, 230
   application, 238–239
Richardson extrapolation
   and numerical integration, 60, 75–77
   and Romberg method, 84, 98–100
   numerical integration, 60, 75–77
RKF_1.pas, 159–161
Romberg algorithm, 98–100
Romberg method, 84
   using trapezoidal rule, 98–100
Romberg.pas, 99–100
Roots of an equation, 13–38
Rotation matrix, 144–148
Runge-Kutta-Fehlberg, 150, 157–161
Runge_1.pas, 155–156
Runge_2.pas, 169–171
Runge-Kutta methods, 150
   fifth-order, 157
   fourth-order, 150–151, 153–161, 166–179
Runge_N.pas, 176–179
Runge_S1.pas, 185–189
Runge_S2.pas, 196–201

S
Secant method, 14
   in nonlinear equations, 152, 202–208
   root of a function using, 23–25
Secant.pas, 24–25
Shoot2.pas, 205–208
Shooting method, 152
   linear, 209–213
   nonlinear, 202–208
Simpson.pas, 86–87
Simpson’s method, 83–93
Splines (see Cubic spline methods)
System requirements, 3

T
TestForRoot, 15
Testlnput, 234, 236
TNArraySize, 11
TNcomplex, 31–32
TNCmpVector, 35–36
TNIntVector, 26–27
TNNearlyZero, 14–15
TNTargetF, 16
TNvector, 10
Trapezoid composite rule, 88–90
Trapezoid method, 83–84
Trapzoid.pas, 89–90
Turbo Pascal, 1–3
   rebuilding with, 260

U
Unpack, 8

W
Wielandt.pas, 141–143
Available at better dealers nationwide.
To order by credit card, call (800) 255-8008; CA (800) 742-1133;
CANADA (800) 237-1136.
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)

Minimum system configuration: Macintosh 512K or Macintosh Plus with one disk drive.
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 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
- menus
- resources and resource files
- desk accessory support
- QuickDraw
- dialogs
- events
- File Manager
- windows
- debugging
- controls

**Suggested Retail Price:** $69.95

Minimum system requirements: Any Macintosh with at least 512K of RAM. Requires Turbo Pascal.

Borland and Turbo Tutor are registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders. Copyright 1987 Borland International. BOR 0381
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)

**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.
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.

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

Suggested Retail Price: $99.95*  
(not copy protected)
Borland Software
ORDER TODAY

To Order By Credit Card, Call (800) 255-8008

In California call (800) 742-1133
In Canada call (800) 237-1136
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.

3. SlideKick*: 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.

*Available Third Quarter 1987.
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.

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.


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.

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.

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.

Productivity

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.
Turbo Pascal, Turbo C, Sprint, and Reflex are 4 of our famous products.

The other 20 are inside . . .
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.)

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.

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.)

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.

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.
Programming Development

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.)

All Borland products are trademarks or registered trademarks of Borland International, Inc. or Borland/Analytica, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders. Copyright 1987 BOR 0291
If you’re looking for software that outperforms anything else on a disk, at a price that beats everything else on a disk, you’re looking to the right people! Borland International. And here’s why:

Our products are guaranteed to perform better than anything else. They’re packed with more speed and power than you’ll find anywhere else. And they come with a friendly user-interface design that’ll get you started right away. They are products built with a long-standing commitment to Quality, Speed, Power and Price!

We invented RAM-resident desktop organizers with SideKick. We set the Pascal and Prolog language standards worldwide with Turbo Pascal and Turbo Prolog. We changed the way people look at data with Reflex: The Database Manager. We also introduced the concept of not copy-protected software.
Turbo Pascal Numerical Methods Toolbox for the Macintosh implements the latest high-level mathematical methods to 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, determinants, 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
2. Exponential
3. Logarithm
4. 5-term Fourier
5. 5-term Polynomial

They’re all ready to compile and run as is.

Minimum system requirements: Macintosh 512K, Macintosh Plus, SE, or II, with one 800K disk drive (or two 400K).

All Borland products are trademarks or registered trademarks of Borland International, Inc. or Borland/Analytica, Inc. Macintosh is a trademark licensed to Apple Computer, Inc. Copyright 1987 Borland International. A Borland Turbo Toolbox product.